Universidade de Évora

Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Bases de dados emergentes: hiper-dimensão, dados não estruturados, alta performance

José Marques

Orientador
Professor Luís Fernando Arriaga da Cunha

Évora, Fevereiro 2013

Mestrado em Engenharia Informática

Bases de dados emergentes: hiper-dimensão, dados não estruturados, alta performance

José Marques

Orientador
Professor Luís Fernando Arriaga da Cunha

Sumário

Para quem precisar instalar uma solução de *storage* que exija elevados desempenhos, com alta disponibilidade, com capacidade em escalar e com custos aceitáveis, nunca descurando a possibilidade do suporte de dados estruturados, semi-estruturados ou não estruturados, que opções tem?

Investigaram-se os motores de bases de dados emergentes, que respondem aos desafios dos nossos tempos, e comparou-se um deles (o VoltDB) com um motor tradicional. Elaborou-se um banco de testes para ambas as tecnologias em igualdade de circunstâncias, para além de testes específicos de escalabilidade e tolerância a falhas, destinados a esta plataforma inovadora.

Reunindo tecnologias já existentes por sistemas NoSql, apetrechando-as com um interface sobejamente conhecido dos programadores, o VoltDB demonstra que é possível fazer mais e melhor com menos recursos. Num dos testes realizados, o VoltDB realizou 225 milhões de transações em 22 minutos, em vez dos 10 dias, que previsivelmente um motor de bases de dados tradicional necessitaria, com o hardware utilizado.

Categorias e Descritores de Assunto: D.4.2 [Operating Systems]: Storage Management; D.4.5 [Operating Systems]: Reliability; D.4.2 [Operating Systems]: Perfomance

Palavras-chave adicionais: Sistemas Distribuidos, OldSql, NoSQL, NewSql, Key-value Store, Bigtable, Dynamo, Cassandra, Eventually Consistent, VoltDB, híper-dimensão, MySql OldOLTP, NewOLTP

Emerging databases: hyper-dimension, unstructured data, high performance

Abstract

What options are available for those who need to install a data storage solution requiring high performance, high availability, capability to scale and having reasonable costs, without neglecting the possibility to support for structured, semi-structured or non-structured data?

Emerging database engines that respond to the challenges of our times were analysed and one of them (VoltDB) was compared with a traditional engine. A test bench was developed for both technologies on equal terms as well as specific tests of scalability and fault tolerance for the innovative platform.

By putting together preexisting technologies in NoSQL systems and equipping them with a well known interface among programmers, the VoltDB demonstrates that it is possible to do more and better with fewer resources. In one of the tests we performed, VoltDB accomplished 225 million transactions in 22 minutes instead of the 10 days that traditional databases engine predictably would require with the hardware we used.

Categories and Subject Descriptors: D.4.2 [Operating Systems]: Storage management; D.4.5 [Operating Systems]: Reliability, D.4.2 [Operating Systems]: Performance

Additional Keywords: Distributed Systems, OldSql, NoSQL, NewSql, Key-Value Store, Bigtable, Dynamo, Cassandra, Eventually Consistent, VoltDB, Hyper-dimension, MySql OldOLTP, NewOLTP

Agradecimentos

Agradeço à minha familía pelo apoio e incentivo que me deram durante a elaboração deste trabalho e ao meu orientador o Prof. Luís Arriaga pela sua sempre disponibilidade e estimulo.

Acrónimos

2NF Segunda Forma Normal

3NF Terceira Forma Normal

4NF Quarta Forma Normal

ACID *Atomicity, Consistency, Isolation, Durability*

BASE *Basically Avaiable, Soft state, Eventual consistency*

BCNF *Boyce-Codd* Forma Normal

CAD *Computer-Aided Design*

CAP *Consistency, Availability, Partition tolerance*

COBOL *Common Business Oriented Language*

CODASYL *Comittee on Data Systems Languages*

CRM *Customer Relationship Management*

DBA *Database Administrator*

DBMS *Database Management System*

DML *Data Manipulation Language*

DNS *Domain Name System*

ERP *Enterprise Resource Planning*

ER *Entity-Relationship*

ETL *Extraction Transforming Loading*

GEM *General Entity Manipulator*

GIS *Geographic Information Systems*

IBM *International Business Machines*

IEEE *Institute of Electrical and Electronics Engineers*

IMS *Information Management System*

IR *Information Retrieval*

IVR *Interactive Voice Response*

JSON *JavaScript Object Notation*

KISS *Keep It Stupidly Simple*

MIT *Massachusetts Institute of Technology*

MMDB *Main Memory Data Base*

NTP *Network Time Server*

OLTP *Online Transaction Processing*

OODBMS *Object Oriented Relational Database Management System*

OQL *Object Query Language*

ORDBMS *Object-Relational Database Management System*

ORDB *Object-Relational Database*

RDBMS *Relational Database Management System*

SDM *Semantic Data Base*

SGML *Standard Generalized Markup Language*

SLA *Service Level Agreements*

UDF *User-Defined Functions*

UNIVAC *Universal Automatic Computer*

W3C *World Wide Web Consortium*

XML *Extensible Markup Language*

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Com a disponibilização da Internet, surgiram os modelos de negócio que operam à escala global. É disso exemplo o Facebook que contava em fevereiro de 2012 com 845 milhões de utilizadores em 70 línguas (*Suposicao de crescimento facil do Facebook e questionada, diz analista* 2012), no que configura um sistema de hiper-dimensão.

Paralelamente, cada vez mais, têm surgido necessidades de tratamento de dados não estruturados, onde o *schema first*[1] não se aplica, nomeadamente dados na forma de documentos, imagens, vídeo e outros formatos multimédia que, apesar de poderem ser processados com o *Object-Relational Database* (ORDB), mas com performances em muitos casos não satisfatórias.

Se a utilização de dados não estruturados pode ser alvo de discussão sobre se o modelo relacional é o mais correto ou não, já a utilização de bases de dados de hiper-dimensão levanta um problema bem claro de escalabilidade que dificulta a utilização do *Relational Database Management System* (RDBMS).

O caminho traçado nos anos mais recentes para contornar as limitações dos sistemas relacionais tradicionais, tem sido o NoSql, com grandes custos de investimento e manutenção de aplicações, à semelhança do que se passava décadas atrás, com as aplicações a gerirem o acesso aos dados, por não haver uma linguagem de alto nível, que facilitasse o trabalho das aplicações. O panorama promete mudar com a chegada do NewSql, que preconiza uma continuidade do interface Sql aplicada a tecnologias que permitem disponibilizar bases de dados de hiper-dimensão, com dados não estruturados e alta performance

Procura-se neste estudo aferir se já existem soluções disponíveis de armazenamento de

[1] Definição do esquema de dados antes de qualquer utilização

dados com escalabilidade e funcionalidades suficientes mantendo uma continuidade com o *know-how* acumulado ao longo das últimas décadas. Para isso, foi selecionada a base de dados emergente VoltDB, desenvolvida por Michael Stonebraker, que visa quebrar as limitações dos RDBMS's tradicionais. Seleccionou-se uma base de dados de referência (MySql) para efetuar testes de comparação com o VoltDB. Diversos testes de escalabilidade e *fault-tolerance* foram realizados sobre a inovadora plataforma VoltDB.

No capítulo 2 realiza-se uma revisão da literatura, apresentando a evolução da tecnologia de base de dados desde o seu despontar, passando pelas diferentes propostas de implementações Sql, até às recentes soluções NoSql e NewSql.

No capítulo 3 explica-se a abordagem efetuada para a realização dos testes, bem como a sua descrição. Os testes são desenvolvidos sob a base de dados emergente VoltDB representante dos novos sistemas NewSql. Utiliza-se o MySql para comparar o OldSql com o NewSql.

O capítulo 4 centra-se nos resultados obtidos nos diversos testes realizados, desde os testes comparativos entre as duas tecnologias, passando pelos testes de escalabilidade e *fault-tolerance* da nova tecnologia, não descurando a análise do comportamento ao longo do próprio teste (evolução).

As conclusões são apresentadas no capítulo 5. Neste estudo demonstra-se ser possível escalar uma base de dados para híper-dimensão, sem perder o conforto da linguagem SQL.

As configurações das bases de dados utilizadas são apresentadas nos anexos A e B.

Capítulo 2

Revisão da Literatura

Ao longo dos tempos, as técnicas de armazenamento de dados têm variado de acordo com as necessidades e com a tecnologia disponível no momento. Muitas vezes as propostas para modelos de novos sistemas de armazenamento de dados são uma reedição de soluções já testadas anteriormente, que por motivos vários não tiveram continuidade. Neste capítulo, procura-se abordar sinteticamente os modelos mais marcantes da evolução dos sistemas de armazenamento de dados, desde a era pré-relacional até ao promissor NewSql, não deixando de abordar as variantes do modelo relacional, bem como de sistemas não relacionais.

Para a realização deste estudo, foram efetuadas pesquisas maioritariamente no repositório de dados do *Institute of Electrical and Electronics Engineers* (IEEE) através do IEEExplore, recuperando artigos científicos a partir do ano 2010 preferencialmente. Outras fontes e outras datas foram utilizadas sempre que justificável.

2.1 Perspetiva histórica de "modelos de bases de dados"

2.1.1 Era Pré-relacional

Na era pré-relacional, os programadores tinham que se preocupar não só em implementar a lógica do negócio nas suas aplicações, mas também gerir o acesso aos dados, envolvendo as problemáticas da concorrência, consistência e integridade dos dados. As aplicações desenvolvidas ficavam reféns da plataforma onde eram desenvolvidas, pois a sua portabilidade, esbarrava com uma grande reengenharia.

Com a crescente complexificação dos sistemas, e as constantes adaptações necessárias para responder às exigências do negócio, os sistemas pré-relacionais necessitavam de um grande

```
****************************************************************
*  Copy File for the VSAM Data Set used for the Demo programs.  *
****************************************************************
*         Copyright (C) 1987-2011 SimoTime Enterprises         *
*                      All Rights Reserved                     *
****************************************************************
*              Provided by SimoTime Enterprises                *
*        Our e-mail address is: helpdesk@simotime.com          *
*     Also, visit our Web Site at http://www.simotime.com      *
****************************************************************
 01  VKSD-RECORD.
     05  VKSD-KEY            PIC X(12).
     05  VKSD-NAME.
         10  VKSD-LAST-NAME  PIC X(28).
         10  VKSD-FIRST-NAME PIC X(18).
     05  VKSD-ADDRESS-1      PIC X(48).
     05  VKSD-ADDRESS-2      PIC X(48).
     05  VKSD-CITY           PIC X(18).
     05  VKSD-STATE          PIC X(18).
     05  VKSD-POSTAL-CODE    PIC X(12).
     05  VKSD-CREDIT-LIMIT   PIC S9(7)V9(2)  COMP-3.
     05  VKSD-FILLER         PIC X(305).
**   End-of-Copy File ------------------------------ KSDCPYB2 *
****************************************************************
```

Figura 2.1: Exemplo de definição dum registo em COBOL.
Fonte: http://www.simotime.com/simorec1.htm. Acedido em 20-11-2011

investimento na sua criação e manutenção.

A manutenção, para além de envolver as alterações exigidas pela evolução do negócio, tinha normalmente um grande investimento em "sobrevivência", onde os técnicos asseguravam constantemente a sua operatividade, pela fraqueza que muitas aplicações apresentavam ao nível da consistência e integridade dos dados.

A corrupção dos ficheiros dos dados era uma constante, com a necessidade de demoradas operações de reconstrução. A concorrência gerida pelas próprias aplicações apresentava graves lacunas, que limitava grandemente a disponibilidade dos sistemas.

No caso dos programas escritos em *Common Business Oriented Language* (COBOL), o custo da sua manutenção ascendia a 75% dos custos totais do seu ciclo de vida (Rugaber & Doddapaneni 1993). Na Figura 2.1 exemplifica-se a definição de um registo utilizando a linguagem, COBOL, onde cada ficheiro a nível de sistema de armazenamento não tinha qualquer relação com os restantes. Era a lógica aplicacional que permitia toda e qualquer interação entre os dados. De referir que esta linguagem foi uma das mais utilizadas a nível comercial, na era pré-relacional, tendo resistido até aos dias de hoje em sistemas *legacy*.

Outras formas de organização mais avançadas de *storage* foram surgindo, para simplificar a tarefa dos programadores de aplicações, e para responderem aos crescentes requisitos impostos pelos utilizadores. Dois modelos de bases de dados foram protagonistas ainda na era pré-relacional: o modelo hierárquico, corporizado pelo *Information Management System* (IMS) da *International Business Machines* (IBM), e o modelo *network* desenvolvido pelo grupo de trabalho *Comittee on Data Systems Languages* (CODASYL), o mesmo que desenvolveu o COBOL.

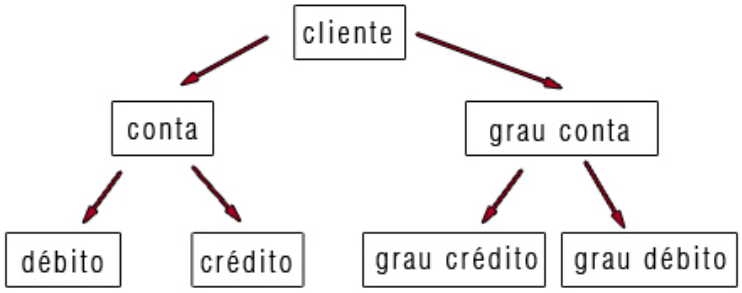


Figura 2.2: Exemplo de base de dados hierárquica, com a sua organização em árvore Fonte: http://img175.imageshack.us/img175/7721/bdhierarquico.jpg. Acedido em 7-01-2012

2.1.2 Hierárquico

As bases de dados hierárquicas viram a luz do dia pela primeira vez em 1968, tendo tido como grande impulsionadora a empresa IBM através do seu produto comercial: IMS, ainda utilizado atualmente.

Basicamente, este modelo é uma coleção de *record types* (registos) de tal forma que cada um (que não seja a *root*) tem um único registo como *parent* e o seu diagrama tem a forma de árvore (Figura 2.2). Um registo é uma coleção de campos de vários tipos de dados, com um subconjunto a constituir a sua chave de acesso.

Este gestor de bases de dados implementou o conceito de independência entre a representação física dos dados e a representação lógica disponibilizada aos programadores de aplicações. Esta abordagem, permite que uma alteração física na estrutura da base de dados, não implique obrigatoriamente uma adaptação das aplicações que serve, já que a representação lógica pode manter-se inalterada. Esta independência é muito importante em qualquer base de dados, já que os ciclos de vida da base de dados e das aplicações são diferentes e uma alteração num lado, não tem que implicar alterações mútuas, reduzindo os custos de manutenção.

Este modelo, apesar de ainda ser utilizado, requer uma complexa implementação, que levou o seu principal mentor (a IBM) a suportar paralelamente outros modelos de bases de dados mais flexíveis. A vantagem de ter a independência entre a representação física e lógica, dos dados, fica ofuscada pela grande restrição do seu modelo em árvore, com o interface para o utilizador a fornecer um registo de cada vez (*record-at-a-time*), que força o programador a otimizar as *queries* manualmente (Stonebraker & Hellerstein 2005).

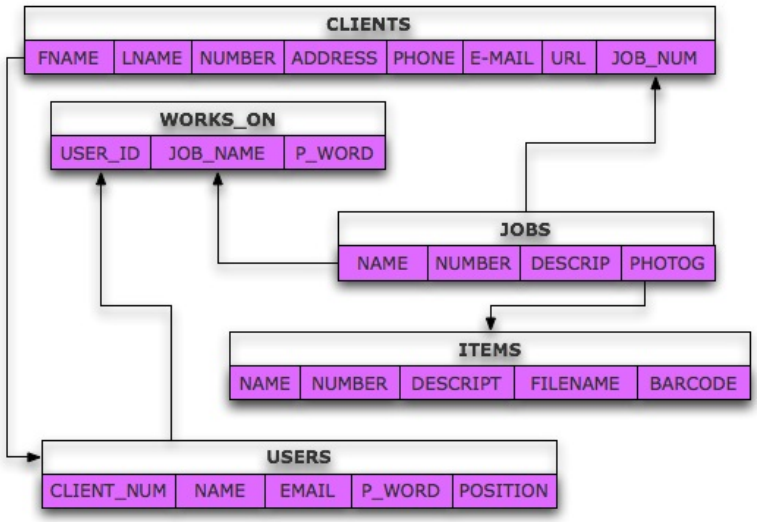


Figura 2.3: Estrutura do modelo de bases de dados *Network* Fonte: http://1.bp.blogspot.com/-ifkbmcp2vUU/Twc9NLmAz2I/AAAAAAAAAEU/RX6gWZKwlwE/s1600/network_model.png. Acedido em 7-01-2012

2.1.3 *Network* (Codasyl)

O grupo CODASYL propôs em 1969 um novo modelo de base de dados, que apresentava algumas inovações face ao modelo hierárquico, como seja a possibilidade de um registo ter vários *parents*, permitindo por exemplo que a tabela *jobs*, dependa das tabelas *clients* e *works_on* conforme representado na Figura 2.3.

Este modelo é mais flexível do que o modelo hierárquico, contudo não deixa de ser um modelo que disponibiliza um registo de cada vez, onde os programadores têm a árdua tarefa de gerir os acessos aos dados através da supervisão do último registo acedido, originando muitos acessos a disco. Apesar de ser um sistema proposto pelo grupo de trabalho CODASYL para resolver os problemas sentidos com o modelo hierárquico, ele não contempla a independência da representação física e lógica, reconhecidamente importante. O modelo bases de dados *Network* CODASYL permite soluções mais flexíveis do que o modelo hierárquico, mas mais complexas de implementar e manter, não existindo utilitários básicos para um sistema deste tipo, como sejam ferramentas de carregamento e de recuperação de dados, o que dificulta o trabalho dos programadores aplicacionais (Stonebraker & Hellerstein 2005).

2.2 Era Relacional

Enquanto o modelo hierárquico da IBM IMS e o modelo *Network* proposto pelo grupo CODASYL, se tentavam impor, Ted Codd propôs o modelo relacional em 1970, que pretendia reduzir o tempo de manutenção das aplicações desenvolvidas sobre o IMS, e melhorar a

independência entre a representação física e lógica dos dados. Para atingir estes objetivos, Codd propôs guardar os dados numa estrutura simples (tabelas), e desenvolver utilitários de alto nível *Data Manipulation Language* (DML) para aceder aos dados, e dispensar os programadores da preocupação da representação física dos dados, que ambos os modelos contemporâneos exigiam. Com esta estrutura, o modelo relacional revelou-se flexível o suficiente para representar quase tudo. Codd foi melhorando de uma forma rigorosa e formal o modelo nos anos seguintes, com a adição do cálculo e álgebra relacional que, pelo facto de o autor ser um matemático, nem sempre era de fácil entendimento para o comum dos mortais. A IBM patrocinou um projeto de pesquisa sobre o modelo relacional no início dos anos 70, o *System* R, com base nas ideias de Codd, desenvolvendo a linguagem de alto nível SQL, entre outras inovações. Apesar deste envolvimento, a gigante informática continuou a comercializar preferencialmente o IMS (News 2003). Durante quase uma década, os três modelos vigentes (IMS, *Network*, Relacional) geraram um debate aceso sobre qual o caminho a seguir, até que, com o surgimento dos minicomputadores VAX, mais económicos, onde empresas como a Oracle e a Ingres apostaram forte com o seu modelo relacional e, adicionalmente, a falta de portabilidade do modelo Network, foi dando vantagem ao modelo relacional. A machadada final no modelo hierárquico e *Network* foi dada pela IBM, ao adotar o modelo relacional em 1984 com o produto DB/2.

Pode-se concluir do debate protagonizado pelos partidários de ambos os modelos, que a independência entre os dados físicos e lógicos funciona melhor com modelos de dados simples do que com modelos complexos, que os grandes *players* do mercado lideram os debates, independentemente da qualidade das soluções tecnológicas e, por último, que a utilização do *query optimizer* é muito mais proveitosa do que o *record-at-a-time* até então utilizado (Stonebraker & Hellerstein 2005).

As bases de dados relacionais vieram simplificar o desenvolvimento das aplicações, pela separação clara entre a lógica aplicacional e a lógica de armazenamento de informação (*storage*), permitindo resolver grande parte dos problemas sentidos pelos sistemas anteriores, de falta de consistência e pouca integridade dos dados. Este novo paradigma veio incrementar largamente a disponibilidade dos sistemas, pela robustez e garantia de funcionamento.

A existência de um interface simples e uma linguagem padrão, o SQL (*Structure Query Language*) forneceu as ferramentas aos programadores aplicacionais para se concentrarem nos modelos de negócio, com a premissa de dispor de um sistema de armazenamento de dados fiável e de fácil utilização (Brito 2010).

A título de exemplo, de acordo com Rugaber & Doddapaneni (1993), um programa em COBOL com 600 linhas podia ser substituído por outro escrito SQL*ReportWriter da Oracle, unicamente com 100 linhas, traduzido num menor esforço de codificação. Os RDBMS oferecem mecanismos que garantem a integridade e consistência dos dados, a tolerância a falhas, a otimização de consultas, a gestão do acesso aos dados, a portabilidade, a normalização, entre outros.

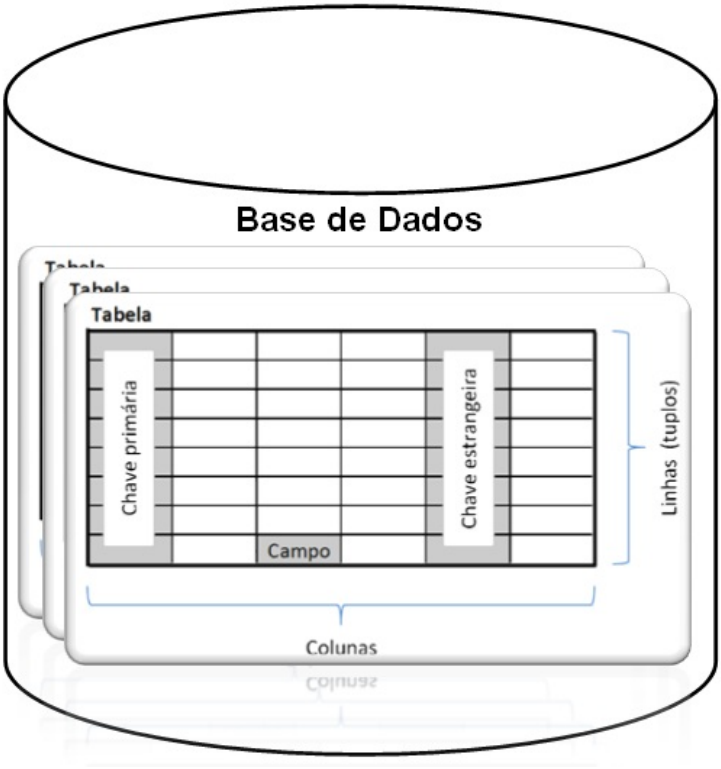


Figura 2.4: Principais componentes de uma base de dados relacional Fonte: Elaboração própria

Uma base de dados organizada de acordo com o modelo relacional tem como principais componentes as tabelas. Cada tabela é composta por linhas (tuplos), e colunas. A interseção entre uma linha e uma coluna é um campo (Figura 2.4).

As bases de dados relacionais caracterizam-se por integridade dos dados e consistência dos dados.

2.2.1 Integridade dos dados

Um dos conceitos básicos do modelo relacional é a relação entre as tabelas que compõem a base de dados, implementado através da chave primária e da chave estrangeira (Figura 2.5). Enquanto a chave primária garante a unicidade do tuplo, a chave estrangeira permite a ligação entre diferentes tabelas da base de dados (Brito 2010). Este mecanismo não permite erros vulgares da era pré-relacional, como por exemplo a inserção numa tabela de vendas de um artigo não existente na tabela de produtos, ou ainda a remoção de um artigo da tabela de produtos deixando os detalhes de vendas desse artigo órfãos.

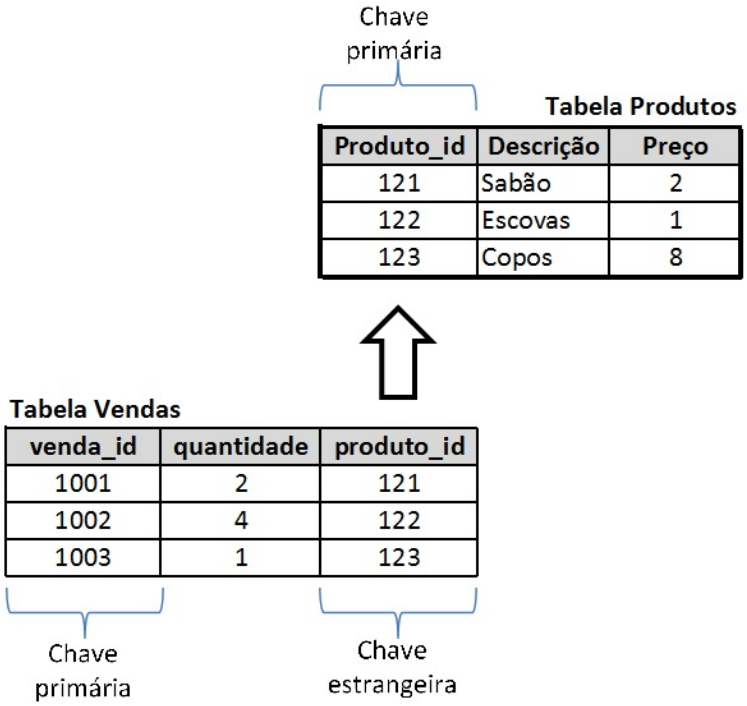


Figura 2.5: A chave estrangeira da tabela de Vendas relaciona-se com a chave primária da tabela de Produtos, obrigando que cada linha inserida em vendas tenha uma correspondência na tabela de produtos através da sua chave primária (produto_id), garantindo assim a integridade dos dados Fonte: Elaboração própria

2.2.2 Consistência dos dados

Todo o peso da gestão da consistência dos dados é da responsabilidade do RDBMS, que tem como grande preocupação as transações *Atomicity, Consistency, Isolation, Durability* (ACID). A denominação do termo ACID teve origem no trabalho de Haerder & Reuter (1983). Os autores preconizaram que numa transação iniciada com um *begin_transaction* e finalizada com um *end_transaction*, seja expectável que as ações nela contidas sejam refletidas na base de dados na íntegra, ou em caso de alguma falha, nenhuma ação seja registada (Figura 2.6). Para responder a estes requisitos, uma transação tem que obedecer às seguintes propriedades:

Atomicity Ou todas as ações da transação são realizadas, ou nenhuma o é.

Consistency Esta propriedade assegura que uma base de dados passa de um estado consistente para outro igualmente consistente, após uma transação ACID.

Isolation As ações de uma transação não podem interferir nas ações de outras transações concorrentes, como por exemplo, a atualização dos mesmos dados em simultâneo por vários utilizadores, se não houvesse *isolation* o resultado seria imprevisível.

Durability Depois de terminada a transação, o sistema tem que garantir que os dados aceites sobrevivem a possíveis falhas posteriores ao *end_transaction*, como por exemplo, falta de ligação entre o cliente e o servidor, entre outras.

2.2.2.1 Tolerância a falhas (fault tolerance)

Perante a ocorrência de falhas diversas, o RDBMS procura recuperar os dados mantendo a sua consistência (Brito 2010), utilizando, por exemplo, a técnica de reconstrução das tabelas a partir dos ficheiros de log. As falhas podem ser de ordem vária como, por exemplo, falha na rede de comunicações, falha nos componentes de hardware, entre outras possíveis.

2.2.2.2 Otimização de consultas

Com um simples comando SQL é possível ao cliente obter os dados pretendidos, que de outra forma, implicaria muitas linhas de código para a realizar. Na Figura 2.7, podemos ver um exemplo de uma consulta onde se pretende obter a marca, a origem, o modelo e ano de fabrico de todos os carros com marcas americanas.

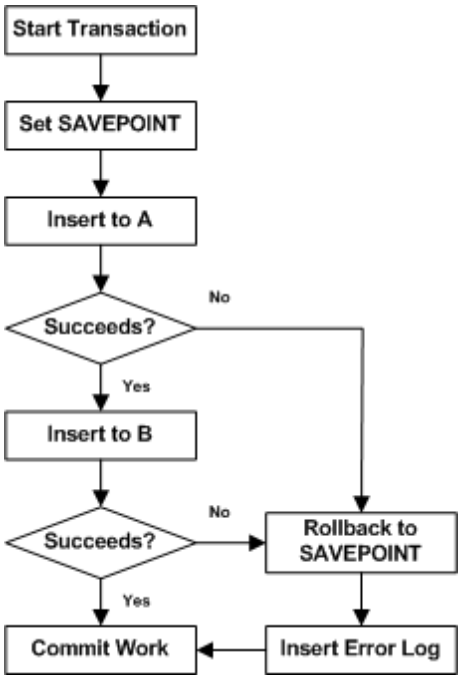


Figura 2.6: Exemplo de transação envolvendo as tabelas A e B. A transação tem início no *start transaction* (*begin_transaction*) e termina no *commit work* (*end_transaction*). Fonte: http://michaelmclaughlin.info/db1/lesson-4-inserting-data/transaction-management/. Acedido em 10/11/2011

```
SELECT m.marca, m.origem, c.modelo, c.ano
FROM marcas as m, carros as c
WHERE m.codMarca = c.codMarca
AND m.origem = 'Americana'
```

[Resultado]

marca	origem	modelo	ano
Ford	Americana	Fiesta	2000
Ford	Americana	Ka	2005
Ford	Americana	Fiesta	2007
GM	Americana	Monza	1995
GM	Americana	Vectra	2006
GM	Americana	Vectra	1999
GM	Americana	Monza	1997

Figura 2.7: *Select* para mostrar a marca, origem, modelo e ano de fabricação de todos os carros com marcas americanas Fonte: http://www.luis.blog.br/exemplos-de-sql-select.aspx. Acedido em 20/11/2011

2.2.2.3 Gestão do acesso aos dados

Estes sistemas contemplam normalmente um controlo de acesso aos dados por utilizador, ao ponto de restringir acessos de *read* ou *write* em tabelas, ou mesmo em determinados atributos (colunas).

2.2.2.4 Portabilidade

Pelo facto de se utilizar uma linguagem estruturada padrão, facilita a migração das aplicações para outras plataformas, como por exemplo a alteração do motor de RDBMS em si, ou ainda a sua mudança, para outro sistema operativo.

2.2.2.5 Normalização

Uma das características da normalização é a separação dos dados em tabelas diferentes, agregados por características comuns, e associados através das suas chaves externas, de forma a minimizar a redundância da informação, com repercussões no espaço ocupado em disco, e na contribuição para a integridade dos dados. Durante a década de 70 foram propostas diversas normalizações como a Segunda Forma Normal (2NF), a Terceira Forma Normal (3NF), a *Boyce-Codd* Forma Normal (BCNF) e a Quarta Forma Normal (4NF) (Stonebraker & Hellerstein 2005).

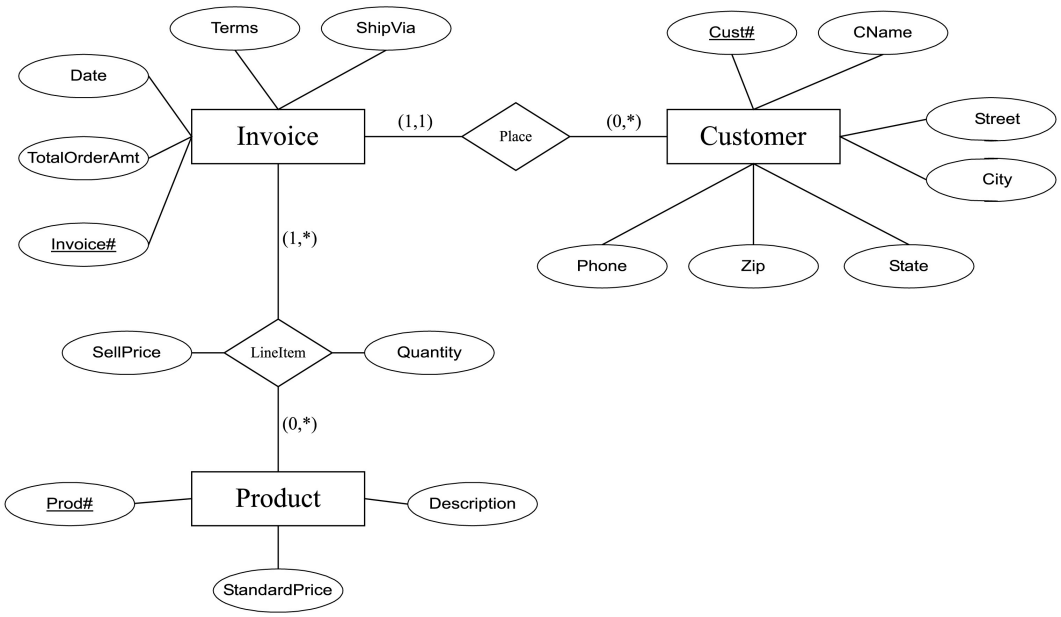
2.3 Outros modelos relacionais

Com o tempo foram surgindo outras propostas de modelos de bases de dados na esfera do relacional, nomeadamente os modelos *entity-relationship, exended relational, object oriented, object relational*, e as implementações de *datawarehouse*.

2.3.1 *Entity-relationship*

Em meados dos anos 70, *Peter Chang* propôs o modelo *Entity-Relationship* (ER), com pretensões a ser uma alternativa aos três modelos em debate da altura (hierárquico, *network*, e relacional). O modelo propunha que os dados se organizassem em torno de entidades, que por sua vez eram constituídas por atributos. Os atributos representam os dados em si, com um ou mais a serem designados como chaves de acesso únicas. As entidades podem estar relacionadas entre si, numa relação de 1-para-1, 1-para-n, n-para-1 ou n-para-m.

O conceito é de difícil implementação prática e, por esse e outros motivos, nunca chegou a ser materializado num *Database Management System* (DBMS). Deixou sim, um grande contributo ao ser aplicado com grande sucesso no desenho de esquemas de bases de dados,



1318

Figura 2.8: Exemplo de um diagrama ER, com a representação das entidades *Invoice, Customer e Product* (retângulos), das relações entre as entidades (losango) e respetivos atributos (elipses). Neste exemplo o "n" é representado pelo "*" Fonte: `http://www.emeraldinsight.com/content_images/fig/0291030302004.png`. Acedido em 8-01-2012

que de uma forma ou de outra, ainda se mantêm em vigor nos dias de hoje (Figura 2.8) (Stonebraker & Hellerstein 2005).

2.3.2 *Extended relational*

Para dar resposta a problemas específicos, onde o modelo relacional base mostrava algumas limitações, como por exemplo nas aplicações *Computer-Aided Design* (CAD), diversos investigadores foram propondo extensões ao modelo base para superar essas mesmas limitações, logo no início dos anos 80. O resultado deste conjunto de propostas designou-se por *Extended Relational Model*. De entre as inovações surgidas, talvez as mais marcantes vieram do trabalho de Zaniolo (1983), com a sua proposta de uma linguagem de bases de dados *General Entity Manipulator* (GEM), que entre outras funcionalidades, trouxe:

Set-valued attributes A possibilidade de um atributo ter um conjunto de possíveis valores como, por exemplo, as cores possíveis para um determinado carro.

Aggregation Permitir referir um atributo de uma determinada tabela em cascata, podendo repetir-se o nome de atributos que pertencem a tabelas distintas. Hoje em dia a sua utilização em SQL é muito semelhante.

```
Retrieve (SUPPLY.Item.Name) where
    {SUPPLY, Dept by SUPPLY.Item} >= {DEPT}
```

Exemplo de comando em linguagem GEM original Fonte: (Zaniolo 1983)

```
Select sp.*, city
    From sp, s
    Where sp.sno=s.sno
```

Exemplo de um comando SQL que utiliza a agregação.

Generalization A generalização introduziu na linguagem de bases de dados o conceito de herança (*inheritance*) de atributos. *Inheritance* permite a reutilização de código, através da herança das características de um objeto e a possibilidade de adicionar novas características (atributos ou métodos).

Apesar das extensões ao modelo relacional trazerem melhorias na formulação de *queries*, elas não trouxeram melhorias significativas de performance.

Novas propostas que não tragam significativas melhorias em performance ou em funcionalidade têm os dias contados, devendo ter-se sempre em conta o princípio *Keep It Stupidly Simple* (KISS) (Stonebraker & Hellerstein 2005).

2.3.3 *Object-Oriented*

Em meados dos anos 80 surgiu o interesse sobre as *Object Oriented Relational Database Management System* (OODBMS)'s, pensadas para a utilização do *persistent* C++, aproveitando a grande popularidade desta linguagem na altura.

A ideia baseava-se na utilização de uma única linguagem para a representação física dos dados, para a representação em memória dos dados, para realização das pesquisas, bem como para a escrita da própria aplicação, se necessário. A grande potencialidade desta estratégia residia no poder que um programador tinha, pelo facto de poder manipular as estruturas nativas de dados, com ganhos de performance.

Esta abordagem permitia a homogeneização de linguagens desde a representação física dos dados, até à aplicação final (Figura 2.9). Se por um lado dava um grande poder aos programadores, por outro exigia que as linguagens de programação lidassem com as funcionalidades das DBMS's, regredindo ao ponto de não haver independência entre a lógica de armazenamento e a lógica aplicacional. Trata-se de uma independência, conquistada desde o fim do CODASYL e que provou ser o caminho a seguir nas últimas décadas.

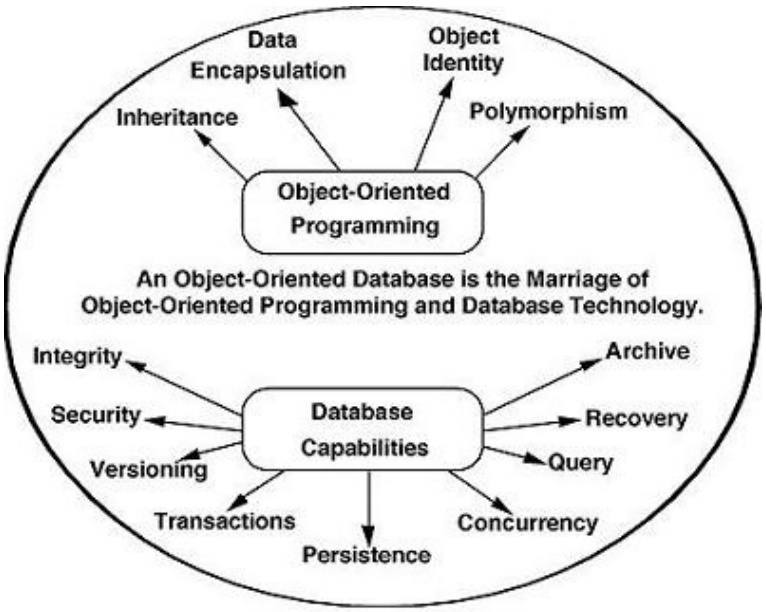


Figura 2.9: Uma OODBMS é um casamento entre uma linguagem orientada a objetos e a tecnologia de base de dados Fonte: http://images.devshed.com/af/stories/Introduction%20to%20RDBMS%20OODBMS%20and%20ORDBMS/1.jpg. Acedido em 8-01-2012

Uma das aplicações deste conceito foi na área do CAD, onde a aplicação carrega para memória um objeto com a representação do desenho de engenharia, modificando-o guardando novamente a estrutura em disco. Este tipo de objetos são normalmente utilizados por um utilizador de cada vez, não necessitando de transações. Os projetos de OODBMS reorientaram a sua atenção para o mercado de CAD, com a manipulação de estruturas de dados em *persistent* C++ e não nas transações necessárias ao processamento de dados de negócio.

Com o OODBMS pode-se aprender que as DBMS têm requisitos de funcionamento muito próprios que exigem soluções muito específicas, assim como as respetivas aplicações. Uma linguagem única em todo o processo, torna-se pouco operacional, e consequentemente as OODBMS's saíram de circulação quase na sua totalidade, restando até há pouco tempo a OODBMS O2, que utiliza a *Object Query Language* (OQL) em vez do SQL, e que de resto, não teve uma grande aceitação global, talvez por não ser uma tecnologia *made in* USA (Stonebraker & Hellerstein 2005).

2.3.4 *Object-Relational*

O conceito ORDB surgiu (1982) com a necessidade de trabalhar as características específicas do *Geographic Information Systems* (GIS). No GIS há a necessidade de trabalhar com coordenadas e efetuar operações diversas sobre elas. As bases de dados de então, ao executarem expressões em SQL sobre índices B-tree unidimensionais, obtinham maus re-

sultados com as aplicações GIS porque os tipos de dados até então existentes não estavam preparados para suportar as características específicas dos dados GIS.

Ao criarem a solução para trabalhar com o GIS, as ORDB's abriram a porta para muitos outros tipos específicos de soluções que necessitam de tipos de dados muito particulares. Assim, surgiu a possibilidade de criação de *user-defined data types*, de *user-defined operators*, de *user-defined functions* e de *user-defined access methods*, e onde um dos maiores impulsionadores deste modelo foi o projeto Postgres.

User-defined data types Possibilita tipos de dados criados pelos utilizadores, por não existir nativamente. Por exemplo um ponto geográfico é composto por dois números reais, a latitude e longitude, e em vez de necessitar de dois atributos para o representar, pode-se criar num novo tipo que contém a latitude e longitude.

User-defined operators Com a possibilidade de criação de tipos de dados personalizados, surge também a necessidade de criação de operadores personalizados para facilitar a sua manipulação. Um operador acaba por ser uma função que recebe argumentos e os processa. Por exemplo, o Postgres suporta operadores personalizados só com um argumento à esquerda ou à direita (*left unary e right unary*), e com argumentos de ambos os lados (*binary operators*).

***User-Defined Functions* (UDF)** Possibilidade de criar funções personalizadas, que podem ser invocadas dentro da linguagem SQL, permitindo criar extensões à própria linguagem de acordo com as necessidades de uma determinada aplicação.

User-defined access methods Funções associadas à instância de um objeto, que quando invocada exerce influência somente sobre essa instância.

Outra forma de implementar as UDF surgiu em meados dos anos 80, através da empresa Sybase que introduziu a possibilidade de implementação de *stored procedures* num motor de bases de dados. As *stored procedures* são funções que podem ser invocadas pelo próprio DBMS em determinadas circunstâncias (*triggers*), como por exemplo antes de um *insert*. As *stored procedures*, ao libertarem os clientes de parte da lógica da aplicação, permitem uma grande poupança em termos de tráfego de rede e de segurança, com o código a ser executado no servidor de base de dados, não estando sujeitas às latências e caprichos das comunicações.

As ORDB foram sendo adotadas gradualmente pelos mercados, com a grande vantagem de permitirem colocar o código no próprio motor de base de dados, com ganhos de performance e segurança. Têm persistido algumas barreiras na sua aplicação, com ausência de standards, já que cada vendedor tem a sua própria linguagem de codificação que dificulta a portabilidade (Stonebraker & Hellerstein 2005).

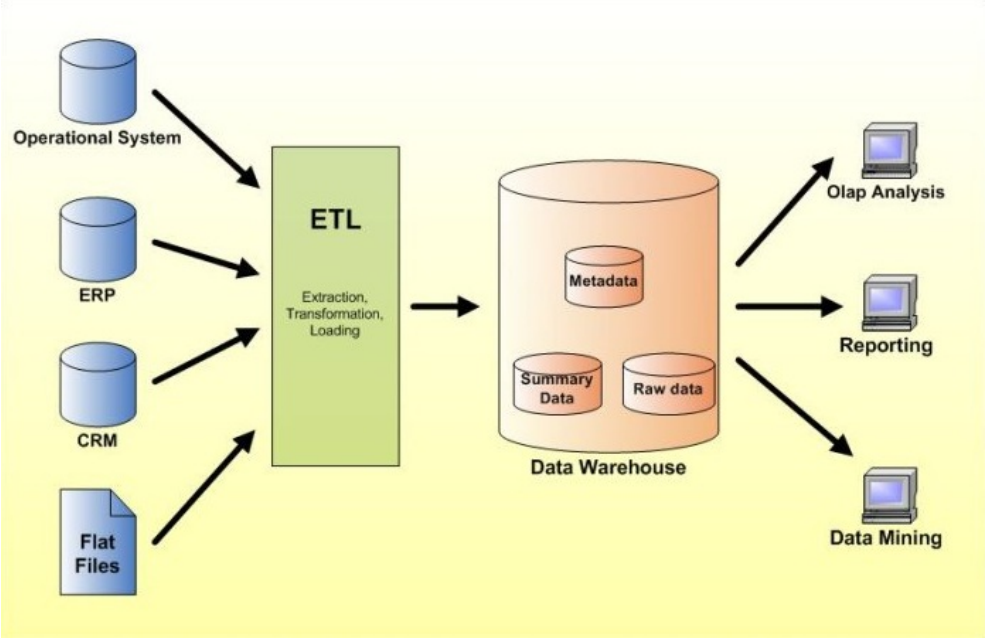


Figura 2.10: Operacionalização genérica de um *datawarehouse* Fonte: http://www.infoescola.com/wp-content/uploads/2010/03/data-warehouse.jpg. Acedido em 10-01-2012

2.3.5 *Datawarehouse*

Datawarehouses são bases de dados utilizadas normalmente para armazenarem dados de histórico, oriundos de diversas plataformas, com a finalidade de apoiarem as decisões de uma organização. Estes sistemas devem ser concebidos de forma a disponibilizem os dados em tempos aceitáveis, a partir de grandes quantidades de informação, respondendo a questões complexas (de Carvalho Costa & Furtado 2007).

Genericamente, os dados são carregados a partir das mais diversas aplicações / plataformas, como seja o sistema *Enterprise Resource Planning* (ERP), o *Customer Relationship Management* (CRM) entre outras. Através de *Extraction Transforming Loading* (ETL), os dados extraídos são transformados e depositados numa base de dados concebida para o efeito, que permitirá efetuar as consultas para apoio à gestão (Figura 2.10)

Existem diversos modelos concetuais para representar a informação dentro num *datawarehouse*, de entre os quais, os esquemas 3NF (*third normal form*), *Star* e *Snowflake* (Lane, Schupmann & Stuart 2005).

3NF O esquema 3NF, baseado na 3ª forma normal, do clássico modelo relacional, que procura reduzir a redundância da informação através da normalização dos dados, organizando-a em entidades bem definidas (Figura 2.11).

Este esquema é utilizado em *datawarehouses* com constantes carregamentos de dados,

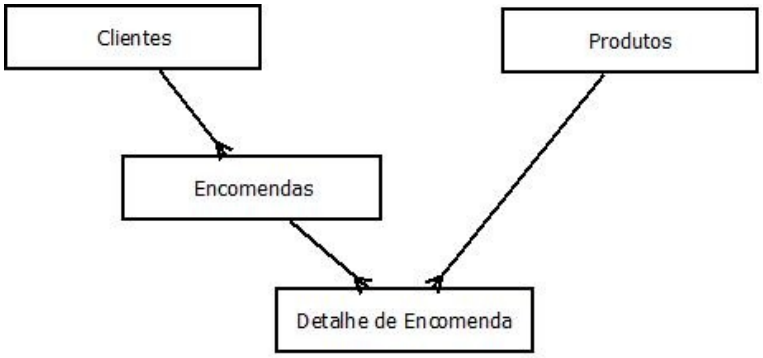


Figura 2.11: Esquema 3NF Fonte: Elaboração própria baseada em (Lane et al. 2005)

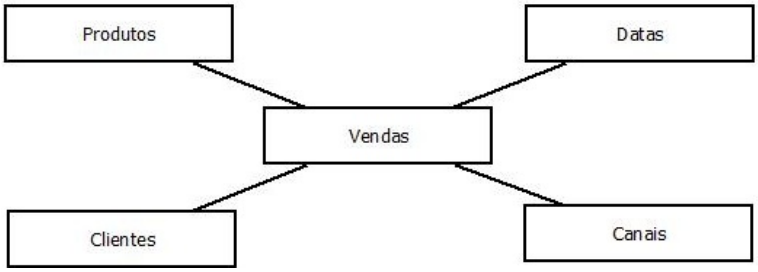


Figura 2.12: Esquema *Star*, com a tabela de factos (vendas) e as tabelas dimensão (produtos, clientes, datas e canais) Fonte: Elaboração própria baseada em Lane et al. (2005)

como seja os *data marts*, e suporta *querys* de longa execução.

As vantagens deste esquema encontram-se na independência da estrutura em relação às estruturas de dados das aplicações que o suportam, bem como a reduzida necessidade de transformações de dados (ETL) , em comparação com outros esquemas.

Star O esquema em *Star* é possivelmente o mais simples e utilizado em *datawarehouse*. No centro da estrela reside uma grande tabela contendo as informações primárias designada de factos, e nas extremidades as tabelas de dimensão, que contêm informações sobre um atributo em particular da tabela de factos (Figura 2.12)

Uma *query* sobre este esquema na forma *star query* é um *join* entre a tabela de factos e um número de tabelas de dimensão. Cada tabela de dimensão é relacionada com a tabela de factos através da chave primária e da chave estrangeira. As tabelas de dimensão não estão relacionadas entre si, permitindo uma grande otimização e eficiência na execução de consultas.

Este esquema permite grandes performances nas consultas *star query*, intuitivas, com um grande número de ferramentas disponíveis, facilitando a sua utilização por parte dos utilizadores e é apropriado para grandes e pequenos sistemas de *datawarehouses*.

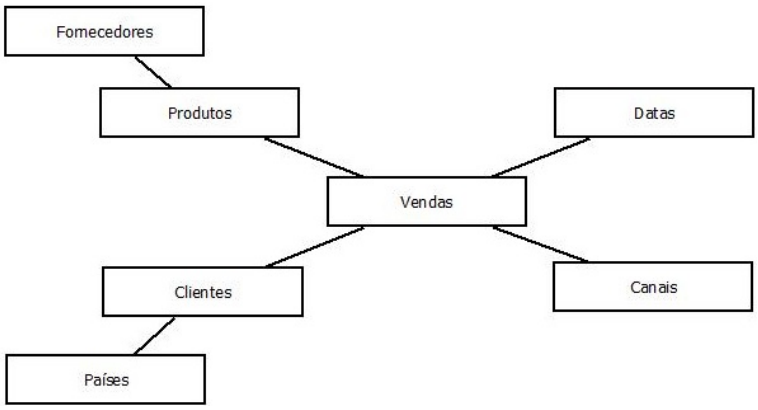


Figura 2.13: Esquema *Snowflake*, com o seu diagrama parecido a um floco de neve Fonte: Elaboração própria baseada em Lane et al. (2005)

Snowflake O esquema *snowflake* (Figura 2.13) é uma variante do esquema em *star*, e tem esta designação, devido ao seu diagrama ser semelhante ao um floco de neve. Neste modelo, as tabelas dimensão são normalizadas de forma a eliminar a redundância, originando várias tabelas dimensão, em vez de uma grande, como no modelo *star*.

Com um número superior de tabelas normalizadas que, se por um lado reduzem o espaço ocupado pelos dados, por outro exigem mais ligações (*joins*) através das suas chaves primárias / estrangeiras para responder às *queries*, resultando numa menor performance do que em relação ao esquema *star*.

2.3.6 *Semantic*

Em meados dos anos 70, surgiu a ideia de lançar um modelo de dados semântico *Semantic Data Base* (SDM), que seria já uma aposta num modelo pós-relacional apesar de o modelo relacional nessa altura ainda não ter vingado plenamente. A SDM aprofunda o conceito de classes, com uma coleção de registos interligados numa determinada configuração. À semelhança da linguagem GEM, utiliza o conceito de agregação e generalização, bem como a noção de conjuntos. A agregação é elevada para um patamar muito alto, onde um atributo de uma classe pode ser um conjunto de instâncias de registos de uma classe. Uma classe pode generalizar outras classes formando um grafo em vez de uma árvore como no GEM. As classes também podem ser uma união, intersecção e diferença entre classes, podendo conter variáveis que também são classes.

Os modelos semânticos propostos demonstraram ser demasiado complexos para serem postos em prática, com a *Universal Automatic Computer* (UNIVAC) a fazer uma implementação experimental, mas que rapidamente abandonou, ao constatar a grande aceitação do SQL, associado ao modelo relacional. Na Figura 2.14, apresenta-se um exemplo de um diagrama desenhado para SDM (Stonebraker & Hellerstein 2005).

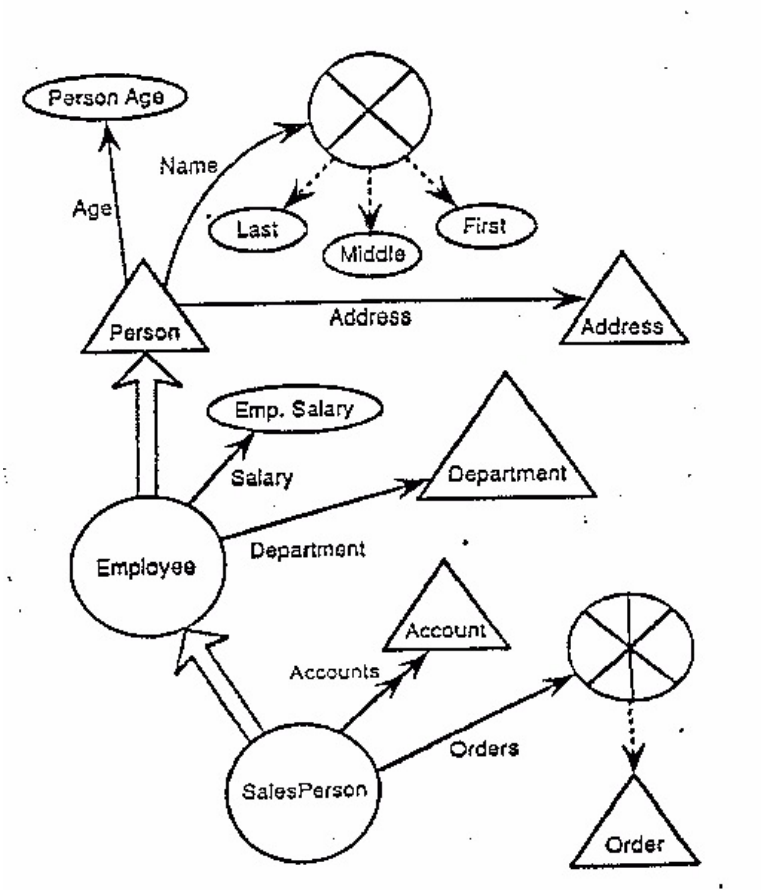


Figura 2.14: Exemplo de um diagrama concebido no modelo semântico Fonte: `http://www.cs.pitt.edu/~chang/156/images/fig1913.gif`. Acedido em 20-01-2012

2.3.7 *Semi Structured Data*

Existem muitos trabalhos e propostas na área dos dados semiestruturados, nomeadamente aqueles baseados no *schema-last* e os baseados no *Extensible Markup Language* (XML).

Schema-Last Com os dados estruturados, o *Database Administrator* (DBA) tem que definir um determinado esquema de dados antes de qualquer utilização (*schema-first*). Todos os dados carregados serão consistentes com o esquema pré-definido, sob pena de serem rejeitados. As propostas no âmbito dos dados semiestruturados vão no sentido do *schema-last*, ou seja, o esquema é definido pela instância dos dados em si, podendo cada instância ser diferente de todas as outras. As instâncias têm que estar acompanhadas de metadados que indicam o significado de cada atributo.

Exemplo de duas instâncias de pessoas:

Instância #1

```
Pessoa
   .Nome:                  Jose
   .Idade:                 40
   .Telefone:              988302061
Fim Pessoa
```

Instância #2

```
Pessoa
   .Nome:                  Maria
   .Data_Nascimento:       28/4/1970
   .Telefone_de_Casa:      945035032
Fim Pessoa
```

Duas instâncias de dados semiestruturados, sobre pessoas, ambas corretas e com informações similares, numa representação de *semantic heterogeneity*. É um verdadeiro desafio comparar os dados destas duas instâncias, devido às suas diferentes estruturas. Neste contexto, qual será a vantagem do *schema-last*?

A estruturação dos dados pode ser classificada de quatro formas (Stonebraker & Hellerstein 2005):

Estrutura rígida dos dados Se os dados têm que estar estruturados de uma forma rígida para manter a consistência entre eles, então o *schema-first* talvez seja a solução mais simples.

Estrutura rígida dos dados com alguns campos de texto livre No caso em que existe uma estrutura rígida dos dados, mas onde há a necessidade de acrescentar informações não estruturada em texto, por exemplo, sob a forma de comentários. Outra vez aqui, o *shema-first* parece ser a solução mais razoável com a utilização de atributos de texto livre disponíveis numa ORDB.

Dados semiestruturados Se as instâncias têm campos diferentes, e são registadas em documentos de texto em que é necessário um processamento para obter as informações relevantes, então o *shema-last* é a solução.

Só texto livre Neste caso, não há estrutura alguma, só texto livre, e estamos a entrar no campo do *Information Retrieval* (IR), e não se pode falar em dados semiestruturados, pois a estrutura pura e simplesmente não existe.

Salvo algumas aplicações pontuais, não existem muitas utilizações para o *schema-last* com dados semiestruturados, apesar de haver muitos ramos de pesquisa em numerosas

universidades. Uma tentativa de utilizar este método, foi efetuada pelo *site monster.com*, especialista em resumos. Esta entidade acabou por abandonar esta abordagem, talvez para forçar a uniformidade dos dados e assim poder facilmente compará-los.

Qualquer aplicação do *schema-last* tem que debater-se com *semantic heterogeneity*, e processar os dados, registo a registo, com grandes custos de processamento envolvidos, pelo que é uma boa ideia evitar a sua utilização.

XML Data Model A ideia do *XMLSchema* é trabalhar com a estrutura de documentos formatados, numa perspetiva de *document markup language*, uma parte do *Standard Generalized Markup Language* (SGML). Os standards nesta área são muito complexos, devido às estruturas diversas e também complexas que os diversos documentos podem conter, e como resultado o *XMLSchema* foi deixado de lado com este propósito. Surgiu então a ideia de criar um gestor de bases de dados, baseado em dados estruturados segundo o *XMLSchema*, com registos XML que podem ser hierárquicos no IMS, com ligações a outros registos como no modelo CODASYL, GEM e SDM e com conjuntos de atributos e herdar as características de outros registos como em SDM. O resultado é o mais complexo modelo de dados alguma vez proposto e está longe do modelo relacional quanto à aplicação do princípio KISS.

Se a ideia for utilizar o XML para guardar dados estruturados, qual é a vantagem em relação ao modelo relacional? Qualquer DBMS tem a capacidade de importar e exportar dados em XML, não necessitando de os guardar nesse formato. Na Figura 2.15, apresenta-se um exemplo da representação de um documento de acordo com o standard da *World Wide Web Consortium* (W3C).

XML é muito popular como standard para transferência de dados entre aplicações através das redes, e possivelmente será o *standard intergalactic* de movimento de dados, mas como DBMS terá algumas dificuldades em impor-se por ser a antítese do KISS (Stonebraker & Hellerstein 2005).

2.3.8 *Deductive*

Uma das propostas de extensão ao modelo relacional, que interliga a lógica de programação e as bases de dados relacionais, é o modelo dedutivo.

O objetivo desta tecnologia é possibilitar extrair, dos dados, informações que não estão explícitas nas relações básicas implementadas através de regras dedutivas derivadas. Um banco de dados é formado pelo conjunto de factos (tuplos) básicos que foram explicitamente definidos e pelas informações extraídas pela aplicação utilizando regras dedutivas. À semelhança da linguagem utilizada nos sistemas relacionais, os sistemas dedutivos usam uma linguagem declarativa, que permite aos utilizadores interagir com os dados indicando o que se pretende fazer, sem ter que se preocupar como se fazem essas operações junto dos dados (Nardon 1996).

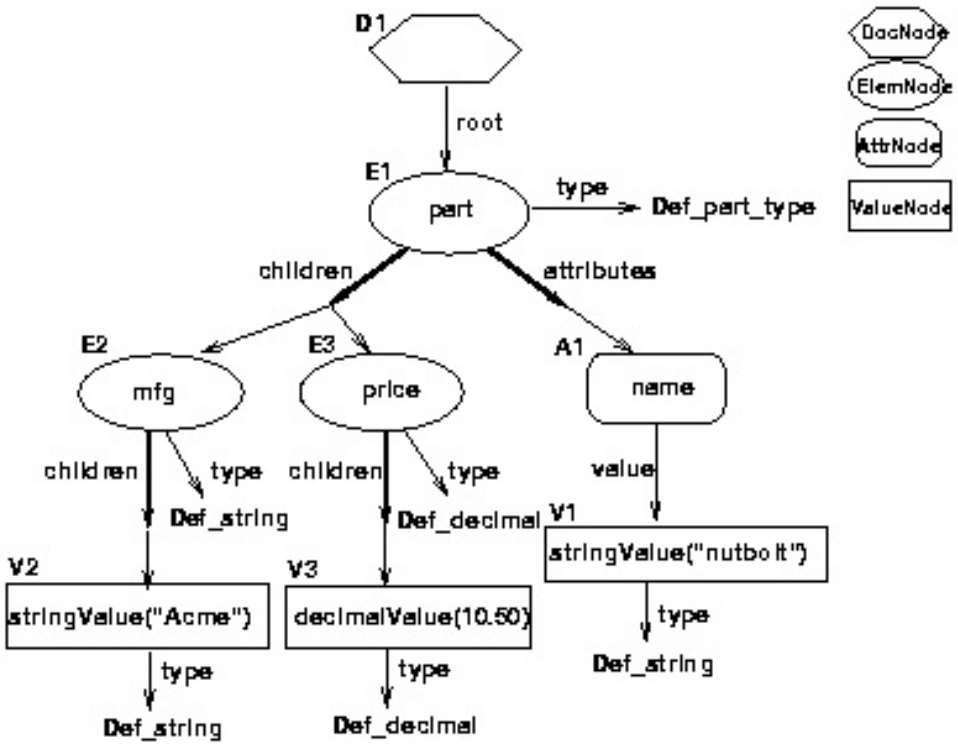


Figura 2.15: Representação de um documento (D1), com os elementos (E1,E2,E3), e os atributos A1 e respetivos valores (V1,V2,V3) de determinado tipo de dados Fonte: http://www.w3.org/TR/2001/WD-query-datamodel-20010215/picture.xfig.gif. Acedido em 20-01-2012

A ideia teve origem numa tentativa de adaptar a linguagem Prolog, que tradicionalmente trabalha com pequenas quantidades de dados em memória, para passar a trabalhar com grandes quantidades de dados normalmente localizados em bases de dados (Ramakrishnan & Ullman 1993).

A dificuldade em conseguir que a linguagem Prolog aceda às bases de dados sem ser numa base de *record-at-time*, e a restrição de só conseguir lidar com regras com prioridades, levou ao desenvolvimento de linguagens alternativas como seja o Datalog (Nardon 1996).

Nos anos 80 e 90, diversos projetos foram iniciados maioritariamente no campo académico. Projetos de investigação nesta área foram desenvolvidos na Universidade de Melborne com o *Aditi*, ou o *Lola* da Universidade de Munique entre muitos (Ramakrishnan & Ullman 1993) e onde a Universidade de Évora também deu um contributo significativo (Abreu 2001). Também do lado comercial, surgiram algumas investigações, nomeadamente pelo projeto *Starburst* da IBM (Ramakrishnan & Ullman 1993).

Apesar das grandes potencialidades desta tecnologia, as referências a este modelo perdem-se algures no final da década de 90. Referindo Stonebraker, talvez o facto da tecnologia

não ser *made in USA* ou ainda o facto de não ir de encontro ao princípio KISS, seja um *handicap* para a sua aceitação global.

2.3.9 MMDB

De acordo com Garcia-Molina & Salem (1992), uma *Main Memory Data Base* (MMDB), utiliza a memória principal como base de armazenamento permanente dos dados, não recorrendo a acessos externos de *storage*, com o intuito de alcançar altos desempenhos. Estes modelos utilizam diferentes estruturas de dados, face aos modelos tradicionais, que recorrem a suportes de informação externos.

A utilização da memória principal tem grandes diferenças, face aos discos magnéticos, com repercussões no desempenho de uma base de dados, nomeadamente:

- O tempo de resposta da memória principal, pertence a outra ordem de magnitude, na ordem dos nano segundos enquanto os suportes magnéticos funcionam na ordem dos milissegundos.
- A memória principal é volátil, enquanto os discos magnéticos não.
- Um disco rígido tem um alto custo fixo por acesso, que não depende da quantidade de informação transmitida, pelo facto do armazenamento ser organizado por blocos. A memória principal não tem esta restrição.
- A organização dos dados num disco rígido é muito importante, com o acesso sequencial mais rápido do que o acesso aleatório. Em memória não se coloca este problema.
- O processador pode normalmente aceder diretamente aos dados em memória, enquanto no caso dos discos não.
- Os dados em memória estão mais propensos a erros de software.

Uma MMDB pressupõe que os dados estão todos em memória, com as suas estruturas de dados a refletirem isso mesmo. Uma base de dados tradicional, com uma grande memória de cache, também vai conseguir colocar toda a base de dados em memória principal, só que toda a sua conceção foi orientada para a utilização de armazenamento externo, recorrendo a técnicas de indexação (por exemplo B-Trees), otimizadas para acessos em suportes externos, que continuarão a consumir recursos de processamento apesar dos dados estarem todos na memória principal.

A confiança numa base de dados em memória, depende da garantia da manutenção dos dados, após um *crash* do sistema. Este problema pode ser minimizado com a utilização de baterias que evitem a falta de energia ao sistema. Como os problemas podem ter origens mais amplas do que uma simples falha de energia, como seja uma falha de hardware, ou

mesmo de software, estes sistemas têm obrigatoriamente que guardar uma imagem dos dados em disco. A forma e a periodicidade da cópia dos dados para disco, determina em parte a confiança num sistema desta natureza.

Aquando da escrita do artigo por Garcia-Molina & Salem (1992), o autor referia exemplos de implementações do modelo como seja o MM-DBMS, HALO, OBE e TBK entre outros. Mais recentemente pode-se referir o projeto VoltDB. O denominador comum de todos os projetos de MMDB assentam no seu alto desempenho, normalmente orientado para sistemas transacionais.

2.3.10 Síntese de outros modelos relacionais

Os sistemas de bases de dados relacionais têm dominado nos últimos 40 anos, pela sua versatilidade, robustez, simplicidade e aplicabilidade numa grande variedade de situações.

De entre os muitos servidores de bases de dados relacionais existentes, nos mais populares encontram-se o Oracle, o Microsoft SQL Server e o IBM DB2 do lado comercial, e o PostgreSQL, o MySQL e SQLite do lado open source.

Muitas propostas têm surgido para novos modelos ao longo de estes anos, e muitas delas repetem os mesmos erros que outras cometeram no passado, ou pura e simplesmente são uma reedição de soluções que não tiveram sucesso. De acordo com Stonebraker & Hellerstein (2005), a história, dos modelos de bases de dados, tem demonstrado que os modelos demasiado complicados de serem entendidos pelo comum dos mortais, não têm vingado, por serem a antítese do princípio KISS.

2.4 Escalabilidade

Cada vez mais, as organizações sentem a necessidade de disponibilizar a sua lógica de negócio junto dos seus clientes (utilizadores), independentemente da sua localização geográfica. Com a utilização da internet, potencialmente, qualquer organização, pode aceder a um mundo inteiro de utilizadores. Para acompanhar o crescimento dos acessos às informações armazenadas pelas suas bases de dados, é necessário fazer crescer os sistemas que as suportam (escalar).

O escalonamento pode ser vertical (*scale-up*) e abrange o desenvolvimento de aplicações em grandes servidores com memória partilhada, ou horizontal (*scale-out*) quando se refere ao desenvolvimento de aplicações com vários computadores pequenos interligados (Michael, Moreira, Shiloach & Wisniewski 2007). Ainda segundo o mesmo autor, a estratégia de *scale-out*, oferece uma melhor relação preço / performance, tendo como revés o aumento da complexidade administrativa do sistema.

Veja-se como é possível escalar um sistema relacional para conseguir responder aos novos desafios que os novos modelos de negócio exigem:

Scale-up A forma mais simples de escalar uma base de dados relacional é através de um *upgrade* do servidor (Figura 2.16). Pode-se instalar mais memória, mais e melhores processadores, mais e melhores discos rígidos. Aumento da capacidade do servidor, não aumentando significativamente a complexidade do sistema, permite portar facilmente as aplicações.

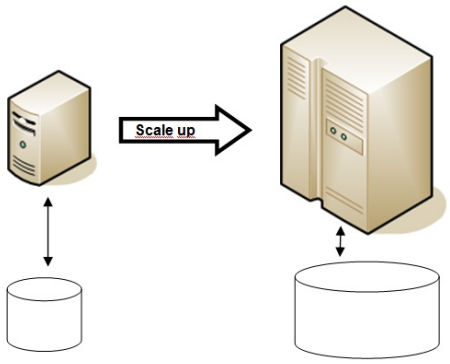


Figura 2.16: Adicionando processadores, discos, memórias, podemos fazer crescer um sistema verticalmente sem aumentar significativamente a complexidade da solução Fonte: Elaboração própria

Scale-out Quando um só servidor, não é suficiente para processar todos os pedidos, uma solução passa pela instalação de um *cluster* de servidores a trabalharem com a mesma base de dados (Figura 2.17). No caso das bases de dados relacionais, isso pode ser atingido adicionando outras máquinas similares, aumentando a capacidade de processamento, suportando mais clientes (Brito 2010).

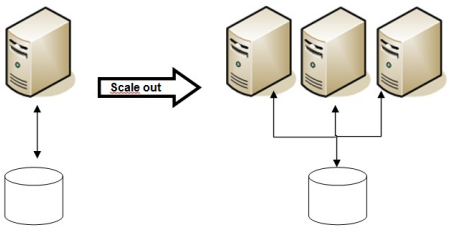


Figura 2.17: Aumentando o número de servidores ligados a uma base de dados, aumenta-se a capacidade do sistema formando um *cluster* Fonte: Elaboração própria

Elasticidade: *Scale-Out/Scale-In* Tem-se assistido a uma grande preocupação com o crescimento dos sistemas de bases de dados, para conseguirem satisfazer mais e mais clientes com mais e mais dados, e com mais e mais rapidez, através da utilização de mais e mais recursos, nomeadamente com a utilização de mais máquinas (*nodes*) contendo partições da base de dados. Minhas, Liu, Aboulnaga, Salem, Ng & Robertson (2012) refere outra preocupação a que os sistemas desta natureza deviam responder: para além de permitirem uma rápida expansão, também devem possibilitar a redução da sua dimensão sem

prejuízo da sua funcionalidade se tal for necessário. A necessidade de reduzir a dimensão do sistema pode estar indexada a picos de acesso a determinados serviços, permitindo aos administradores de sistemas uma melhor gestão dos recursos com a sua redistribuição, de acordo com as necessidades. A elasticidade (*scale-out/in*) é de acordo com Minhas et al. (2012), a capacidade de um sistema de bases de dados crescer, com a adição de mais *nodes* e encolher com a operação inversa, mantendo os serviços que suporta operacionais. Este autor refere a dificuldade dos sistemas atuais em efetuar *scale-in*, pois a sua conceção foi somente orientada para o crescimento horizontal. No seu artigo, Minhas et al. (2012) refere as capacidades de elasticidade do sistema de bases de dados VoltDB.

Replicação de Bases de dados Uma outra abordagem do crescimento numa perspetiva de disponibilidade do sistema, com acessos geograficamente dispersos, é a replicação da base de dados. Nesta abordagem a mesma base de dados é duplicada pelos diversos servidores (*nodes*) intervenientes, garantido assim uma maior disponibilidade do mesmo. A replicação de bases de dados pode-se classificar em *multi-master* e *master-slave* (Filip, Vasar & Robu 2009):

Replicação *Multi-Master* Os clientes podem ler e escrever em qualquer um dos *nodes*, sendo garantido o paradigma ACID, à custa da perda de disponibilidade, pois os *nodes* têm que garantir a atualização síncrona dos dados entre eles, antes de confirmar uma alteração ao cliente, recorrendo às comunicações em rede, que podem conter grandes latências.

Replicação *Master-Slave* Os clientes podem ler a partir de qualquer um dos *nodes*, mas somente um deles, suporta operações de escrita. O *node* que permite a leitura / escrita é o *master* e os restantes são os *slaves*. Sempre que é efetuada uma alteração nos dados no *node master*, essas alterações são replicadas para os *nodes slaves*, de uma forma assíncrona, não garantindo que as leituras efetuadas nos momentos seguintes nos *nodes slaves* reproduzam as alterações efetuadas. Este modelo apresenta uma grande disponibilidade, para situações de leituras, pois os servidores podem estar geograficamente perto dos clientes e potencialmente disponíveis, reduzindo as latências nas comunicações (Tellez, Ortiz & Graff 2007).

Na Figura 2.18, é exemplificado como é possível escalar uma base de dados relacional, utilizando um *middleware* gestor de balanceamento de carga com *failover*.

Organizações que operam à escala global, que na sua maioria oferecem serviços na Internet, viram as suas necessidades de armazenamento / disponibilização de informação crescerem exponencialmente nos últimos anos, onde os tempos de resposta reduzidos são uma exigência do negócio. Empresas como a Google, o Facebook e a Amazon, são exemplos de empresas centradas na web com necessidades em manter e disponibilizar em tempo útil (na ordem dos milissegundos), quantidades "assombrosas" de informação.

Numa perspetiva economicista, relacionando o preço e a performance, entre a opção de

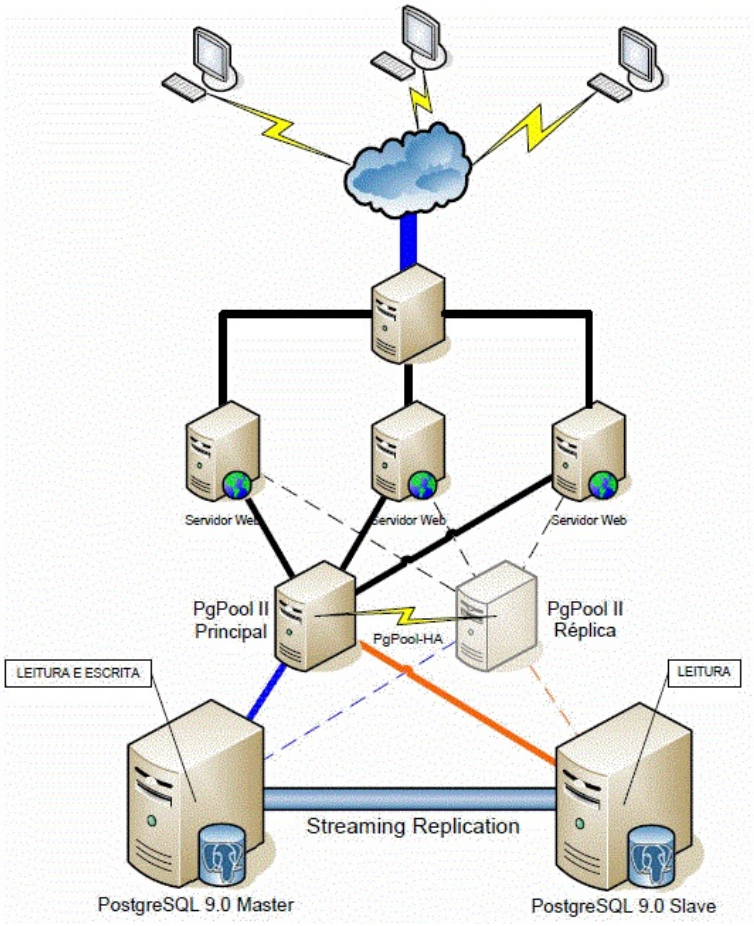


Figura 2.18: Exemplo de escalabilidade horizontal, utilizando o motor PostgreSQL com o middlware PgPool Fonte: http://blogs.dextra.com.br/bdextra/2011/alta-disponibilidade-e-alta-performance-com-postgresql-9-0-pgpool-ii/ acedido em 27/11/2011

scale-up vs scale-out, Michael et al. (2007) conclui que, para dois sistemas com custos de aquisição semelhantes, a abordagem *scale-out* pode ser até 5 vezes mais rápida, o que só por si justifica a sua escolha, em detrimento do crescimento *scale-up*.

O modelo relacional, também tem limitações em escalar a partir de determinado nível, quer seja *scale-up* ou *scale-out*. Quando uma aplicação necessita de responder a mais utilizadores, o tempo de resposta aumenta. Para recuperar o tempo de resposta exigido, podem-se efetuar sucessivos upgrades no servidor (*scale-up*), até ao ponto em que os tempos de resposta ganhos não justificam o investimento realizado.

Instalar mais servidores (*scale-out*), partilhando a mesma base de dados (*cluster*), resolvem parte dos problemas do acesso por parte dos clientes ao sistema. A partir de determinado ponto de crescimento, a própria base de dados já não consegue responder em tempo útil à procura de informação, por parte do elevado número de servidores do *cluster*. É aqui que se torna necessário escalar a própria base de dados.

Uma solução para escalar a base de dados, é partir a base de dados em pequenas frações (particionar a base de dados), distribuindo-a por vários servidores (*Sharding*). Cada servidor contém uma pequena "fatia" da informação do "bolo" total, conseguindo tempos de resposta muito rápidos, resolvendo o problema de performance em sistemas de grande dimensão.

Particionar uma base de dados relacional não é uma tarefa trivial, quando se trata de gerir o acesso de milhões de utilizadores, a petabytes de informação (Brito 2010).

De acordo com o teorema *Consistency, Availability, Partition tolerance* (CAP) de Brewer, que aborda a problemática da consistência, da disponibilidade, e da tolerância ao particionamento dos dados através da rede, em sistema distribuídos, não é possível garantir em simultâneo mais do que duas dessas três características (Brewer 2000).

Podemos ter uma boa consistência e disponibilidade, com uma fraca tolerância ao particionamento dos dados ou uma boa consistência com tolerância ao particionamento, mas com uma fraca disponibilidade ou ainda uma boa disponibilidade com tolerância ao particionamento, apresentando uma fraca consistência (Figura 2.19).

A título de exemplo, encontramos as bases de dados relacionais e a gestão de ficheiros centralizada, com prioridade à consistência e disponibilidade, a replicação de bases de dados *multi-master* onde a prioridade vai para a consistência e tolerância ao particionamento e, por último, os servidores *Domain Name System* (DNS) com disponibilidade e tolerância ao particionamento.

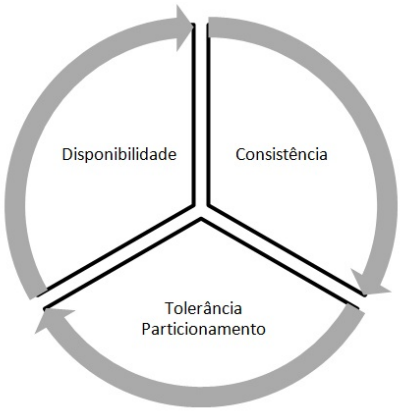


Figura 2.19: De acordo com o teorema de Brewer, das três características desejáveis, um sistema distribuído só consegue garantir duas Fonte: Elaboração própria

Os sistemas de informação baseados no modelo relacional têm vindo a ser apetrechados de novas extensões para tentar responder a grande parte das problemáticas, nomeadamente através do modelo *Object-Relational Database Management System* (ORDBMS), que permite a criação de novos tipos de dados não estruturados associados aos dados estruturados, bem como a criação de funções específicas para problemas específicos.

Contudo, o modelo relacional desenvolvido numa base de *one size fits* all (uma solução para todos os problemas), que abarca a maioria das necessidades, deixa de lado uma franja dos problemas, em parte pela incapacidade de conseguir responder a sistemas de hiper-dimensão com alta-performance e alta-disponibilidade.

A dificuldade em escalar para níveis muito grandes, demonstrado pela incapacidade que tem para responder eficazmente à problemática que envolve o teorema de Brewer, leva as entidades que necessitam de sistemas de hiper-dimensão a procurar soluções à margem do modelo relacional.

2.5 Novos modelos de bases de dados

2.5.1 Panorama geral

Com a web 2.0 surgiu a possibilidade dos utilizadores para além de acederem à informação, serem eles mesmo criadores de conteúdos. Num piscar de olhos, em vez de milhares de produtores de conteúdos, passou-se a ter muitos milhões de produtores de conteúdos, o que implica o tratamento e armazenamento de informação em quantidades colossais. Por outro lado, os dados deixaram de estar isolados, mas sim interligados, residindo em documentos não estruturados.

Empresas, cujo principal negócio assenta na internet, e que se tornaram *players* globais devido à inovação dos seus produtos ou serviços, rentabilizando o facto de haver um canal direto com muitos milhões de pessoas em todo o mundo, sentiram a necessidade de criar elas próprias as soluções informáticas de suporte ao seu negócio, entre as quais, soluções para a problemática do *storage*.

Empresas como a Google, Amazon e Facebook, necessitaram criar formas de ultrapassar as limitações do teorema de *Brewer*, por vezes, elaborando abordagens completamente novas, ou reciclando tecnologias já esquecidas nos baús poeirentos, visto que as empresas especializadas em *storage* não possuíam as soluções adequadas.

Este tipo de sistemas, que não obedece ao modelo relacional, designa-se usualmente por NoSQL. Longe de ser um novo standard, o NoSQL representa todos os sistemas que não estão implementados segundo o modelo relacional. O termo surgiu em 1998, como abreviatura de *Not Only SQL*. O seu objetivo não é substituir o modelo relacional, largamente implementado com provas dadas, mas sim promover soluções onde tal modelo não tem resposta (Brito 2010).

Uma das primeiras implementações do NoSQL, surgiu em 2004 com o Bigtable da Google, seguido por outros casos de sucesso, como o Dynamo da Amazon em 2007 ou Cassandra desenvolvido no Facebook em 2008 e, mais tarde, adotado pelo Twiter em 2010 (Brito 2010).

Para além da problemática do armazenamento de dados de dimensões massivas, outras abordagens têm surgido ao longo dos tempos, rompendo com o modelo relacional, que segundo Stonebraker & Cattell (2011) deve-se à insatisfação com o modelo relacional.

2.5.2 Paradigma BASE

O paradigma dos sistemas NoSQL é o BASE, em oposição ao ACID que, em traços gerais, indica que o sistema tem que estar altamente disponível, e eventualmente consistente, ao passo que o ACID privilegia a consistência dos dados em detrimento da disponibilidade. O sistema é:

Basically avaiable Privilegia a disponibilidade acima de tudo.

Soft state As alterações aos dados, não levam de um estado consistente a outro igualmente consistente, podendo haver dados ligeiramente desatualizados.

Eventual consistency Uma alteração aos dados não será repercutida de imediato para todo o sistema, mas essa alteração será propagada e visível em todo ele (Vogels 2008)

2.5.3 Sistemas NoSQL

Existem muitas abordagens a sistemas NoSQL, mas de acordo com von der Weth & Datta (2011), de entre os principais sistemas NoSQL encontramos os *Pure Key-Value Stores*, os *Column Stores*, os *Document Stores* e os *Graph databases*.

2.5.3.1 *Pure key-value stores*

São sistemas de armazenamento, que disponibilizam três funções básicas na sua API, *put, get e delete*, sendo o acesso aos dados efetuado através da sua chave primária. Nesta categoria, podem-se referir as implementações do Dynamo e Voldemort, entre as mais utilizadas. Os dados não estão relacionados uns com os outros, somente existem dados que têm uma chave de acesso, e qualquer ligação entre eles é efetuada pela aplicação, o que o torna num modelo muito simples, que permite escalar facilmente.

2.5.3.2 *Column stores*

Sistemas de armazenamento que utilizam tabelas organizadas por colunas, com uma grande otimização na utilização de tabelas esparsas. Neste modelo, os campos que não são preenchidos para uma determinada linha da tabela, não são criados, resultando uma

grande racionalização do espaço ocupado, permitindo um acesso aos dados muito rápido. Outra potencialidade deste modelo prende-se com a facilidade na criação de novos campos, sem alteração da estrutura da tabela. Na prática, e fazendo analogia aos registos do modelo relacional, neste modelo cada registo pode ter o seu próprio esquema. Os sistemas column stores oferecem na sua API, a possibilidade de pesquisas por atributos chave, para além do *put, get e delete*. O Bigtable, o PNUTS (Yahoo), o Hadoop Hbase e o Hypertable, encontram-se entre as mais populares implementações deste modelo.

2.5.3.3 *Document stores*

Muito semelhante ao modelo anterior, mas direcionado para o armazenamento de dados semiestruturados, otimizado para acesso a documentos. Os documentos são normalmente guardados recorrendo ao *JavaScript Object Notation* (JSON), permitindo o armazenamento de dados complexos. Sistemas como o CouchDB, o MongoDB e o Terrastore, são exemplos de implementações deste modelo.

2.5.3.4 *Graph stores*

Baseado na teoria dos grafos, onde não há linhas nem colunas, nem tabelas, mas sim grafos, com nós, arestas e propriedades. Pretende dar resposta a problemáticas que envolvam algoritmos de grafos, permitindo representar dados de grande complexidade. Neo4j e HypergraphDB, são exemplos de implementações deste tipo de bases de dados.

2.5.4 Escalabilidade em NoSQL

No caso da Amazon, que tem no seu centro de negócio a venda de produtos *on-line*, é vital, que os utilizadores possam aceder aos dados dos produtos e efetuem as suas compras, sem falhas e em tempos considerados aceitáveis, independentemente da sua localização, à escala global (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall & Vogels 2007).

Sabendo que problemas constantes de acesso a um serviço deste tipo levam os clientes a procurarem alternativas, que podem destruir o negócio em si, a Amazon definiu como *Service Level Agreements* (SLA), que 99,9% dos pedidos sejam tratados pelo sistema até 300ms, para picos de 500 pedidos por segundo. Para realizar esta qualidade de serviço, e fazer face aos crescimentos de negócio, foi desenvolvido internamente um modelo de armazenamento de informação que responde às exigências do SLA: o Dynamo.

A Figura 2.20 representa a arquitetura implementada pela Amazon. Os pedidos de clientes chegam a um dos muitos *datacenters*, oriundos da nuvem. Aí, é atendido pelo web server que estiver mais disponível que, de seguida, encaminha as solicitações de informação de acordo com o serviço, através dos agregadores de serviços. Os agregadores de serviços

direcionam o pedido para a *farm* do serviço requerido. Os serviços têm os dados particionados em *shards*, distribuídos por diversos *datacenters*, de forma a garantirem uma boa tolerância a falhas.

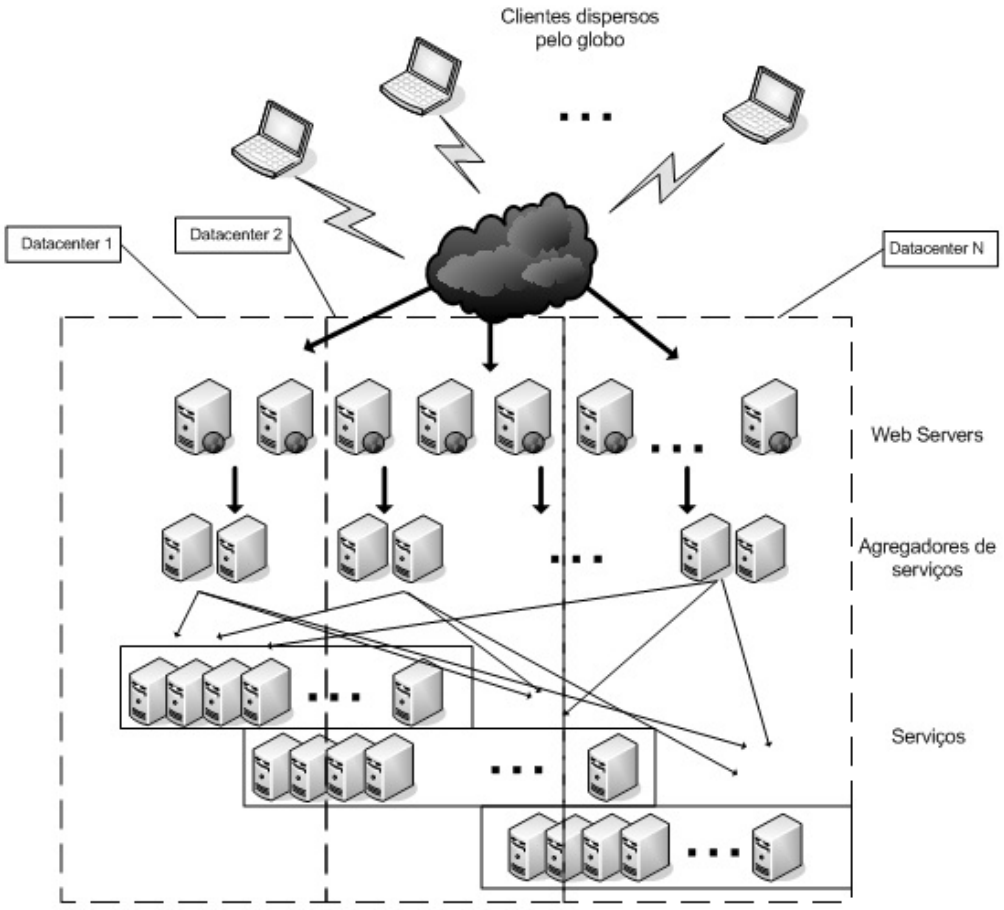


Figura 2.20: Arquitetura do Dynamo da Amazon. Fonte Elaboração própria, baseada em DeCandia et al. (2007)

Para conseguir atingir os objetivos definidos internamente pela Amazon, e que serve de referência para outros sistemas NoSQL, foi desenvolvido um sistema com as seguintes características:

2.5.4.1 *Interface*

Interface muito simples, com três funções básicas disponibilizadas aos programadores, *put, get e delete*, que permitem aceder aos dados através das chaves primárias, visto que a grande maioria da informação necessária ao negócio, responde satisfatoriamente a esta forma de pesquisa.

2.5.4.2 *Eventually Consistent*

Num sistema ACID, as transações são garantidas pelo gestor de armazenamento, em que ou todas as operações que a compõem têm sucesso, ou nenhuma o tem (Figura 2.6). Por exemplo, quando se faz uma transferência de dinheiro de uma conta para outra, é necessário atualizar as duas contas, sob pena de perda de consistência.

Em sistemas altamente distribuídos, não é possível efetuar um *lock* aos dados enquanto dura a transação, pois é suposto o sistema estar a ser acedido por milhões de utilizadores, e sabendo que os dados podem estar em muitos *nodes*, espalhados por vários *datacenters* em pontos diferentes do globo.

Não conseguindo ter alta disponibilidade, alta consistência e tolerância ao particionamento (teorema de *Brewer*), estes sistemas optam claramente pela alta disponibilidade e tolerância ao particionamento, em detrimento da consistência, sendo esta última, eventual. Esta abordagem enquadra-se nas consistências fracas (*weak consistency*) em que o sistema de armazenamento, garante que se os dados forem atualizados, eventualmente, todos os acessos posteriores devolverão a versão correta.

Uma das técnicas para garantir que os dados são os mais recentes, é a implementação de versões dos dados utilizando um *timestamp* para as distinguir. Sabendo que uma leitura envolve um pedido a todos os *nodes* que contêm aquela chave primária, e que devido a problemas de sincronização, podem conter versões diferentes dos mesmos dados, e que a leitura é aceite quando a aplicação receber um pré-determinado número de respostas, há a necessidade por parte da aplicação de saber qual das versões a utilizar.

A primeira abordagem é efetuada pelo próprio sistema de armazenamento, em que um dos servidores que atua como coordenador para uma determinada partição de dados, analisa as diferentes versões existentes nos diferentes *nodes*, e caso detete inconsistências, tenta efetuar uma reconciliação, (reconciliação sintática). Caso a inconsistência não seja possível de realizar com a reconciliação sintática a nível do servidor aquando da leitura, a lógica aplicacional, vai ter que lidar com o problema e realizar a consistência dos dados (reconciliação semântica) (Vogels 2008).

2.5.4.3 Thrust environment

Eliminação dos processos de controlo de acessos às aplicações internas, partindo do princípio que estão dentro de um ambiente não hostil em vez da tradicional verificação utilizador + password para acesso aos servidores / serviços / dados, poupando valiosos milissegundos.

2.5.4.4 *Sharing nothing*

Os computadores utilizados são simples pc's, que não utilizam técnicas de partilha de disco, nem memória com os outros computadores, de forma a eliminar os tempos de sincronização a esse nível, conseguindo-se aqui, mais um ganho substancial de tempo (milissegundos).

Referindo Stonebraker & Cattell (2011), um motor de bases de dados pode ser suportado por três tipos de arquitetura de hardware:

SMP (*Shared Memory Multiprocessing*) Significa que o motor corre numa única máquina que poderá ter vários processadores que partilham uma única memória principal, assim como o espaço em disco. As limitações começam a surgir logo na partilha da largura de banda à memória principal por parte dos diversos processadores.

Clusters Vários *nodes* com memória principal privada partilham o mesmo espaço em disco. Devido a cada *node* deter um *buffer* privado em memória, este necessita de ser sincronizado com os *buffers* dos restantes intervenientes do *cluster*, o mesmo sucedendo com a tabela de *locks*, que limitam a escalabilidade de um sistema deste tipo. Não é vulgar encontrar um motor de base de dados a correr num *cluster* com mais do que 10 *nodes*.

Sharing nothing Cada *node* não partilha nem a memória principal, nem o disco com os outros *nodes*, formando uma coleção de *nodes* ligados através da rede de dados. Um sistema bem construído baseado em *sharing nothing*, pode escalar até esgotar a largura de banda da rede de dados que o suporta. É frequente, sistemas NoSQL conterem mais de 100 *nodes*, com o Bigtable a conter milhares de *nodes*.

2.5.4.5 *Sharding*

A base de dados global é particionada em pequenas frações, distribuídas por cada *node*, num processo designado por *sharding*. Esta técnica permite que o acesso a essa parte reduzida dos dados seja extremamente rápido. Na Figura 2.21, está esquematicamente representada a forma como um pedido obtém os dados com a técnica *sharding*.

2.5.4.6 *Fault tolerance*

Para evitar que falhas de servidores ou falhas de rede ou mesmo falhas de *datacenters* inteiros, possam afetar o normal processo de uma compra na Amazon, por parte de um cliente, cada *shard* (unidade de particionamento existente num *node*), está replicado por diversos *nodes*, que se encontram em mais do que um *datacenter*, organizados em *ring*. Portanto, cada *shard* tem algures uma réplica que pode assegurar a continuação do processo sem que o cliente chegue a perceber caso haja algum problema.

As operações de leitura e escrita são efetuadas por quórum, em que o seu sucesso é aceite se pelo menos um determinado número de *nodes* responder positivamente. Para evitar a perda de tempo a efetuar um pedido de operação a um *node* que está demasiado ocupado, ou que falhou, e que depois seria necessário reenviar o pedido para outro *node* que também contém a réplica dos dados, as operações são lançadas para todos os *nodes* que contêm o *shard* alvo, e logo que exista um nº de respostas que façam quórum, é determinado que a operação teve sucesso.

O pedido do cliente é direcionado para um *node* do serviço acedido [1] que, através de *hash code*, determina em que *nodes* estão as réplicas dos *shard's* que contêm o documento (*shard* E). A configuração indicada de N=3, W=2 e R=2, significa que existem 3 *nodes* com réplicas (N=3) do mesmo *shard*, e que para haver quórum de escrita têm que responder 2 *nodes* (W=2) e para quórum de leitura, têm que responder também 2 *nodes* (R=2). Alterar os valores de N,W,R, altera os níveis de disponibilidade e consistência.

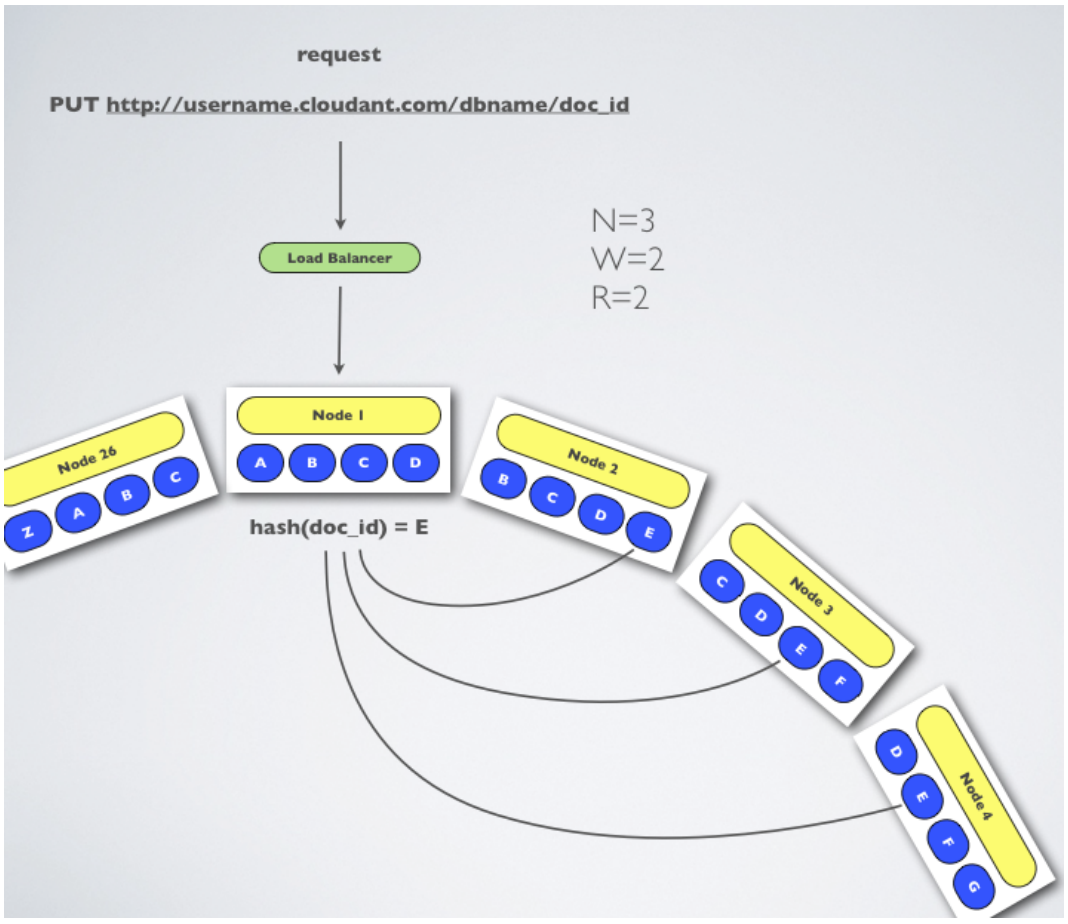


Figura 2.21: O pedido recebido pelo *node* 1, oriundo dos sistemas de *load balancing*, é direcionado para todos os *nodes* que contêm o *shard* onde está o documento solicitado Fonte: `http://blog.cloudant.com/dynamo-and-couchdb-clusters/` acedido em 27/11/2011

[1] Por exemplo no caso do Google o mail, é um serviço especifico, que contém uma infraestrutura dedicada

Esta aceção de que uma operação de leitura ou escrita está correta, quando há um quórum aceitável, distancia-se de outros modelos onde é exigida uma tolerância às falhas bizantina (*Byzantine Fault Tolerance*) onde todos os *nodes* têm que responder afirmativamente para considerar a operação concluída com sucesso (Pedone, Schiper & Armendariz-Inigo 2011).

2.5.4.7 Modelo de dados

No caso do Google Bigtable, que é um dos sistemas orientado a colunas (*column storage*), com grande eficiência em armazenamento de tabelas esparsas, as células vazias não são materializadas, poupando valiosos recursos em oposição ao modelo relacional, que reserva o espaço para todos os atributos em todas as linhas de uma tabela, mesmo que não sejam preenchidos.

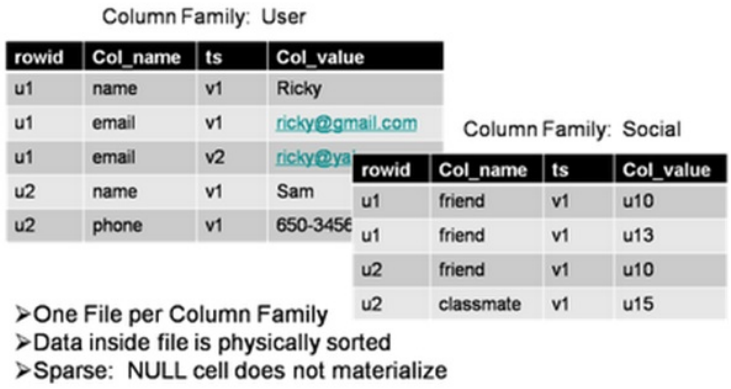


Figura 2.22: O modelo de armazenamento orientado a colunas, do Google Bigtable Fonte: http://horicky.blogspot.com/2010/10/bigtable-model-with-cassandra-and-hbase.html acedido em 19/11/2011

O Bigtable contém uma única chave de acesso, a chave primária constituída pelas colunas *rowid* + *Col_name* + *ts* Figura 2.22. A coluna *Col_name* contém o nome dos campos, a coluna ts (*timestamp*) regista o momento da inserção da linha, permitindo a existência de várias versões dos mesmos dados, a coluna *Col_value* suporta os dados em si. Em essência a estrutura é muito simples, só tem a chave primária, o nome da coluna o *timestamp* e valor da coluna. As várias famílias de dados têm objetivos similares às tabelas do sistema relacional, agregando dados com afinidades entre si.

As atualizações aos dados neste modelo de dados, não passam de uma nova inserção de dados com um novo *timestamp*. Este processo gera diversas versões dos mesmos dados, que depois são eliminadas por um processo próprio sempre que já não sejam necessárias.

2.5.4.8 Acesso aos dados (modelos de)

Os sistemas NoSQL oferecem alta performance e alta disponibilidade, sacrificando toda uma série de facilidades oferecidas por outros sistemas, que poderiam consumir precioso tempo de computação com a sua disponibilização.

Se por um lado se obtém maiores performances, por outro, com a perda das funções de acesso aos dados avançadas, tornam o desenvolvimento das aplicações, mais difíceis para o lado dos programadores das aplicações, com as suas responsabilidades a terem início na gestão do acesso aos dados (lógica de armazenamento) até à lógica aplicacional, porque nestes modelos não existe uma separação entre estas duas perspetivas, em grande contraste com o modelo relacional (von der Weth & Datta 2011).

Com esta organização dos dados, é extremamente fácil a uma aplicação, criar novos campos, através da inserção de uma nova linha com um novo valor em nome de coluna e como as chaves primárias estão ordenadas, é muito rápido o acesso aos dados (Ho 2010). O acesso aos dados é efetuado normalmente pela chave primária, mas se é necessário efetuar consultas que não utilizem a chave primária, podem-se utilizar as seguintes abordagens:

Divide & Conquer Como não se está a aceder por chave primária, a aplicação não sabe em que *shard* se encontra a informação pretendida (no limite, poderá estar em todas as shards), e por isso efetua o pedido a um determinado *node*. O *node* contactado replica o pedido a todos os *nodes* por onde a tabela está particionada. Cada *node* processa localmente o pedido com os dados constantes na sua *shard*, e devolve os resultados ao *node* inicial que, por sua vez, os disponibiliza à aplicação. Cada *node* efetua uma pequena operação, porque o *shard* da sua responsabilidade não é de grande dimensão, e pode inclusivamente estar todo em memória, produzindo um resultado muito rápido. Uma das mais conhecidas aplicações que utiliza esta técnica, é o MapReduce utilizado pelo BigTable, HBase, CouchDB e MongoDB. (von der Weth & Datta 2011)

Índices Secundários Outra forma de aceder aos dados, é a utilização de chaves secundárias, não disponibilizadas nativamente pelo sistema de armazenamento, mas possíveis de utilizar se forem criadas e geridas pela própria aplicação. Essas chaves utilizam a estrutura básica de armazenamento (chave primária e dados) para construírem os índices secundários, como se de dados vulgares se tratassem. (von der Weth & Datta 2011)

2.5.5 *NewSQL*

Tradicionalmente as grandes empresas têm diversos sistemas OLTP normalmente baseadas em sistemas relacionais. Para se proceder a uma análise do negócio, invariavelmente é necessário carregar esses dados para uma *datawarehouse*, através de um processo de ETL (*Extract Transform and Load*). Raramente os recursos destinados ao OLTP são

partilhados pela *datawarehouse*, devido ao peso das *querys* exigidas a estes sistemas de armazenamento de dados por parte dos gestores de negócio, que não raramente podem demorar de dezenas de minutos a horas a serem processadas. Stonebraker (2011) designa estes sistemas por **OldOLTP**, assentes em **OldSQL**. O surgimento dos negócios centrados na web vem mudar os requisitos dos sistemas, para dar resposta ao crescimento, rapidez e disponibilidade exigidas. Stonebraker (2011) designa estas novas soluções como **NewOLTP**.

De acordo com Stonebraker (2011), os sistemas NewOLTP têm as seguintes características:

2.5.5.1 Alto desempenho (*Throughput*)

Com uma grande variedade de serviços oferecidos pela internet, acedidos por uma grande quantidade de dispositivos, desde os tradicionais computadores (PC's) até aos *smartphones*, que permitem que quase todas as pessoas (pelo menos no mundo ocidental) estejam constantemente a interagir de alguma forma com serviços na internet, releva para planos secundários os tradicionais sistemas relacionais. O NewOLTP necessita de DBMS com melhores capacidades de crescimento e desempenho.

2.5.5.2 Análise em tempo real

Os serviços baseados na web, não se compadecem com *querys* que demoram dezenas de minutos a serem processadas. NewOLTP exigem respostas em tempo real.

Para construir um sistema NewOLTP, de acordo com Stonebraker (2011) existem as seguintes possibilidades:

2.5.5.3 Sistema tradicional OLTP

O OldSql, pedra basilar do OldOLPT, não consegue responder aos novos requerimentos, com as *datawarehouses* incapazes de responder em tempo real.

2.5.5.4 NoSql

Surgiram nos últimos tempos, muitas propostas de sistemas NoSql, que reclamam uma grande escalabilidade e grande performance, alcançada através da redução ou mesmo a eliminação da permissão à realização de transações, disponibilizando interfaces de baixo nível no acesso aos dados. Segundo o autor citado, uma das exigências do NewOLTP é a manutenção das transações ACID, descartado ou menosprezado pelos sistemas NoSql, migrando a problemática da consistência para dentro das aplicações. A construção da consistência dentro das aplicações torna-as muito mais difíceis de desenvolver.

2.5.5.5 NewSql

Sistemas que mantêm uma linguagem de alto nível (o SQL) de acesso aos dados, que preservam as transações ACID e apresentam uma grande capacidade de escalabilidade (*scale-out*) e performance, capazes de responder aos desafios dos serviços centrados na web. Nesta inovadora abordagem, podem-se referir sistemas de gestão de bases de dados como seja o Clustrix, o NimbusDB e o VoltDB.

Capítulo 3

Metodologia

Nos anos mais recentes, surgiu a necessidade de sistemas de armazenamento de dados com desempenhos muito para além de todas as previsões anteriores. Os sistemas de hiper-dimensão com alto desempenho e alta disponibilidade não surgiram das fontes mais espectáveis, como as universidades e as grandes empresas tecnológicas, mas sim, de simples *startup's*.

Com a disponibilização de um canal de informação de reduzido custo e acessível mundialmente, surgiram os modelos de negócio que operam à escala global. Se uma mercearia de bairro pode ter umas quantas centenas de clientes limitados à sua localização geográfica, um negócio centrado na web pode, virtualmente, abarcar todos os habitantes do planeta, bem como todos os produtos / serviços possíveis.

Algumas organizações têm conseguido crescer ao ponto de angariarem milhões de clientes. Veja-se o caso do Facebook que conta em fevereiro de 2012 com 845 milhões de utilizadores em 70 línguas (*Suposicao de crescimento facil do Facebook e questionada, diz analista* 2012), no que configura um sistema de hiper-dimensão.

Estas empresas necessitam de disponibilizar quantidades colossais de informação em qualquer parte do planeta, para milhões de utilizadores em simultâneo numa base temporal muito curta, na ordem dos milissegundos necessitando para isso de alta performance.

Sistemas deste nível, que disponibilizam serviços à escala planetária, não podem ser reféns de possíveis falhas nos seus sistemas, pois se, por exemplo, um utilizador localizado num determinado ponto do planeta está a realizar uma compra, essa transação tem que ser realizada, mesmo que nesse momento um furação esteja a destruir um dos *datacenters* do fornecedor. Resulta daqui uma necessidade de alta disponibilidade que implica uma grande tolerância a falhas.

Paralelamente, cada vez mais, têm surgido necessidades de tratamento de dados não estruturados, onde o *schema first* não se aplica, nomeadamente dados na forma de documentos, imagens, vídeo e outros formatos multimédia que, apesar de poderem ser processados com o ORDB, a sua performance não satisfaz todos.

Se a utilização de dados não estruturados pode ser alvo de discussão sobre se o modelo relacional é o mais correto ou não, já a utilização de bases de dados de hiper-dimensão levanta um problema bem claro de escalabilidade que dificulta a utilização do RDBMS.

Neste trabalho, procura-se aferir se já existem soluções disponíveis de armazenamento de dados, que a custos acessíveis, qualquer entidade possa criar o seu sistema de *storage*, com escalabilidade suficiente e não perdendo as funcionalidades a que os programadores se habituaram ao longo de quatro décadas com o modelo relacional e a sua linguagem de alto nível Sql.

3.1 Seleção da base de dados emergente para testes

Como ficou demonstrado na revisão da literatura, existem muitos modelos de bases de dados emergentes que pretendem responder aos requisitos dos negócios baseados na web: alto desempenho, alta disponibilidade, com alta escalabilidade. Se analisarmos a forma como no passado recente algumas instituições resolveram o problema, desde o desenvolvimento ou melhoria dos sistemas operativos existentes, passando pela criação de sistemas de armazenamento de dados de raiz, que implicou a contratação de verdadeiros exércitos de programadores, só possível pela grande disponibilidade de recursos financeiros. Esta abordagem está fora do alcance da esmagadora maioria das entidades, e o caminho tem que obrigatoriamente ser outro. Uma potencial solução à problemática em estudo, com custos muito aceitáveis, é oferecida pelo VoltDB. Apesar de ser muito recente, esta solução vem sendo referida em diversos estudos, como muito promissora. Por ser Open Source, pode ser utilizada por qualquer entidade, e pelo facto de poder utilizar hardware comum, resulta numa solução económica. O VoltDB promete revolucionar as soluções de *storage*, oferecendo o conforto dos sistemas SQL, mas com as potencialidade do NoSql. Pelos fatores já apresentados foi selecionado o motor de bases de dados VoltDB para um estudo mais aprofundado neste trabalho, representando uma nova geração de soluções para sistemas de hiper-dimensão com alta-performance.

3.2 O VoltDB

Segundo (Voltdb 2012), este motor de bases de dados apresenta-se como um sistema de alto desempenho, relacional, escalável para grandes volumes de dados com respostas em tempo real. Criado pelo professor *Michael Stonebraker* e outros cientistas do *Massachusetts Institute of Technology* (MIT), com o intuito de correr RDBMS em infraestruturas escaláveis horizontalmente. O VoltDB utiliza o interface de alto nível SQL, permitindo transações

ACID, representando o que (Stonebraker 2011) designa por NewSql para responder ao NewOLTP. Os responsáveis desta nova arquitetura afirmam que ela é muito mais rápida do que os modelos tradicionais, por eliminar grande parte do *overhead*. Na figura 3.1, é apresentada a distribuição do tempo de processamento, pelas tarefas necessárias ao funcionamento de um DBMS tradicional.

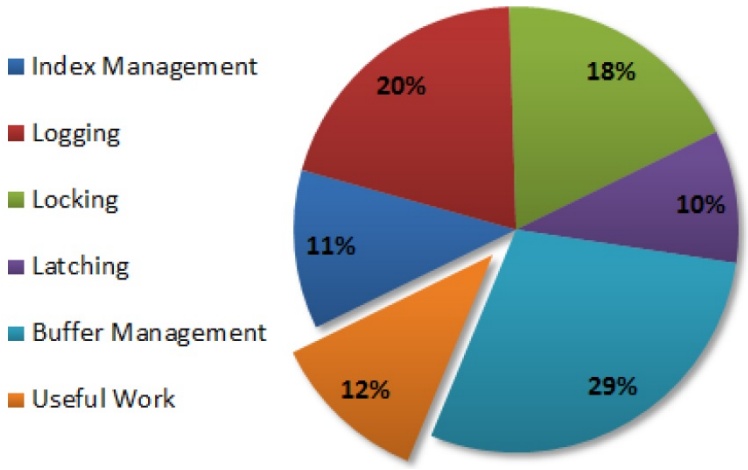


Figura 3.1: Distribuição dos recursos de processamento pelas diferentes tarefas Fonte: (Voltdb 2012)

O *overhead* numa base de dados tradicional é causado por (Voltdb 2012):

Index Management A gestão dos indexes que são atualizados sempre que é efetuada uma alteração à base de dados.

Logging Sempre que é efetuada uma alteração à base de dados, esta é escrita duas vezes. Uma nos dados em si e outra no log da base de dados, que é de imediato escrita em disco, para garantir a durabilidade perante potenciais falhas do sistema.

Locking Para efetuar uma transação, é necessário efetuar um *lock* aos dados para eliminar o potencial conflito de concorrência, implicando filas de espera por parte de outros processos.

Latching Num ambiente *multi-threading*, é necessário efetuar *locks* a todas as estruturas partilhadas, como sejam os indexes, tabelas dos dados, tabelas de sistema, entre outros, implicando filas de espera.

Buffer Management Os dados estão normalmente sediados em páginas de tamanho fixo no disco rígido. É necessário um processo que faça a gestão das páginas em *buffer*

num determinado momento.

Useful Work O trabalho que produz um resultado realmente útil, que dado o pedido do utilizador, seleciona a informação requerida no universo disponível devolvendo-a. Num sistema tradicional RDBMS, só uma pequena fração do processamento resulta em algo útil, com o processador a dispender a maior parte do seu tempo a gerir a concorrência entre os diferentes processos aos mesmos recursos e a esperar pelas respostas dos acessos a disco.

O VoltDB afirma (Voltdb 2012) eliminar o *overhead* do sistema, proporcionando-lhe uma grande performance através de:

Arquitetura *shared-nothing* Dados e processos associados são distribuídos pelos diversos CPU's em partições num *cluster shared-nothing*.

Main memory database Os dados residem na memória principal eliminando a necessidade de gestão de *buffers*.

Single Threading As transações são executadas sequencialmente em memória, eliminando a concorrência e a necessidade de *locking* e *latching*.

Replicação *multi-master* Alta disponibilidade garantida através de replicação *multi-master* síncrona, através de múltiplos *nodes*.

3.2.1 A arquitetura VoltDB

Este motor de bases de dados foi concebido para tirar partido da memória RAM, e dos múltiplos *cores* disponíveis em máquinas que se encontram no mercado a preços acessíveis. A base de dados VoltDB tem as seguintes características:

Alta performance e baixa latência em operações SQL

Para conseguir uma alta performance com uma baixa latência, o VoltDB utiliza as seguintes técnicas (Voltdb 2012):

Operações realizadas em memória Com a redução dos preços da memória RAM, é possível aos computadores de hoje, disponibilizarem centenas de Gigabytes de RAM. Este grande volume de RAM é suficiente para conter os dados na íntegra, eliminando a necessidade de acesso a discos externos. As operações realizadas em memória não têm

que esperar pela resposta dos discos externos aquando da realização de uma transação, eliminando o *latching* e *buffer management*.

Utilização de stored procedures Ao viajar pela rede entre o cliente e o servidor, os dados reduzem significativamente o tempo de resposta de um sistema. O acesso aos dados é efetuado recorrendo à utilização de stored procedures, como a forma mais eficaz de tirar partido das potencialidades do sistema, reduzindo o tráfego na rede. Com esta técnica, existe somente um pedido de informação e uma resposta a esse pedido. A execução de uma *stored procedure* resulta numa transação, que termina com um *commit* ou *rollback*, garantindo a consistência da base de dados.

Execução de stored procedures em single-threading A execução de diversas transações em simultâneo por parte dos sistemas tradicionais implica uma onerosa gestão da concorrência a todos os recursos acedidos (latching). O VoltDB elimina o *overhead* associado ao m*ulti-threading*, permitindo que só uma transação seja executada de cada vez (*single-threading*) em cada partição do *cluster*.

Single Partition Transaction Cada partição do *cluster* executa autonomamente as transações, em relação às outras partições (Figura 3.2). Desta forma, se as *querys* e os dados forem cuidadosamente planeados, é possível obter grandes desempenhos.

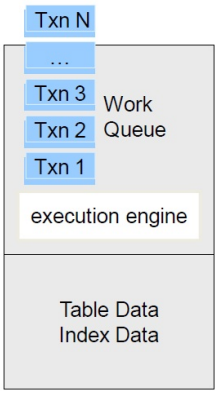


Figura 3.2: VoltDB *single partition transaction* Fonte: (Voltdb 2012)

Multiple Partition Transaction Se os dados necessários à transação não se encontram numa única partição, é executada uma transação *multi-partition*, em que um *node* atua como coordenador da transação. O coordenador pede os dados aos outros *nodes* envolvidos e disponibiliza-os finalmente ao cliente (Figura 3.3).

Ambiente seguro de comunicação (*trust environment*) Partindo do princípio que o cluster estará num ambiente seguro e que os processos que acedem aos dados são de

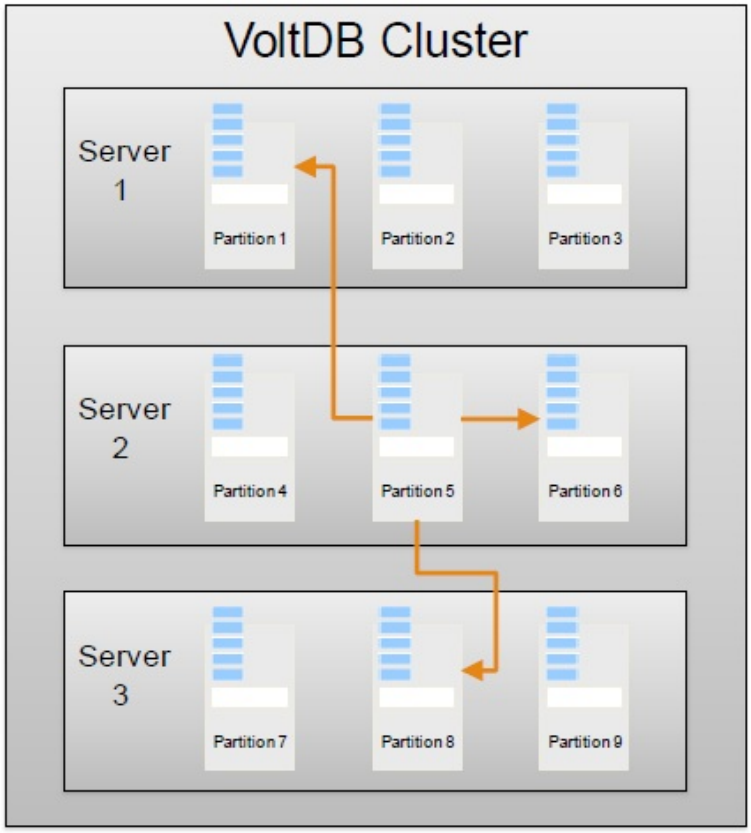


Figura 3.3: VoltDB *Multiple Partition Transaction* Fonte: (Voltdb 2012)

confiança, o VoltDB pode ser configurado para não exigir qualquer autenticação no acesso aos dados, poupando preciosos milissegundos no processo de autenticação (Figura 3.4). É claro que os utilizadores finais são atendidos por outros servidores, que esses sim, efetuam as autenticações e proteções adequadas.

3.2.2 Escalabilidade

O VoltDB permite o crescimento elástico vertical e horizontal, com poucas ou nenhumas alterações no esquema da base de dados e no código das aplicações que serve. Na implementação de uma solução potencialmente escalável, devem ter-se em conta os seguintes princípios:

Ligação cliente / servidor Os clientes da base de dados devem estabelecer uma ligação com todos os *nodes* do *cluster* em simultâneo, sendo atendido pelo *node* com mais disponibilidade (*multiple partition transaction*), ou aquele que dispõe a partição com os dados pretendidos (*single partition transaction*) (Figura 3.5).

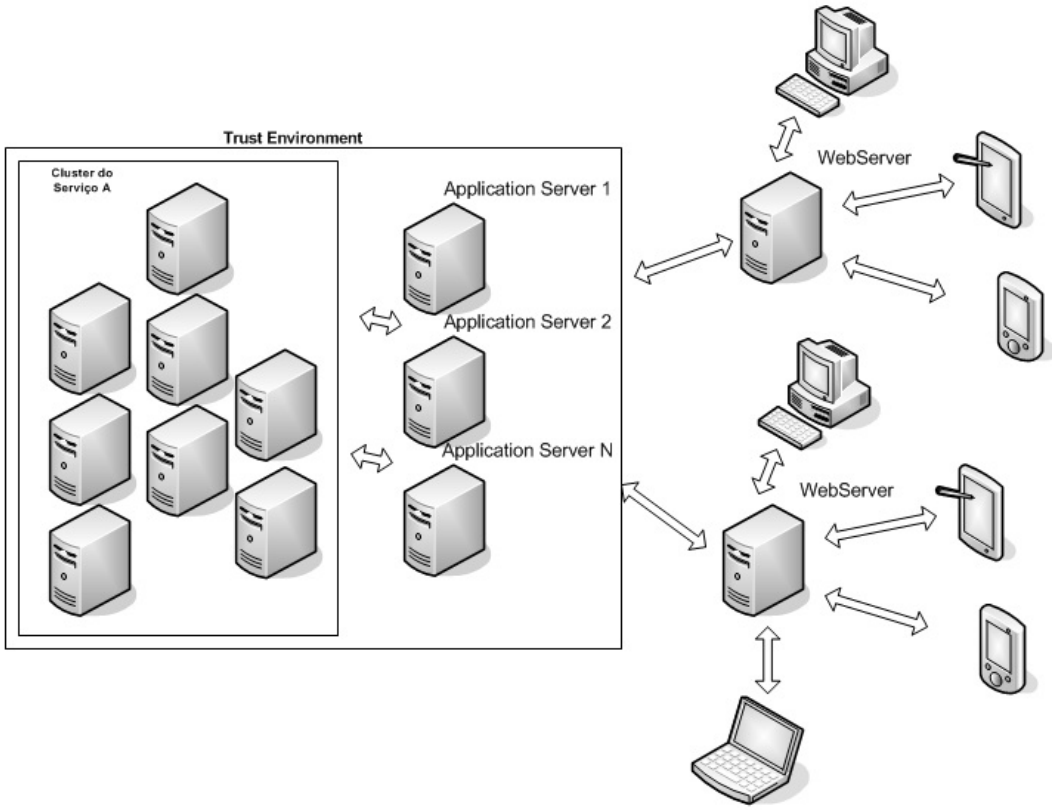


Figura 3.4: Processo típico de acesso aos dados, desde o cliente final até à base de dados, onde entre os *application servers* e os *nodes* do *cluster* não é necessária qualquer autenticação. Fonte: Elaboração própria

Tabelas particionadas e tabelas replicadas O VoltDB contempla dois tipos de tabelas: particionadas e replicadas. Se a tabela apresenta um grande crescimento ou uma grande manutenção, pode e deve ser particionada pelos diferentes *nodes*. Neste sistema a gestão do particionamento é automática após indicação de uma chave única de acesso à partição. Se a tabela apresenta uma reduzida manutenção e dimensão, pode ser replicada por todos os *nodes*, reduzindo assim as transações *multi-partition* com custos de processamento superiores às transações *single-partition*.

Alta disponibilidade O VoltDB foi desenhado de origem para fornecer alta disponibilidade através do *K-safety, network fault detection* e *live node rejoin*.

K-safety* e *Network fault detection Cada *node* pode ter diversas réplicas, de acordo com o nível de segurança exigido. As transações são executadas sincronamente entre as diferentes réplicas, oferecendo assim uma grande disponibilidade e segurança, mesmo após falhas de diversa ordem nos servidores e na rede de dados.

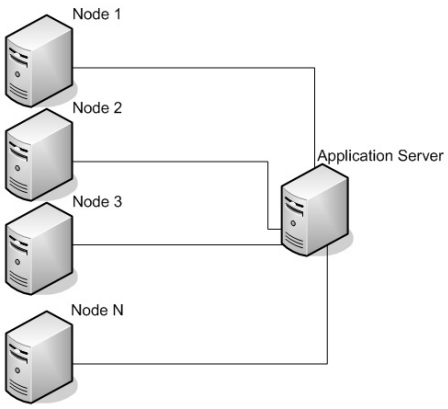


Figura 3.5: A ligação do cliente a todos os *nodes* do *cluster* é uma boa prática de funcionamento. Fonte: Elaboração própria

Live node rejoin Após a reparação de uma falha, o node envolvido, pode juntar-se ao cluster através da operação rejoin, sem necessidade de parar qualquer serviço. O *node* readmitido começará a servir transações normalmente logo após a atualização da sua partição a partir de outra réplica.

Durability Apesar de todos os mecanismos desenvolvidos para reduzir o *downtime*, o VoltDB desenvolveu ferramentas para recuperação da base de dados de uma forma rápida e precisa após a ocorrência de um *crash*, nomeadamente *snapshotting* e *command logging*.

Snapshots É possível definir a elaboração de *snapshots* da base de dados para disco rígido a intervalos pré-programados, que permitem recuperar os dados até esse ponto.

Command Logging Complementarmente aos *snapshots*, existe um log de comandos efetuados, que permite repor a base de dados desde o último *snapshot* até à última alteração efetuada. As bases de dados tradicionais normalmente efetuam um log ao nível dos registos, que implica a escrita em disco de todas as alterações na base de dados. Com o log dos comandos e, pelo facto de serem realizados sequencialmente, é possível repor as alterações à base de dados com um reduzido custo de *logging* .

Real-Time Analytics Diversas técnicas foram incorporadas para que este sistema consiga responder às exigências de processamento em tempo real, nomeadamente a utilização do MapReduce e a materialização de views.

Utilização do MapReduce De uma forma muito simplista, num acesso *multi-partition*, cada partição processa as informações constantes nas suas tabelas e envia os dados para o *node* responsável por agregar o resultado final.

***Views* materializadas** Uma funcionalidade muito interessante é a existência de *views* materializadas, em que as *views* disponibilizam a informação em tempo real, sem haver a necessidade de serem populacionadas, no momento da sua invocação, o que as torna extremamente rápidas.

Cluster Management Com o *cluster* a funcionar (*on the fly*), é possível alterar *stored procedures*, adicionar/remover tabelas, substituir *nodes* avariados, adicionar ou remover *nodes* e distribuir as partições pelos *nodes* automaticamente.

Tabelas *Key-Value* Como já existem diversas aplicações desenvolvidas em ambiente NoSql, a VoltDB não deixou de prever a necessidade da sua migração. Nesse sentido, disponibiliza adicionalmente a possibilidade de utilização de tabelas com dois valores, a chave de acesso e o valor a guardar.

3.2.3 *Benchmarks* realizados anteriormente

Existem já alguns testes realizados com o VoltDB, nomeadamente aqueles realizados por (International 2010) e por (Schwartz 2012)

3.2.3.1 Testes realizados pela SGI

(International 2010) A *Silicon Graphics International*, fabricante de estações e servidores de alto desempenho, testou a base de dados VoltDB1.1 nos seus equipamentos. O teste de escalabilidade horizontal do sistema, com o incremento do número de clientes e o número de servidores, até 6 clientes e 30 *nodes*, teve como resultado a marca de 3370000 TPS. De acordo com o gráfico apresentado na Figura 3.6, o desempenho foi linear ao crescimento do *cluster*.

3.2.3.2 Testes realizados pela Percona

(Schwartz 2012)

(Schwartz 2012) elabora testes mais profundos do que os referidos no ponto anterior. A base de dados é testada no seu crescimento horizontal com diferentes fatores de segurança. São efetuados testes sem redundância de *nodes*, ou seja, sem segurança (*K-Safety*=0), com a recomendada segurança que implica a duplicação dos nodes (*K-Safety*=1), e com extra segurança (*K-Safety*=2) onde existem sempre 3 partições replicadas em 3 diferentes nodes. Este estudo demonstra que a base de dados atinge o mais alto *throughput* com a configuração utilizada, aos 35 *nodes* com *K-Safety*=0, aos 39 nodes com *K-Safety*=1 e aos 46 nodes com *K-Safety*=2, como é indicado na Figura 3.7. Os autores do estudo

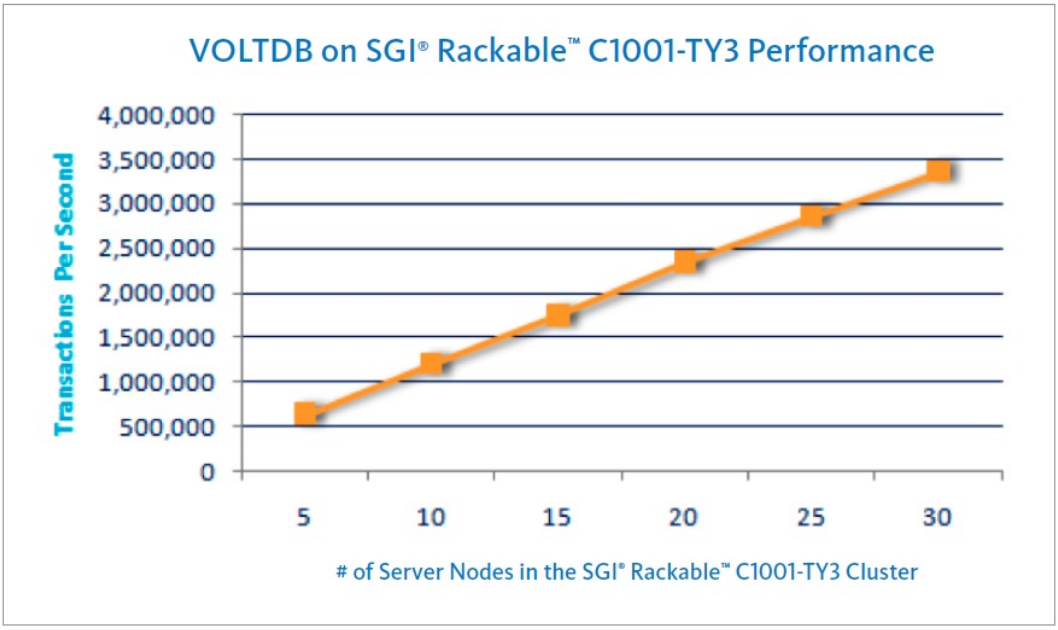


Figura 3.6: Crescimento linear do desempenho com a adição de novos *nodes*. Fonte: (International 2010)

afirmam que "*scaling a synchronously replicated, active-active master, fully ACID, always-consistent database to a 40-server cluster is impressive*" (Schwartz 2012)

3.2.4 Exemplo em produção

A título de exemplo, pode-se referir as implementações desta tecnologia nas empresas *Sakura Internet* e *Booyah*.

3.2.4.1 *Sakura Internet*

Segundo (Writer 2012), a *Sakura Internet*, um *Internet Service Provider* japonês, selecionou o VoltDB com a base de dados de monitorização do tráfego em tempo real fornecido pelos seus serviços. A Sakura disponibiliza serviços de alojamento de páginas de internet e e-mail, bem como a hospedagem de centros de dados na íntegra. A escolha da base de dados Voltdb deveu-se, de acordo com os responsáveis da empresa, ao facto de ser uma *super-fast datastore* que suporta SQL permitindo digerir o massivo tráfego IP através das infraestruturas de comunicações.

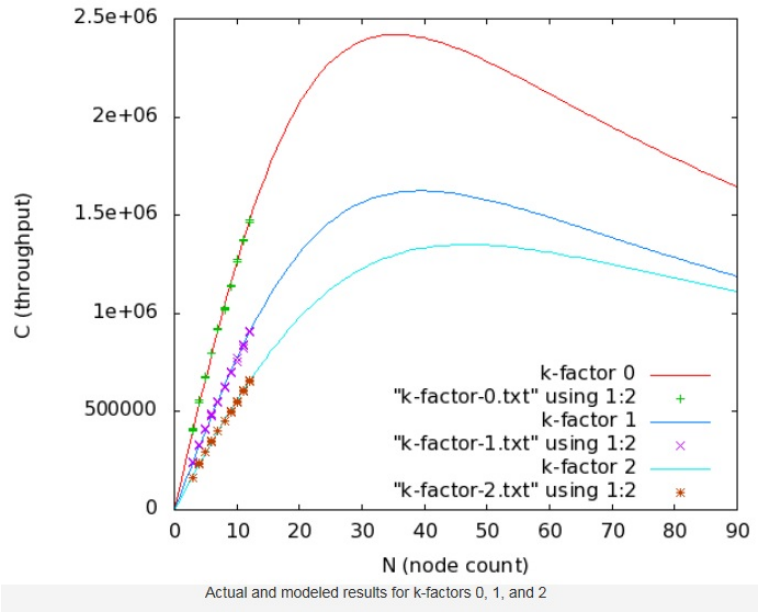


Figura 3.7: Resultado dos testes realizados por (Schwartz 2012). Fonte: (Schwartz 2012)

3.2.4.2 *Booyah*

Segundo (Kazmi 2011), a empresa *Booyah* sediada em São Francisco nos EUA, fornecedora de jogos *online* nas redes sociais, selecionou o VoltDB para lidar com o número massivo de jogadores em simultâneo, que na data do artigo superava os 20 milhões. Os critérios de seleção basearam-se no *throughput*, crescimento e consistência dos dados oferecidos pelo VoltDB, características necessárias ao negócio em tempo real da *Booyah*.

A fazer fé nas informações disponibilizadas pelo VoltDB, e pelos diferentes estudos indicados, a base de dados VoltDB, com a sua nova arquitetura vem colmatar uma lacuna existente entre a necessidade de escalabilidade e a continuidade do *know how* do mundo relacional. O NewSql do VoltDB vem relegar para segundo plano o NoSql, pelo menos ao nível dos programadores aplicacionais, já que no seu núcleo, está recheado de técnicas desenvolvidas nos últimos tempos pelos sistemas NoSql. De certa forma o NewSql do VoltDB é um sistema NoSql, com um interface Sql, em que o teorema CAP de Brewer é contornado, pela rapidez da *Main Memory Database* aliada à consistência dos dados nativa, apesar de utilizar as mesmas técnicas que certos sistemas NoSql aplicam ao nível aplicacional (Figura 3.8).Veja-se como exemplo a reconciliação de dados baseada em versões do registo à semelhança dos sistemas NoSql mas que, em NoSql, é uma árdua tarefa a cargo das aplicações.

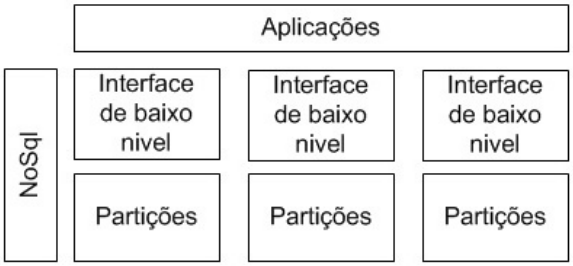


Figura 3.8: O VoltDB, utilizando o chamado NewSql, disponibiliza às aplicações o mesmo interface que os sistemas relacionais tradicionais, apesar de estar recheado de técnicas utilizadas pelos sistemas NoSql. Fonte: Elaboração própria.

3.3 Seleção da base de dados de referência

Para poder de alguma forma aferir o desempenho desta nova arquitetura, foi necessário compará-la com um sistema de arquitetura tradicional, para melhor ilustrar o que se ganha ou perde com a sua utilização. Para efetuar a comparação, procedeu-se a uma seleção de um motor de base de dados de referência no mercado, que obedeceu aos seguintes pressupostos:

Acessível Na seleção, foram excluídos os sistemas que envolvam custos significativos na sua implementação, como seja a exigência de hardware específico ou licenças de utilização.

Open Source Como o VoltDB é *open source*, é lógico selecionar um sistema de referência igualmente *open source*.

Plataforma idêntica O VoltDB está disponível para diversas plataformas, como seja o CentOS a partir da versão 5.6, Ubuntu a partir da versão 10.4 ou ainda o Sun a partir da versão JDK 6 Update 20 (versões de 64 bits). Foi tomado como referência, pela facilidade de instalação e documentação existente, o Ubuntu 10.04 LTS amd64 (*Lucid*) com o kernel Linux 2.6.32-38-generic. A base de dados de referência deve pois correr sobre esta mesma plataforma, por uma questão de equidade.

Base alargada de conhecimento O DBMS selecionado deveria ter uma grande aceitação no mercado com um largo número de instalações e estar bem documentado.

Face aos requisitos expostos, a escolha natural foi o MySql, na sua versão tradicional, para comparação entre VoltDB NewSql e o MySql OldSql. De referir que o MySql não é o objeto de estudo deste trabalho, pelo que não são exploradas as suas potencialidades, até porque já existem novas versões deste popular motor de base de dados com abordagens arquiteturais semelhantes ao VoltDB . O MySql é somente utilizado para comparar duas

arquiteturas distintas, e poder haver um ponto de referência para a nova arquitetura, orientada para OLTP. A escolha da versão do MySql recai para a 5.1.62, por ser aquela que vem nativa no Ubuntu 10.04 LTS.

3.4 A ferramenta de testes estudada (*Bechmark*)

Foram analisadas diversas ferramentas de *benchmark*, nomeadamente o *Hammerora* e o TPC-C/E.

3.4.0.3 *Hammerora*

O *Hammerora* (`http://hammerora.sourceforge.net/`) é uma ferramenta *open source* simples e eficaz, desenvolvida, inicialmente, para testar o DBMS Oracle, mas que hoje permite ligar-se aos principais motores de bases de dados. O problema é que esta ferramenta foi concebida para testar um DBMS tradicional, com a ligação a um único servidor ou a um *midleware*, e não a um *cluster* com as características do VoltDB. Com o VoltDB, é aconselhado que cada cliente estabeleça uma ligação a todos os *nodes*, perfeitamente praticável, já que os clientes diretos são normalmente *applications servers*, mas não o *Hammerora*. Outra limitação à utilização desta ferramenta é a exigência da utilização de *stored procedures* por parte do VoltDB. Pelas limitações referidas, a utilização desta ferramenta exigiria uma profunda reengenharia, o que inviabilizou a sua utilização neste estudo.

3.4.0.4 TPC-C/E

Os testes TPC, são geridos pela *Transaction Processing Performance Council* (`http://www.tpc.org/tpcc/`), que é uma organização certificadora da performance de diferentes sistemas. Um teste TPC pode ser comparável com qualquer outro teste TPC independentemente do *hardware* ou *software* utilizados. Existem diversos testes TPC, sendo o TPC-C o mais utilizado em sistemas de bases de dados tradicionais e o TPC-E a procurar aferir as bases de dados emergentes nomeadamente em sistemas NoSql, e seria o mais adequando para este caso. Para a sua utilização, tem que haver, de alguma forma, o envolvimento desta organização, com a ferramenta adequada a necessitar de uma personalização para o teste pretendido. É claro que estas personalizações de testes têm custos que inviabilizaram a sua utilização neste trabalho. De referir que muitas organizações, evitam os custos dos testes TPC, elaborando testes similares, mas por não serem acreditados, são designados *TPC-Like*. Neste trabalho, nunca foi objetivo a comparação dos resultados com outros já publicados, pelo que não foi vital a utilização dos testes TPC.

Face à panorâmica explorada, foi decidido elaborar a própria ferramenta de testes, com base num exemplo já existente disponibilizado pelo VoltDB.

3.4.1 Ferramenta de testes implementada

Pretende-se testar a base de dados segundo uma ótica de OLTP, e para isso adaptou-se um exemplo existente, para funcionar em circunstâncias semelhantes no VoltDB e no MySql. A ferramenta desenvolvida simula uma votação via telefone num concurso qualquer, muito em voga nos dias de hoje na televisão. Existe uma lista de concorrentes possíveis em que os espetadores podem votar através de uma chamada telefónica, onde indicam somente o número do concorrente preferido. O sistema de *Interactive Voice Response* (IVR) guarda o número de telefone e o número do concorrente numa base de dados para contabilizar o resultado da votação. Concursos deste tipo geram uma grande quantidade de transações num curto espaço de tempo, com a duração a abranger todo o programa televisivo ou somente uns minutos desse mesmo programa. Com base nos dados recolhidos pelo sistema telefónico IVR, é determinado o código da localidade através do indicativo telefónico. Cada votante pode votar até 3 vezes nos concorrentes que desejar. No processo de obtenção do número de concorrente e de número de telefone (ambos aleatórios para efeito de simulação), em 0,1% dos casos é gerado um número de concorrente inexistente, e também em 0,1% dos casos é forçada a repetição do mesmo número de telefone 5 vezes, para obrigar os mecanismos de validação a funcionar. Paralelamente ao processo de votação existe um processo de consulta de tempo ajustável (em segundos), para informação em tempo real do resultado das votações. Trata-se pois da simulação de um sistema OLTP intenso, para um curto período de tempo.

3.4.1.1 Linguagem de programação

A linguagem de programação adotada foi C# desenvolvida no *Microsoft Visual Studio 2010*. A sua seleção foi baseada no *know how* que o mestrando dispõe, bem como a excelente documentação existente.

3.4.1.2 Esquema da base de dados

A base de dados teste é composta de 3 tabelas (Figura 3.9), cujo esquema foi adotado na íntegra: -Tabela *Area_code_state* com as abreviaturas dos diferentes estados que compõem os Estados Unidos da América -Tabela *Contestants* com o nome dos concorrentes -Tabela *Votes* com as votações registadas. Esta tabela é a única que apresenta um crescimento potencial muito grande, pelo que na implementação em VoltDB é particionada pela chave *phone_number*.

3.4.1.3 Esquema de funcionamento

Na Figura 3.10, representa-se o esquema de funcionamento da ferramenta de testes, com um ciclo que gera números de telefone e o número de concorrente, inserindo-os na base de

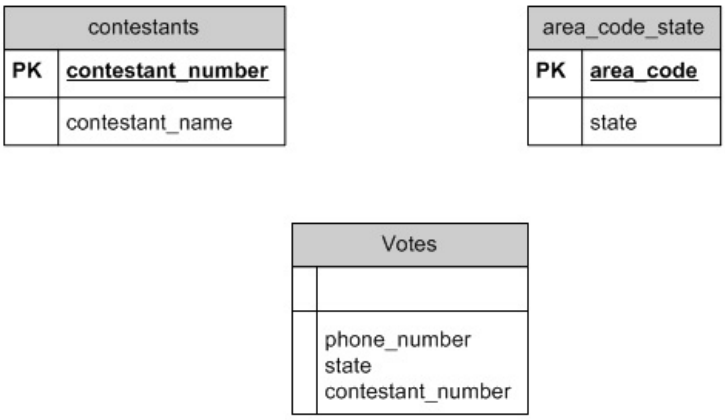


Figura 3.9: Esquema da base de dados de teste, com o particionamento da tabela *Vote*, pela chave *phone_number*. Nota-se a ausência do *referential integrity*, por não ser suportado pelo VoltDB. Fonte: Elaboração própria

dados através de uma *stored procedure*. Paralelamente são apresentados os resultados da votação.

3.4.1.4 *Stored procedures Vote(nTelefone,nConcorrente)*

A *stored procedure* representada esquematicamente na Figura 3.11, verifica se o concorrente existe na tabela *contestants* e se existir, obtém o *state*[1] na tabela *area_code_state* com base no número de telefone. De seguida verifica se já existem mais do que 3 votos para esse número de telefone. Se tudo estiver correto, é registado na tabela o novo voto. De referir que o processo de validação do número máximo de votos por número de telefone, é baseado numa *view* em VoltDB, que por ser materializada é muito rápida. No MySql, a utilização de uma *View*, resulta neste caso num processamento bem mais lento, pelo que foi adotado um simples comando Select.

3.4.2 Outra ferramenta de testes utilizada

Se para os testes comparativos entre o MySql e o VoltDB, foi utilizada a ferramenta de testes em C# adaptada para trabalhar com ambas as plataformas em igualdade de circunstâncias, já para o teste de escalabilidade em VoltDB, não foi possível a sua utilização. Em VoltDB, um cliente, deve efetuar uma ligação a todos os servidores, para melhor aproveitar as capacidades do *cluster*. Com a configuração e versões testadas, essa ligação multi-ponto, nunca funcionou utilizando o C#. Após diversos contactos com o fabricante, que se mostrou sempre disponível e cooperante, foi registado o problema para resolução. Como não foi indicado nenhum prazo de resolução por parte da VoltDB, foi decidido

[1]Nome da região de onde provém o telefonema

avançar com os testes, mas desta vez utilizando uma ferramenta por eles disponibilizada escrita em Java, com um funcionamento semelhante para um problema semelhante, que requereu apenas pequenas configurações. Após alguns testes de avaliação, constatou-se que esta ferramenta apresentava valores análogos à ferramenta desenvolvida em C#, pelo que a sua utilização pode ser comparável.

3.4.3 Descrição dos testes

Os testes foram classificados em Comparação, Escalabilidade, Faul-Tolerance e Evolução. Para os 3 primeiros grupos de testes, o objetivo é a realização do maior número possível de transações durante um período de 5 minutos. Os testes foram repetidos 3 vezes com os mesmos parâmetros, sendo tomado o resultado da média. Em todos os testes, o número de concorrentes é 10 e o número máximo de votos por número de telefone é 3. É registado o número total de transações efetuadas, para cálculo do TPS.

Os clientes utilizados neste *benchmark*, referem-se às máquinas que estabelecem uma ligação com os servidores de bases de dados. Estes clientes em aplicações produtivas, são normalmente *application servers*, que por sua vez, concentram os pedidos de outros servidores como sejam os *web servers* sendo esses que permitem o contacto com clientes finais.

O número ótimo de clientes a utilizar numa solução deste género, está dependente de muitas variáveis, e os valores aqui encontrados, não devem ser extrapolados para outros casos, já que neste caso um cliente não tem outras tarefas que não seja tentar registar o máximo de transações possíveis. Para a elaboração dos testes, recorreu-se ao seguinte hardware:

```
Router
  Cisco WRVS4400NV2 4 portas

Switch
  Gb D-Link DGS-1008D 8 Portas

Servidor
  CPU: I5-650 3.2 GHz 4 Cores 8 Threads
  RAM: 8 GB DDR3
  HDD: RAID 0 - 1,5TB
  NIC: Gb
  OS: Ubuntu 10.04      Kernel Linux 2.6.32-38-generic

Clientes (Application server)
  CPU: Core 2DUO 3.06GHz 2 Cores 4 Threads
  RAM: 3 GB DDR2
  HDD: 160GB
```

```
  NIC: 100Mbit
  OS: Win7 amd64 / Ubuntu 10.04      Kernel Linux 2.6.32-38-generic

Cluster Nodes
  CPU: I3-540 3.06GHz 4 Cores 8 Threads
  RAM: 4 GB DDR3
  HDD: 500GB
  NIC: Gb
  OS: Ubuntu 10.04      Kernel Linux 2.6.32-38-generic
```

3.4.3.1 Comparação

Este teste tem como objetivo comparar as duas arquiteturas: OldSql versus NewSql. Com base num único servidor, que suporta o MySql ou o VoltDB, corre-se o teste com 1,2,3,4,5,6,7 clientes em simultâneo respetivamente. Na Figura 3.12 representa-se esquematicamente o teste. Ambas as plataformas foram afinadas para o maior desempenho possível em detrimento da segurança. A tabela de votos, foi limpa entre cada teste, e no caso do MySql, todos os ficheiros relacionados com a base de dados foram removidos e recriados entre cada configuração de testes (esquema da base de dados + ibdata1). De referir que foi o único conjunto de testes que envolveu as duas plataformas (MySql e VolDB), insidindo os restantes testes somente sobre o VoltDB.

3.4.3.2 Escalabilidade

Este conjunto de testes pretendem de avaliar a performance desta tecnologia, à medida que se efetua o crescimento horizontal do sistema. A necessidade de escalar este sistema horizontalmente resulta no facto do espaço disponível, ser limitado pela RAM da máquina, de nada valendo aumentar a capacidade em disco. Se for necessário mais espaço para conter a base de dados, obrigatoriamente é necessário aumentar o número de máquinas que compõem o *cluster*. Corre-se o teste com combinação de 1,2,3,4,5,6 *nodes* com 1,2,3,4 clientes em simultâneo. Na Figura 3.13 representa-se esquematicamente o teste.

3.4.3.3 *Faul-Tolerance*

Neste caso, pretende-se comparar o desempenho do VoltDB, utilizando, 3 *nodes* e 1 cliente com uma configuração similar utilizando o nível de segurança recomendado, de uma réplica por *node*. Assim é possível aferir a performance com e sem segurança. Sabe-se que quanto maior for o número de réplicas, maior é a segurança e disponibilidade do sistema. Em VoltDB a replicação é dada pelo parâmetro K-Safety, com valores de zero (segurança mínima), 1 (segurança recomendada) ou >1 para segurança extra. Na Figura 3.14, representa-se esquematicamente este teste.

3.4.4 Evolução

Os testes de evolução resultam de um aproveitamento dos ficheiros de log, produzidos pelos testes anteriores, não previstos inicialmente, e por isso não foram alvo de um tratamento sistemático, mas que mesmo assim, permitem algumas ilações interessantes.

Se nos testes anteriores, se procurou apurar o desempenho num curto espaço de tempo (5 minutos), neste caso procurou averiguar-se o ponto de desempenho máximo com uma determinada configuração.

Para a análise da evolução do MySql foi analisado um ficheiro de log dos testes anteriores, selecionado aleatoriamente.

Para o caso do VoltDB aproveitou-se o log de testes que decorreram por mais tempo que o previsto, por erro de introdução do tempo do teste (normalmente 300s). O ficheiro recuperado, referia-se a um teste com 5 *nodes* e um cliente, sem redundância que decorreu por mais de 30 minutos.

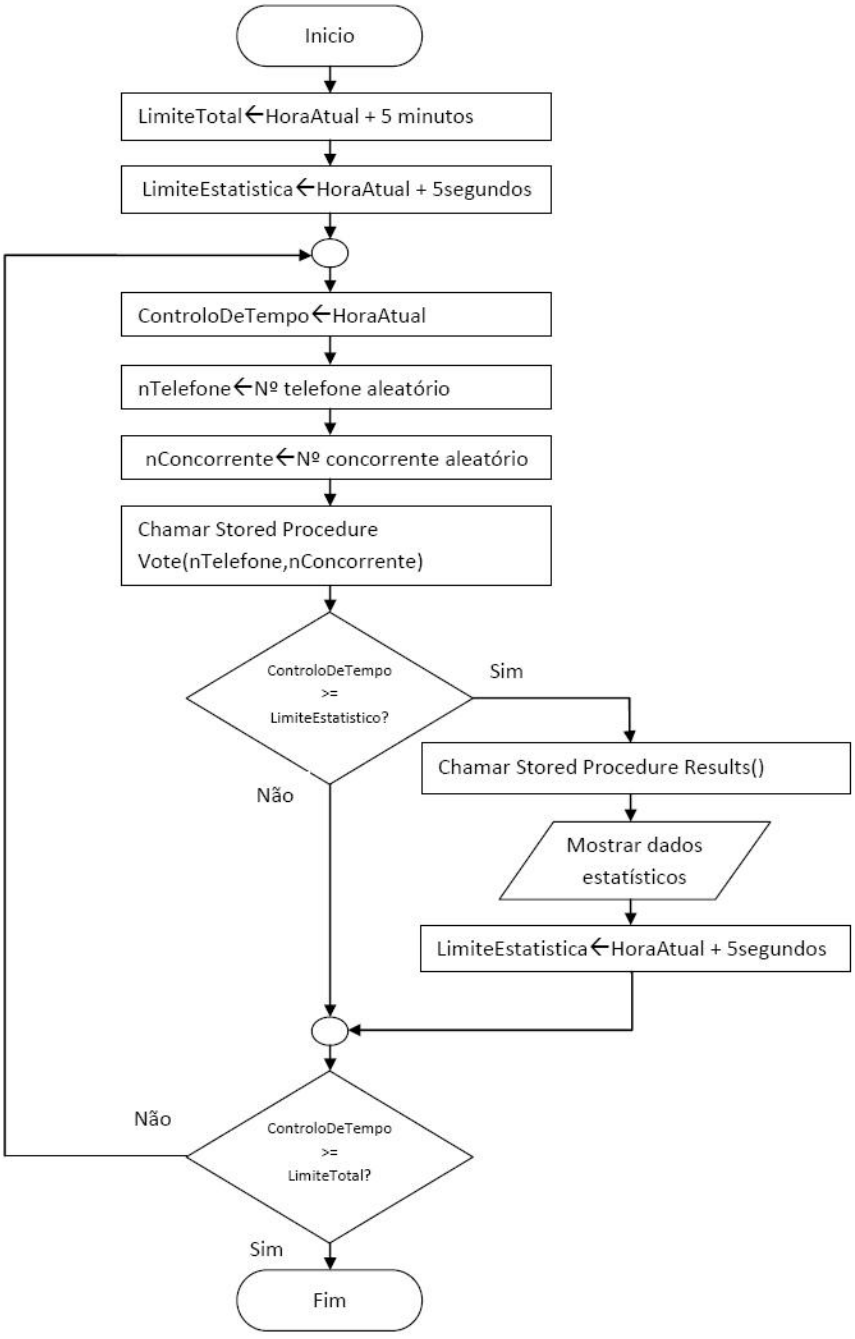


Figura 3.10: Fluxograma do funcionamento do processo de votação, com um tempo total de processamento de 5 minutos, com apresentação de estatísticas de 5 em 5 segundos. Fonte: Elaboração própria

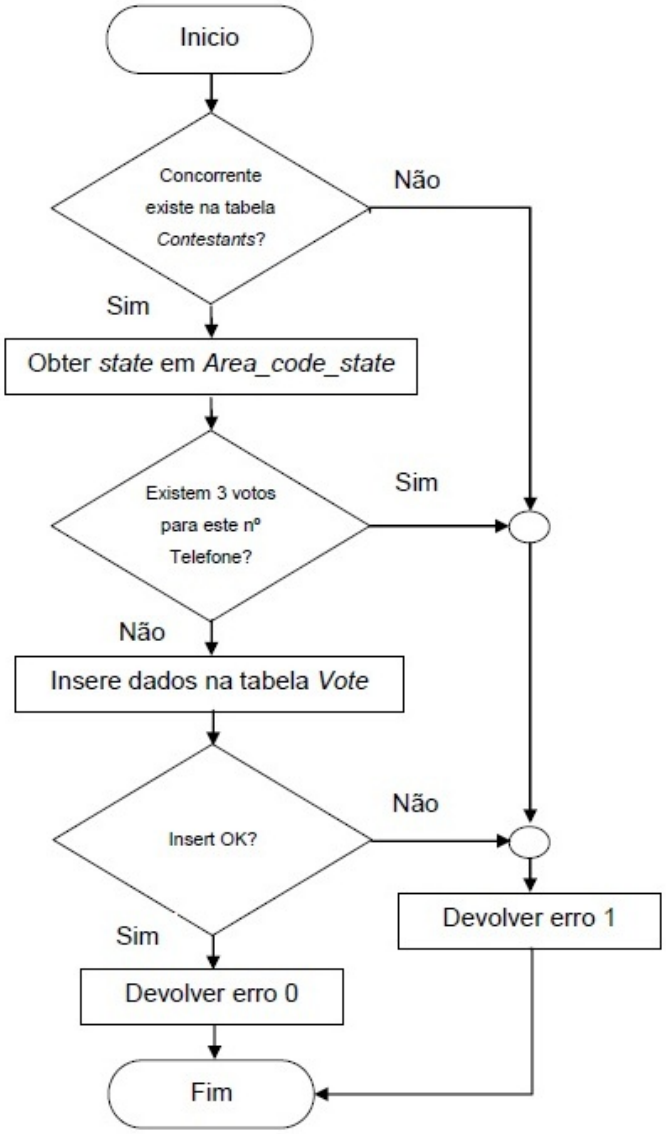


Figura 3.11: *Stored procedure Vote* Fonte: Elaboração própria

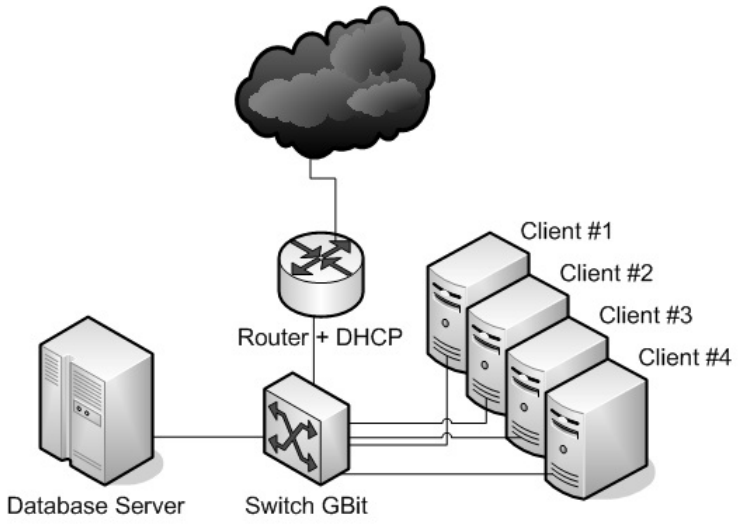


Figura 3.12: Esquema do teste comparativo MySql vs VoltDB, com 1,2,3, 4,5,6,7 clientes em simultâneo, respetivamente. Fonte: Elaboração própria

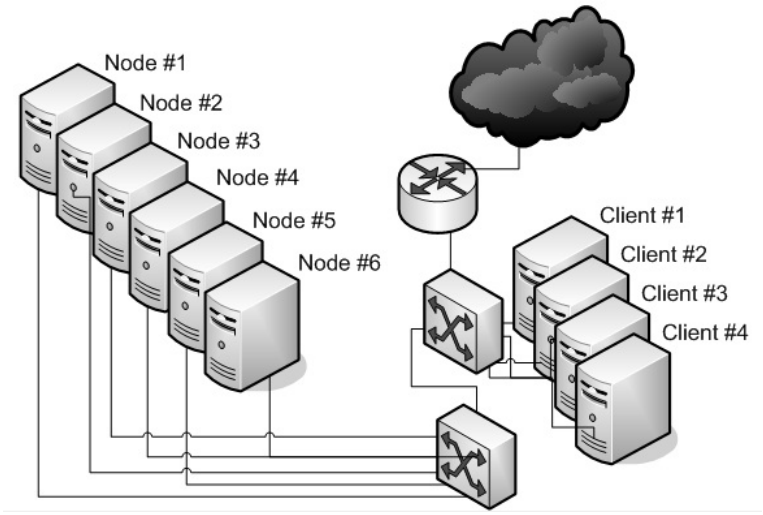


Figura 3.13: Esquema do teste de escalabilidade do VoltDB, com uma combinação de 1,2,3,4,5,6 *nodes* com 1, 2, 3, 4 clientes Fonte: Elaboração própria

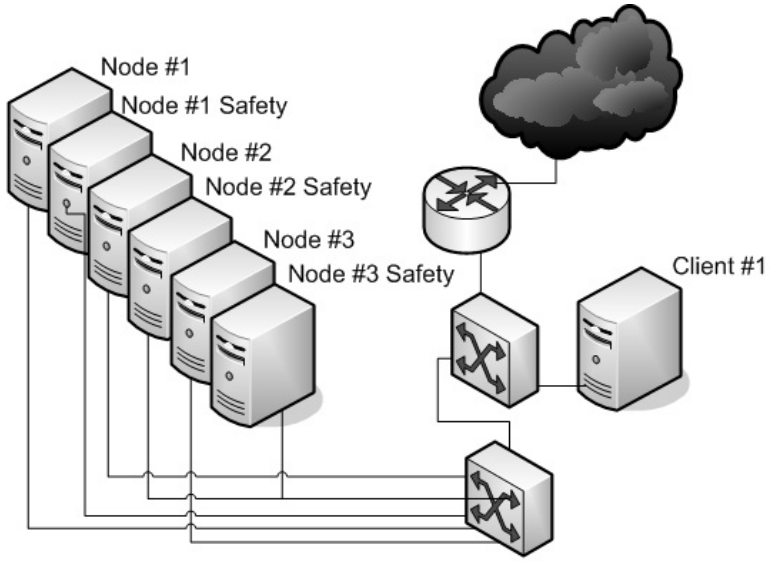


Figura 3.14: Teste de escalabilidade do VoltDB, com *K-Safety* (número de réplicas de uma partição) a variar de 0 a 1. Fonte: Elaboração própria

Capítulo 4

Resultados

Face aos procedimentos descritos, foram registados 114 testes, classificados como Comparação, Escalabilidade e *Faul-Tolerance*, segundo os resultados apresentados no Anexo A. Os ficheiros de log foram aproveitados para se efetuar uma análise da evolução do desempenho ao longo do tempo de ambas as tecnologias, e não somente o resultado obtido ao fim de 5 minutos (Anexo B). Esta fase deste estudo teve uma duração de 4 meses, com início em Março e término em Junho de 2012. Este foi o tempo necessário à apreensão desta nova tecnologia, do estudo/desenvolvimento da ferramenta de *benchmark* e da preparação/realização dos testes. De referir que todo o ambiente de desenvolvimento dos testes foi instalado em máquinas virtuais, recorrendo ao *VirtualBox*. A passagem do ambiente virtual para o real, revelou algumas surpresas que foram sendo ultrapassadas à medida que foram surgindo, com o crescimento do número de máquinas, sendo a mais significativa a sincronização dos relógios das máquinas que compõem o *cluster*. Para dois *nodes* em ambiente virtual não foi necessária qualquer sincronização enquanto que para os mesmos dois *nodes* em ambiente real foi necessário sincronizar os relógios com um relógio de referência na internet. Com a adição de mais *nodes*, essa sincronização revelou-se insuficiente, sendo adotada a sincronização local entre os *nodes*, recorrendo ao serviço *Network Time Server* (NTP).

4.1 Comparação

Utilizando *hardware* e *software* em igualdade de circunstâncias, os resultados da comparação entre as duas tecnologias, são avassaladores. O VoltDB obtém resultados melhores de até 3000% em determinadas configurações (Figura 4.1).

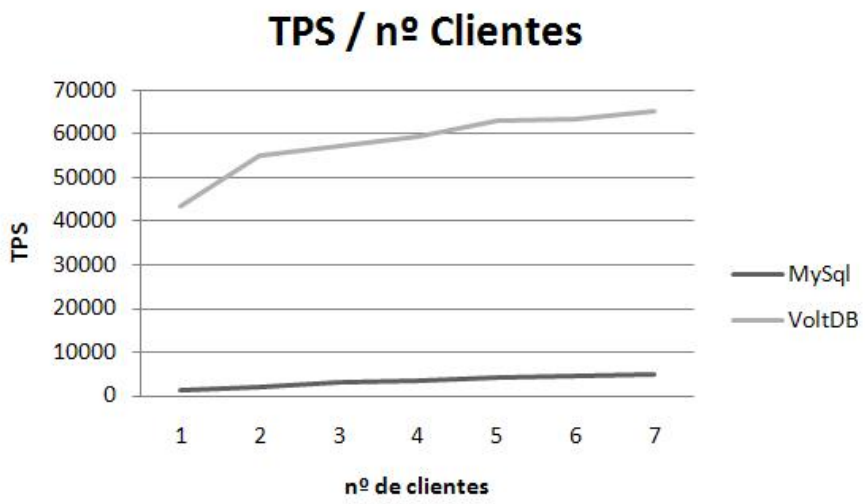


Figura 4.1: Número de TPS alcançados pelas duas tecnologias em igualdade de circunstâncias

Como se pode verificar no gráfico apresentado na Figura 4.2, se formos adicionando clientes a ambas as plataformas, o MySql consegue retirar um melhor partido deste aumento de recursos, mas nunca chegando perto do VoltDB, mesmo com um só cliente. Esta nova tecnologia tira partido de toda a sua potência com uma única máquina, tendo algum ganho com a segunda e sendo de desprezar mais clientes. Estas ilações aplicam-se somente à configuração utilizada, sendo necessário estudar a necessidade de vários clientes caso a caso, de acordo com a complexidade dos processos implementados. Já para o MySql, existem ganhos consideráveis com a utilização de até 3 clientes, com a adição de novos recursos a originar ganhos marginais.

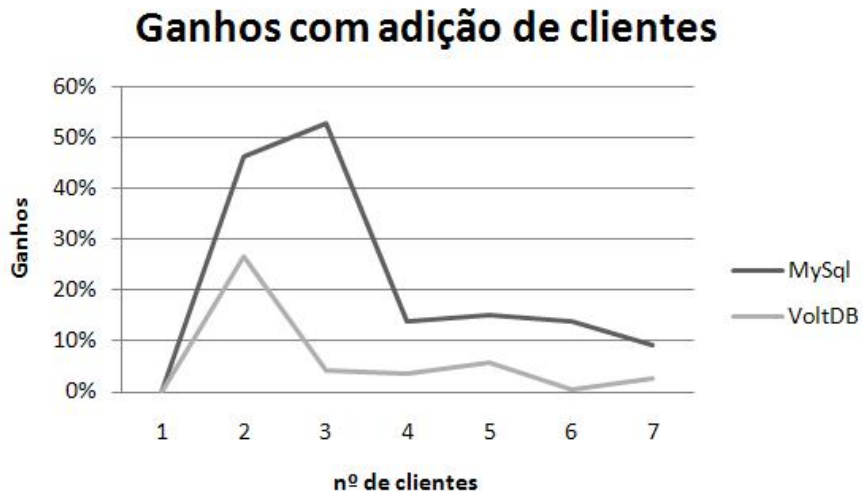


Figura 4.2: Ganhos com a utilização de vários clientes

Após a realização deste teste, foi decidido que nos testes seguinte com o VoltDB, seriam só utilizados até 4 clientes, visto que ficou demonstrado que não é necessário muitos clientes para obter bons resultados.

4.2 Escalabilidade

O gráfico apresentado na Figura 4.3 mostra que a adição de uma segunda máquina ao *cluster*, resulta normalmente numa quebra de desempenho, já que o sistema passa a despender alguns recursos para a replicação / sincronização / gestão de partições. Quanto maior for o número de máquinas que compõem o *cluster*, maior o desempenho obtido, se não for ultrapassado o número de clientes ótimo para essa configuração. Para 2 ou 3 clientes não se obtém melhor desempenho com mais de 5 *nodes*, mas para o caso de 6 *nodes*, o desempenho aumentou sempre até ao quarto cliente.

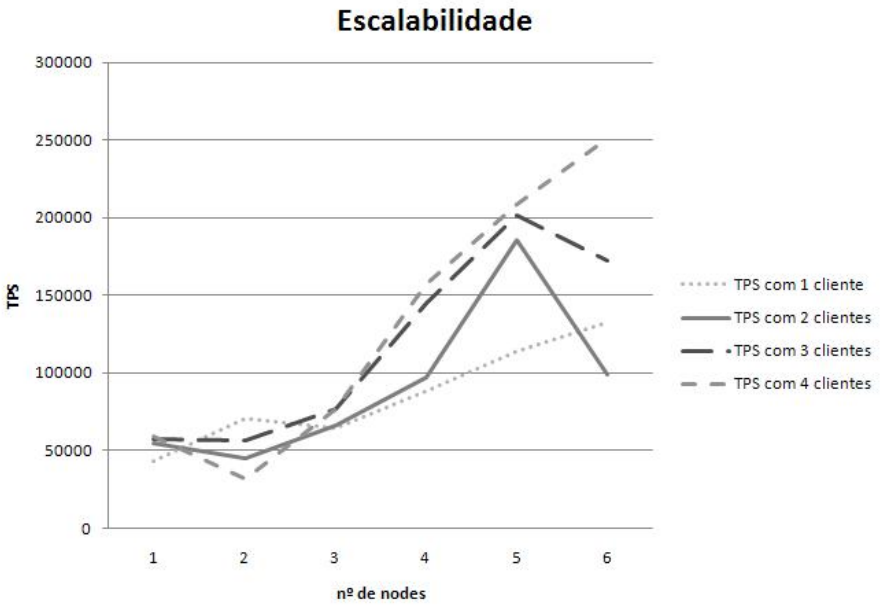


Figura 4.3: Evolução do desempenho com a adição de novas máquinas ao *cluster*, com diferentes cargas de trabalho, resultando um crescimento linear com a configuração certa

Como podemos observar no gráfico da figura 4.4, onde constam os dados anteriores mas numa visão por %, para uma carga constante, variando o número de *nodes*, verificam-se ganhos substanciais a partir da utilização do 4 node, com um ganho de 165%, 251% e 321% respetivamente para 4, 5 e 6 máquinas. Se o espaço em memória não for problema, verifica-se que não vale a pena instalar um cluster com menos de 3 máquinas, pois o seu desempenho não vai melhorar.

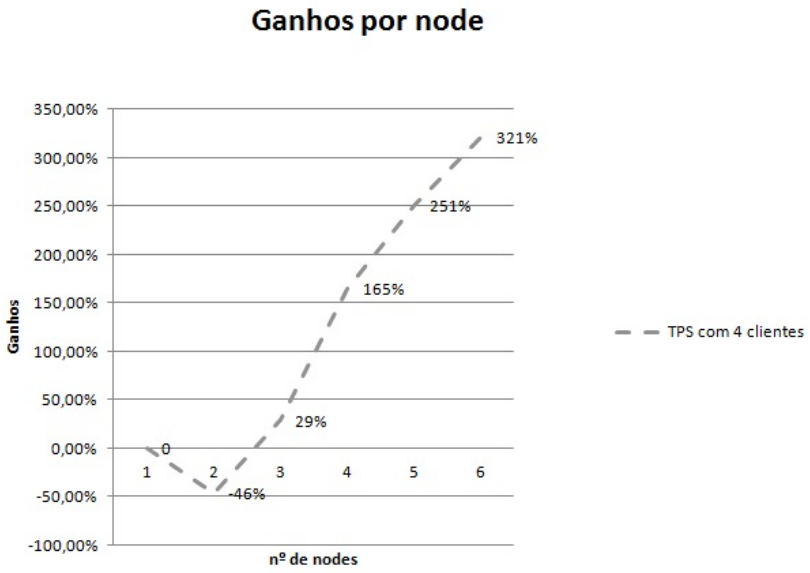


Figura 4.4: Ganhos em % com o crescimento do cluster para uma carga constante de 4 clientes

4.3 Fault-Tolerance

Este teste utiliza 3 *nodes* + 3 réplicas num total de 6, utilizando o nível de segurança recomentado pelo VoltDB (K-Safety=1). No gráfico da Figura 4.5, pode-se observar que face à não utilização de segurança, nota-se uma redução do desempenho mesmo com o aumento do número de clientes. Aumentado a durabilidade dos dados, diminui-se performance do sistema, como seria de esperar, refletindo os custos associados à replicação de todas as estruturas de dados contidas nos *nodes*, bem como a sua gestão. Ao contrário do particionamento dos dados, em que cada *node* contém uma parte dos dados que processa independentemente dos outros *nodes*, no caso da replicação por motivos de segurança, todas as alterações são replicadas para os *nodes* espelhos.

De acordo com a Figura 4.6, quantos mais clientes forem adicionados, maior a redução do desempenho, em comparação com um *cluster* de 3 *nodes* sem segurança.

4.4 Evolução

Ao analisar-se o ficheiro de log do teste efetuado com 5 nodes e um cliente com k-safety=0, podemos verificar nas Figuras 4.7 e 4.8, que o desempenho foi melhorando ao longo do tempo, de uma forma linear até cerca dos 22 minutos, com quase 160000 TPS, e atingindo a espantosa soma de 225 milhões de transações em 30 minutos.

Verifica-se que ao contrário da tecnologia tradicional, o VoltDB consegue durante algum

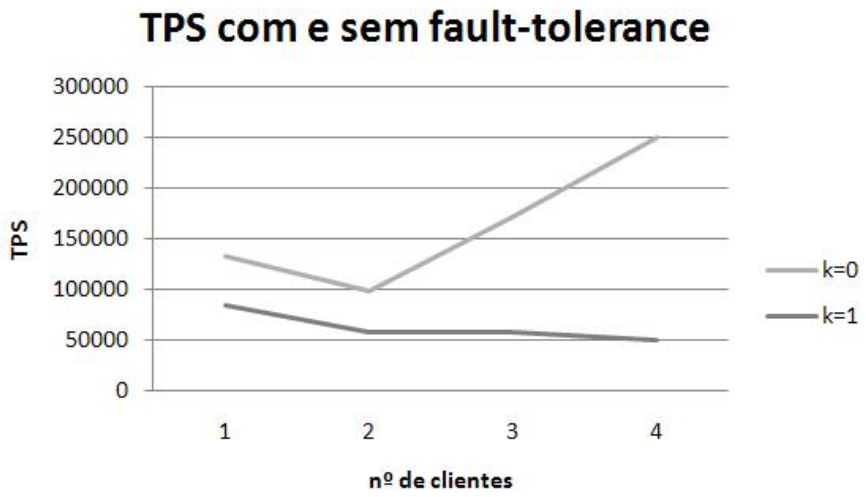


Figura 4.5: Comparação do desempenho do *cluster*, utilizando segurança contra falhas (k=1) e sem segurança (k=0)

tempo aumentar o desempenho do sistema.
Na análise à evolução do MySql ao longo de 5 minutos patente na Figura 4.9, verifica-se que o seu desempenho decresce segundo a linha de tendência representada pela função $y = 1447,4x^{-0,015}$. Para se conseguir obter os resultados alcançados pelo VoltDB de 225 milhões de transações em 30 minutos, o MySql utilizando o mesmo hardware necessitaria de 255 horas, o que só por si demonstra o ganho que se obtém em aplicações OLTP com a nova tecnologia.

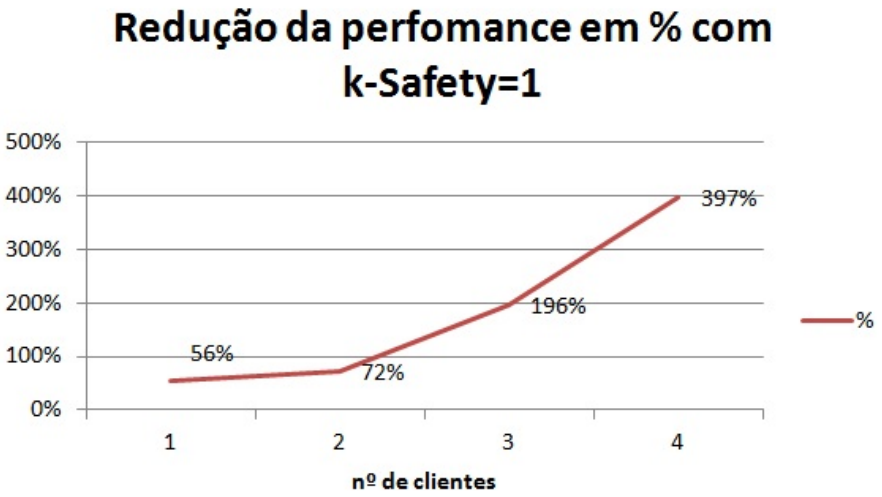


Figura 4.6: Resultado da introdução do fator de segurança K-Safety=1

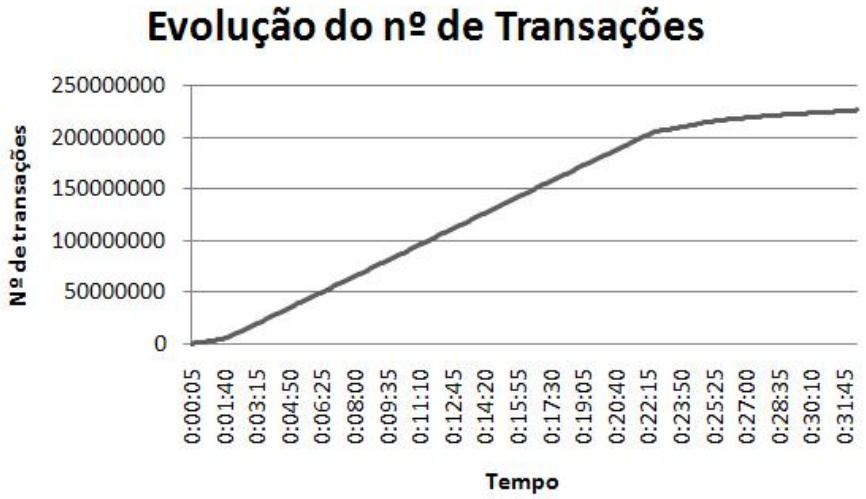


Figura 4.7: Acumulado de transações efetuadas ao longo de 31 minutos e 45 segundos

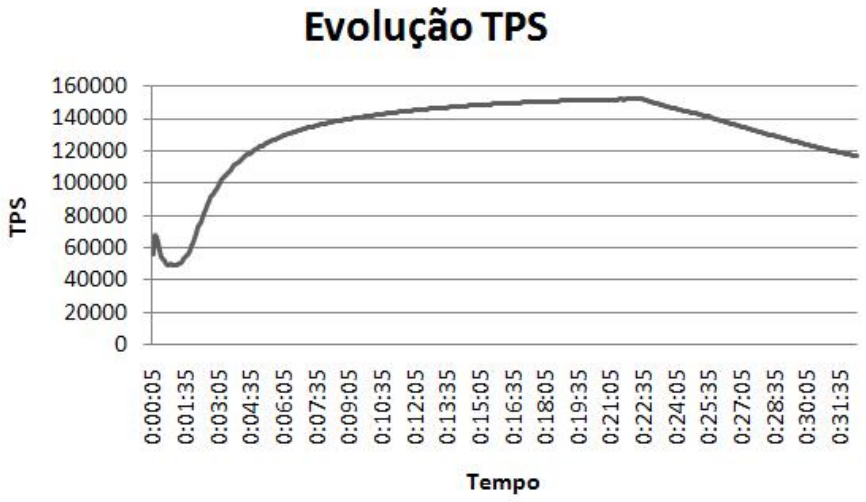


Figura 4.8: Em termos de TPS, foi atingido um máximo aos 22 minutos

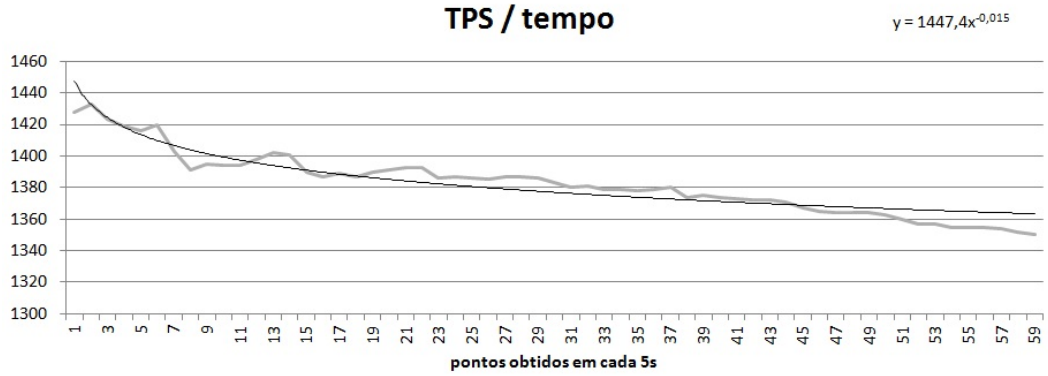


Figura 4.9: Evolução do desempenho MySql ao longo do tempo, com um decréscimo de performance notória

Capítulo 5

Conclusões e trabalho futuro

Comparando o NewSql VoltDb com o OldSql MySql, verifica-se que se está a lidar com desempenhos noutra escala de grandeza. Esta nova tecnologia é orientada para soluções OLTP, e vem preencher a lacuna sentida nos anos mais recentes com a pouca escalabilidade dos sistemas relacionais, não sendo necessário abdicar-se de grande parte das funcionalidades destes últimos sistemas. Se até há relativamente pouco tempo, para criar um sistema de alto desempenho com alta disponibilidade era necessário desenvolver um sistema NoSql, esquecendo quatro décadas de funcionalidades, regressando aos acessos *record-at-time*, similar aos tempos de linguagens como o COBOL, agora já existem soluções que tendo por base técnicas NoSql, disponibilizam um interface Sql, mantendo a continuidade das funcionalidades e *know-how* adquiridos.

Para a implementação de bases de dados que suportam as actividades de *cloud computing*, o NewSql do VoltDB vem oferecer uma alternativa muito aliciante face aos sistemas tradicionais, bem como aos sistemas NoSql, pelos seus baixos custos, pela fácil instalação / desenvolvimento e pela possibilidade de escalar elasticamente.

Cada vez mais, há a necessidade de utilização de dados semiestruturados, ou mesmo não estruturados. Não se deve exigir aos sistemas de gestão de bases de dados a sua manipulação, sob pena de complexificar demasiado o sistema. Estes sistemas de gestão de armazenamento devem sim, permitir a sua organização, e eventualmente o seu armazenamento mas não a sua interpretação, que deve ficar a cargo das aplicações.

Dado o tempo disponível e os recursos existentes para a realização deste trabalho, não foi possível analisar com exaustão todos o sistemas existentes que prometem revolucionar a forma como encaramos o armazenamento de dados, pelo que foi decidido seleccionar uma solução que se apresentasse promissora e acessível.

Os testes elaborados representam uma visão do sistema, podendo ser restritiva em relação ao universo de abordagens possíveis, mas que pelos motivos de limite temporal e recursos (materiais, financeiros e humanos), não são abordados neste trabalho. Outra limitação prende-se com a disponibilidade dos recursos, que pelo facto de terem outras aplicações, não foi possível utilizá-los incondicionalmente.

Nos testes realizados, verificou-se a grande superioridade em todos os parâmetros analisados do VoltDB, com ganhos significativos. A realização de 225 milhões de transações em 22 minutos pelo VoltDB, com uma previsão de 255h (10 dias) por parte do MySql é um exemplo bem expressivo desta nova tecnologia. O NewSql do VoldDB catapulta a performance de uma base de dados acessível a todos, para outra ordem de magnitude, com respostas na ordem das dezenas de milhares de transações por segundo, enquando que o OldSql usado para referência nestes testes em circunstâncias identicas, não passou das centenas de transações por segundo.

De referir que o *cluster* testado utilizou equipamentos de gama baixa que se podem adquirir num supermercado a preços razoáveis. Para escalar a tecnologia antiga com vista a responder ao desafio anterior, seria necessário escalar verticalmente (*Scale-Up*), recorrendo a equipamentos topo de gama, com custos elevados, e não garantindo os resultados pretendidos.

Neste estudo demonstra-se ser possível escalar uma base de dados para híper-dimensão, sem perder o conforto da linguagem SQL.

Propõe-se aprofundar a forma como o VoltDB reage ao *fault-tolerance*, estabelecendo um teste longo, durante o qual, se simulam as mais variadas avarias, e a forma como é possível recuperar os dados, bem como o seu comportamento em termos de performance com a variação do número de *nodes*, assim como analisar a performance versus nível de segurança relacionada com a dimensão do *cluster*.

Bibliografia

Abreu, S. (2001). Isco: A practical language for logic-based construction of heterogeneous information systems, Universidade de Evora, Universidade de Evora, Universidade de Evora.

Brewer, E. A. (2000). Towards robust distributed systems (abstract), *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, ACM, New York, NY, USA, pp. 7–.
URL: *http://doi.acm.org/10.1145/343477.343502*

Brito, R. (2010). Bancos de dados nosql x sgbds relacionais: Análise comparativa, *Faculdade Farias Brito e Universidade de Fortaleza* .

de Carvalho Costa, R. L. & Furtado, P. (2007). An sla-enabled grid datawarehouse, *Proc. 11th Int. Database Engineering and Applications Symp. IDEAS 2007*, pp. 285–289.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store, *Amazon.com* pp. 205–220.

Filip, I., Vasar, C. & Robu, R. (2009). Considerations about an oracle database multi-master replication, *Proc. 5th Int. Symp. Applied Computational Intelligence and Informatics SACI '09*, pp. 147–152.

Garcia-Molina, H. & Salem, K. (1992). Main memory database systems: an overview, *Knowledge and Data Engineering, IEEE Transactions on* **4**(6): 509 –516.

Haerder, T. & Reuter, A. (1983). Principles of transaction-oriented database recovery, *ACM Comput. Surv.* **15**: 287–317.
URL: *http://doi.acm.org/10.1145/289.291*

Ho, R. (2010). Bigtable model with cassandra and hbase.
URL: *http://horicky.blogspot.com/2010/10/bigtable-model-with-cassandra-and-hbase.html*

International, S. G. (2010). Linear voltdb scalability on a 30-node sgi rackable c1001-ty3 cluster for cloud applications. Acedido em 16-05-2012.
URL: *http://www.sgi.com/pdfs/4238.pdf*

Kazmi, S. (2011). Booyah selects voltdb database to support high throughput, low latency gaming applications. Acedido em 24-05-2012.
URL: *http://www.supercomputingonline.com/news/this-years-stories/1054-booyah-selects-voltdb-database-to-support-high-throughput-low-latency-gaming-applications*

Lane, P., Schupmann, V. & Stuart, I. (2005). *Oracle Database - Data Warehouse Guide 10g Release 2 (10.2)*, Oracle.
URL: *http://docs.oracle.com/cd/B19306_01/server.102/b14223.pdf*

Michael, M., Moreira, J. E., Shiloach, D. & Wisniewski, R. W. (2007). Scale-up x scale-out: A case study using nutch/lucene, *Proc. IEEE Int. Parallel and Distributed Processing Symp. IPDPS 2007*, pp. 1–8.

Minhas, U. F., Liu, R., Aboulnaga, A., Salem, K., Ng, J. & Robertson, S. (2012). Elastic scale-out for partition-based database systems, *International Workshop on Self-Managing Database Systems (SMDB '12), ICDE Workshops*.

Nardon, F. B. (1996). Estudo e construcao de um sistema gerenciador de banco de dados dedutivo, *Universidade Federal do Rio Grande* .

News, I. R. (2003). Former ibm fellow edgar (ted) codd passed away on april 18. Acedido em Fevereiro 2012.
URL: *http://www.research.ibm.com/resources/news/20030423_edgarpassaway.shtml*

Pedone, F., Schiper, N. & Armendariz-Inigo, J. E. (2011). Byzantine fault-tolerant deferred update replication, *Dependable Computing, Latin-American Symposium on* **0**: 7–16.

Ramakrishnan, R. & Ullman, J. D. (1993). A survey of research on deductive database systems, *Journal of Logic Programming* **23**: 125–149.

Rugaber, S. & Doddapaneni, S. (1993). The transition of application programs from cobol to a fourth generation language, *Proc. Conf Software Maintenance ,1993. CSM-93*, pp. 61–70. Não é para imprimir.

Schwartz, B. (2012). Is voltdb really as scalable as they claim?, MySQL Performance Blog. Acedido em 17-05-2012.
URL: *http://www.mysqlperformanceblog.com/2011/02/28/is-voltdb-really-as-scalable-as-they-claim/*

Stonebraker, M. (2011). New sql: An alternative to nosql and old sql for new oltp apps.
URL: *http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext*

Stonebraker, M. & Cattell, R. (2011). 10 rules for scalable performance in 'simple operation' datastores, *Commun. ACM* **54**: 72–80.
URL: *http://doi.acm.org/10.1145/1953122.1953144*

Stonebraker, M. & Hellerstein, J. M. (2005). What goes around comes around.

Suposicao de crescimento facil do Facebook e questionada, diz analista (2012). http://g1.globo.com/tecnologia/noticia/2012/02/suposicao-de-crescimento-facil-do-facebook-e-questionada-diz-analista.html, acedido em 10-04-2012.
URL: *http://g1.globo.com/tecnologia/noticia/2012/02/suposicao-de-crescimento-facil-do-facebook-e-questionada-diz-analista.html*

Tellez, E. S., Ortiz, J. & Graff, M. (2007). Pgsync: A multiple-reader/single-writer table replication tool for high loaded distributed relational sql databases, *Proc. Electronics, Robotics and Automotive Mechanics Conf. CERMA 2007*, pp. 735–739.

Vogels, W. (2008). Eventually consistent - revisited, *All Things Distributed* .
URL: *http://www.allthingsdistributed.com/2008/12/eventually_consistent.html*

Voltdb (2012). *VoltDB Technical Overview*, VoltDB, Inc.

von der Weth, C. & Datta, A. (2011). Multi-term keyword search in nosql systems, (99). Early Access.

Writer, C. S. (2012). Voltdb provides real-time internet traffic monitoring, analysis for sakura. Acedido em 17-05-2012.
URL: *http://telecoms.cbronline.com/news/voltdb-provides-real-time-internet-traffic-monitoring-analysis-for-sakura-050112?utm_source=twitterfeed*

Zaniolo, C. (1983). The database language gem, *SIGMOD Rec.* **13**: 207–218.
URL: *http://doi.acm.org/10.1145/971695.582226*

Anexos

Anexo A

Configuração MySql

A instalação do MySql foi realizada recorrendo ao instaladores de pacotes *Synaptic* do Ubuntu. O objetivo dos testes foi obter o melhor desempenho, dado o hardware disponível. Assim, foi utilizado um controlador de disco sata, configurado para RAID 0 contendo 3 discos de 500GB. Foi utilizado o motor InnoDB e foram alterados os seguintes parâmetros no ficheiro my.conf:

innodb_buffer_pool_size Representa a quantidade de memória principal, reservada para guardar os dados lidos / a escrever no disco rígido. Utilizar uma grande área de *buffer*, reduz significativamente os acessos ao disco, melhorando o desempenho do sistema.
Valor configurado: innodb_buffer_pool_size = 4G

innodb_additional_mem_pool_size Zona de memória reservada para guardar as estruturas diversas da organização da base de dados. Se estas estruturas não poderem ser mantidas na memória principal, elas serão mantidas em memórias secundárias, com grandes latências.
Valor configurado: innodb_additional_mem_pool_size=16M

innodb_flush_log_at_trx_commit O MySQL guarda o log das transações em disco, de forma a poderem ser reconstruídas em caso de falha do sistema. Este parâmetro indica que a escrita deve ser efctuada após um *commit* ou periodicamente, ou ainda não efetuar a escrita em log (valor=0). Ao desabilitar a escrita em log das transações, melhora o desempenho, mas reduz-se a segurança.
Valor configurado: innodb_flush_log_at_trx_commit=0

Anexo B

Configuração VoltDB

A instalação do VoltDB não é por enquanto disponibilizada pela Canonical na distribuição Ubuntu do sistema operativo Linux, mas a sua instalação é relativamente simples. Para além de desempacotar os ficheiro obtido no site do VoltDB, é necessário necessário instalar os pacotes openjdk-6-jdk e ntp. O pacote openjdk-6-jdk é a biblioteca necessária ao java e o pacote ntp é o serviço de sincronização do relógio.

Sincronização do relógio É um requisito obrigatório, ter os relógios de todas as máquinas que constituem o *cluster*, devidamente sincronizados. Para isso utiliza-se utilitário NTP, para sincronizar os relógios locais, com um relógio de referência disponível na rede. Neste caso é mais importante que os relógios estejam todos sincronizados entre si, do que sincronizados com um relógio de referência.
Exemplo de configuração do serviço NTP em todas as máquinas do *cluster* através do ficheiro ntp.conf.

```
# /etc/ntp.conf, configuration for ntpd; see ntp.conf(5) for help
server ntp.ubuntu.com iburst
peer 192.168.10.51 iburst burst minpoll 4 maxpoll 4
peer 192.168.10.55 iburst burst minpoll 4 maxpoll 4
peer 192.168.10.56 iburst burst minpoll 4 maxpoll 4
peer 192.168.10.57 iburst burst minpoll 4 maxpoll 4
peer 192.168.10.58 iburst burst minpoll 4 maxpoll 4
peer 192.168.10.59 iburst burst minpoll 4 maxpoll 4
peer 192.168.10.60 iburst burst minpoll 4 maxpoll 4
peer 192.168.10.61 iburst burst minpoll 4 maxpoll 4
server 127.127.1.0
```

Com esta configuração, as máquinas vão eleger um coordenador de sincronização e depois todas vão desenvolver esforços para acertar os respetivos relógios. Quanto maior for o

número de máquinas, mais tempo vai ser necessário para a convergência horária, não sendo de estranhar levar 1/2 hora para 8 máquinas.

Deployment.xml Após o esquema da base de dados estar devidamente criada, para fazer variar o número de *nodes*, basta alterar o ficheiro de configuração deployment.xml e adicionar nos clientes mais uma ligação para esses *nodes*. Alterando o parâmetro *hostcount*, define-se a quantidade de máquinas que constituem um *cluster*. Ao nível da segurança, o parâmetro kfactor, indica o número de réplicas de cada *node*.
Exemplo de ficheiro de configuração deployment.xml, com duas máquinas e nenhuma réplica, portanto segurança mínima.

```
<?xml version="1.0"?>
<deployment>
    <cluster hostcount="2" sitesperhost="2" kfactor="0" />
    <httpd enabled="true">
        <jsonapi enabled="true" />
    </httpd>
</deployment>
```

Anexo C

Resultados dos testes

Núm	Tipo	Descrição	#Nodes	#Cli	K	Soma TPS-Iteração #1							Soma TPS-Iteração #2							Soma TPS-Iteração #3							TPS Total	Média TPS
						Cli#1	Cli#2	Cli#3	Cli#4	Cli#5	Cli#6	Cli#7	Cli#1	Cli#2	Cli#3	Cli#4	Cli#5	Cli#6	Cli#7	Cli#1	Cli#2	Cli#3	Cli#4	Cli#5	Cli#6	Cli#7		
1	Comp	MySql	1	1	na	1387							1402							1350							4139	1380
2	Comp	MySql	1	2	na	1039	998						1057	997						1012	945						6048	2016
3	Comp	MySql	1	3	na	1145	1077	1219					1028	945	982					976	914	963					9249	3083
4	Comp	MySql	1	4	na	1007	964	1049	917				893	841	893	780				838	798	846	704				10530	3510
5	Comp	MySql	1	5	na	1050	987	964	1014	864			827	812	760	830	687			699	682	669	675	610			12130	4043
6	Comp	MySql	1	6	na	981	994	894	918	1008	867		797	780	746	750	752	660		623	628	614	584	654	582		13832	4611
7	Comp	MySql	1	7	na	915	992	998	904	901	912	816	681	735	706	661	640	696	630	579	602	575	533	529	589	514	15108	5036
8	Comp	VoltDB	1	1	0	43330							51248							35806							130384	43461
9	Comp	VoltDB	1	2	0	30083	23064						26932	28913						29036	26913						164941	54980
10	Comp	VoltDB	1	3	0	25685	26739	5.028					27758	6911	23087					20663	11086	25210					172167	57389
11	Comp	VoltDB	1	4	0	21910	4609	24919	9716				11594	24492	1503	21433				3600	26713	8148	19917				178554	59518
12	Comp	VoltDB	1	5	0	25699	3956	24408	4413	10733			17603	8506	22958	5246	5620			22180	5267	23922	999	7651			189161	63054
13	Comp	VoltDB	1	6	0	10234	2919	20610	22510	5378	1643		14435	5969	399	23663	3948	16262		1907	939	14933	4340	2231	4	17872	190275	63425
14	Comp	VoltDB	1	7	0	10397	2483	9109	19692	7207	2397	16366	18899	20410	194	7416	438	2597	1315	27219	1160	2173	14506	1189	5	4741 3165	195616	65205
15	Escal	VoltDB	2	1	0	66620							76430							69854							212904	70968
16	Escal	VoltDB	2	2	0	14253	35738						19043	15051						18567	33042						135694	45231
17	Escal	VoltDB	2	3	0	19049	5994	18657					12288	4683	9	12265				10210	10709	33508					169519	56506
18	Escal	VoltDB	2	4	0	8501	4728	5847	4996				4751	21357	6418	8071				6937	9905	8230	6703				96444	32148
19	Escal	VoltDB	3	1	0	63944							44395							85312							193651	64550
20	Escal	VoltDB	3	2	0	53075	46572						11549	9087						69914	9017						199214	66405
21	Escal	VoltDB	3	3	0	29098	26086	26073					25162	24771	24638					24936	24162	24835					229761	76587
22	Escal	VoltDB	3	4	0	16651	19945	18969	17589				17003	18041	17580	17339				24268	21833	21015	20481				230714	76905
23	Escal	VoltDB	4	1	0	105266							100401							58642							264309	88103
24	Escal	VoltDB	4	2	0	45841	48828						44151	37444						60557	55101						291922	97307
25	Escal	VoltDB	4	3	0	49904	48902	48790					46679	44184	44274					50139	51476	50233					434581	144860
26	Escal	VoltDB	4	4	0	42724	44080	41310	42045				42699	41982	40238	40703				42092	42139	40204	12175				472391	157464
27	Escal	VoltDB	5	1	0	111402							113629							115666							340697	113566
28	Escal	VoltDB	5	2	0	90911	92201						90784	94743						93950	95309						557898	185966
29	Escal	VoltDB	5	3	0	65522	65871	68094					66091	66573	69219					66796	67329	68279					603774	201258
30	Escal	VoltDB	5	4	0	55746	55887	50114	48316				54539	55071	48018	49064				55664	55861	49441	48307				626028	208676
31	Escal	VoltDB	6	1	0	137328							138303							122696							398327	132776
32	Escal	VoltDB	6	2	0	64548	56452						47509	43166						50580	35054						297309	99103
33	Escal	VoltDB	6	3	0	58685	57551	47177					14778	78915	89849					45981	53515	71566					518017	172672
34	Escal	VoltDB	6	4	0	69480	70053	55041	56340				69480	70053	55041	56340				69985	70517	55381	54229				751940	250647
35	FaultT	VoltDB	6	1	1	77698							100566							77890							256154	85385
36	FaultT	VoltDB	6	2	1	25852	31854						29276	29048						31490	25718						173238	57746
37	FaultT	VoltDB	6	3	1	19704	19023	21064					16927	17040	18503					20415	21637	20763					175076	58359
38	FaultT	VoltDB	6	4	1	12603	12712	14064	14074				11796	11908	11832	11827				11727	12048	13412	13412				151415	50472

Anexo D

Log de teste MySql

Evolução MySql ao longo de 5 minutos

Tempo	TPS	Tempo	TPS
0:00:05	1428	0:02:35	1380
0:00:10	1433	0:02:40	1381
0:00:15	1423	0:02:45	1379
0:00:20	1419	0:02:50	1379
0:00:25	1416	0:02:55	1378
0:00:30	1420	0:03:00	1379
0:00:35	1403	0:03:05	1380
0:00:40	1391	0:03:10	1374
0:00:45	1395	0:03:15	1375
0:00:50	1394	0:03:20	1374
0:00:55	1394	0:03:25	1373
0:01:00	1398	0:03:30	1372
0:01:05	1402	0:03:35	1372
0:01:10	1401	0:03:40	1371
0:01:15	1390	0:03:45	1367
0:01:20	1387	0:03:50	1365
0:01:25	1389	0:03:55	1364
0:01:30	1387	0:04:00	1364
0:01:35	1390	0:04:05	1364
0:01:40	1391	0:04:10	1363
0:01:45	1393	0:04:15	1360
0:01:50	1393	0:04:20	1357
0:01:55	1386	0:04:25	1357
0:02:00	1387	0:04:30	1355
0:02:05	1386	0:04:35	1355
0:02:10	1385	0:04:40	1355
0:02:15	1387	0:04:45	1354
0:02:20	1387	0:04:50	1352
0:02:25	1386	0:04:55	1350
0:02:30	1383	00:05:00	0

Anexo E

Log de teste VoltDB

Tempo	Transações	TPS	Tempo	Transações	TPS	Tempo	Transações	TPS	Tempo	Transações	TPS	Tempo	Transações	TPS	Tempo	Transações	TPS
0:00:05	280376	56391	0:08:15	68142801	137669	0:16:25	147256561	149502	0:24:35	212890561	144335	0:27:15	219501791	134254			
0:00:10	679433	68100	0:08:20	68950730	137908	0:16:30	148068202	149567	0:24:40	213248360	144089	0:27:20	219642464	133930			
0:00:15	998435	66665	0:08:25	69762994	138151	0:16:35	148872787	149624	0:24:45	213587146	143832	0:27:25	219751575	133597			
0:00:20	1254761	62804	0:08:30	70569150	138377	0:16:40	149673027	149677	0:24:50	213924191	143575	0:27:30	219905150	133278			
0:00:25	1460016	58457	0:08:35	71375494	138599	0:16:45	150481415	149736	0:24:55	214340279	143374	0:27:35	220082032	132982			
0:00:30	1624352	54190	0:08:40	72164505	138784	0:16:50	151288278	149794	0:25:00	214599630	143069	0:27:40	220243556	132679			
0:00:35	1806586	51652	0:08:45	72973634	139004	0:16:55	152097675	149853	0:25:05	214859353	142766	0:27:45	220401143	132375			
0:00:40	2008337	50239	0:08:50	73781771	139217	0:17:00	152898684	149904	0:25:10	215177412	142504	0:27:50	220540906	132065			
0:00:45	2229769	49576	0:08:55	74588713	139424	0:17:05	153708200	149963	0:25:15	215530914	142267	0:27:55	220694699	131767			
0:00:50	2473494	49493	0:09:00	75394336	139625	0:17:10	154515315	150018	0:25:20	215736670	141934	0:28:00	220823814	131445			
0:00:55	2740120	49841	0:09:05	76199496	139822	0:17:15	155318214	150069	0:25:25	215966273	141619	0:28:05	220969713	131141			
0:01:00	2952336	49224	0:09:10	77005265	140015	0:17:20	156117670	150116	0:25:30	216250626	141342	0:28:10	221128686	130847			
0:01:05	3186574	49042	0:09:15	77783237	140156	0:17:25	156928962	150175	0:25:35	216537722	141069	0:28:15	221256282	130536			
0:01:10	3443797	49213	0:09:20	78587016	140340	0:17:30	157727070	150220	0:25:40	216743181	140745	0:28:20	221401403	130238			
0:01:15	3729219	49738	0:09:25	79381570	140504	0:17:35	158538286	150277	0:25:45	216895988	140388	0:28:25	221547187	129942			
0:01:20	4041953	50539	0:09:30	80185079	140681	0:17:40	159338451	150323	0:25:50	217041764	140029	0:28:30	221651532	129623			
0:01:25	4382967	51581	0:09:35	80992937	140863	0:17:45	160141968	150371	0:25:55	217180454	139668	0:28:35	221761933	129312			
0:01:30	4764351	52950	0:09:40	81796060	141033	0:17:50	160929596	150405	0:26:00	217296865	139295	0:28:40	221869982	128996			
0:01:35	5176329	54501	0:09:45	82598990	141200	0:17:55	161730947	150451	0:26:05	217427249	138933	0:28:45	221970025	128680			
0:01:40	5633336	56346	0:09:50	83404065	141368	0:18:00	162529231	150493	0:26:10	217566476	138579	0:28:50	222037236	128347			
0:01:45	6132369	58416	0:09:55	84203745	141524	0:18:05	163328600	150536	0:26:15	217725126	138240	0:28:55	222141609	128037			
0:01:50	6687061	60804	0:10:00	85004156	141679	0:18:10	164135466	150586	0:26:20	217854139	137884	0:29:00	222258878	127737			
0:01:55	7281947	63334	0:10:05	85810435	141841	0:18:15	164935574	150629	0:26:25	218046926	137571	0:29:05	222370557	127435			
0:02:00	7940029	66180	0:10:10	86607109	141987	0:18:20	165741721	150677	0:26:30	218191332	137229	0:29:10	222476182	127135			
0:02:05	8657775	69275	0:10:15	87409561	142135	0:18:25	166540600	150719	0:26:35	218342411	136894	0:29:15	222565843	126820			
0:02:10	9420611	72480	0:10:20	88214210	142286	0:18:30	167347631	150767	0:26:40	218470361	136546	0:29:20	222655858	126511			
0:02:15	10213593	75669	0:10:25	89027378	142449	0:18:35	168155298	150815	0:26:45	218636799	136224	0:29:25	222756944	126210			
0:02:20	11009158	78650	0:10:30	89839157	142607	0:18:40	168949949	150851	0:26:50	218784942	135893	0:29:30	222870602	125917			
0:02:25	11801470	81402	0:10:35	90648842	142760	0:18:45	169755286	150897	0:26:55	218919843	135556	0:29:35	222980520	125624			
0:02:30	12595939	83985	0:10:40	91460580	142912	0:18:50	170544894	150928	0:27:00	219051944	135219	0:29:40	223095145	125336			
0:02:35	13399372	86460	0:10:45	92261821	143047	0:18:55	171347333	150970	0:27:05	219181641	134883	0:29:45	223205439	125047			
0:02:40	14204721	88792	0:10:50	93068943	143188	0:19:00	172149041	151011	0:27:10	219341906	134568	0:29:50	223296567	124748			
0:02:45	15008505	90973	0:10:55	93881781	143336	0:19:05	172947063	151049	0:27:15	219501791	134254	0:29:55	223404584	124461			
0:02:50	15806658	92993	0:11:00	94694522	143482	0:19:10	173749851	151090	0:27:20	219642464	133930	0:30:00	223498375	124168			
0:02:55	16617202	94967	0:11:05	95507020	143624	0:19:15	174554741	151133	0:27:25	219751575	133597	0:30:05	223604485	123882			
0:03:00	17420342	96792	0:11:10	96323669	143771	0:19:20	175357263	151173	0:27:30	219905150	133278	0:30:10	223698680	123592			

0:03:05	18233290	98571	0:11:15	97142793	143920	0:19:25	176162019	151215	0:27:35	220082032	132982	0:30:15	223793296	123304
0:03:10	19041836	100232	0:11:20	97946057	144043	0:19:30	176964291	151254	0:27:40	220243556	132679	0:30:20	223876742	123011
0:03:15	19858904	101853	0:11:25	98756210	144175	0:19:35	177758695	151287	0:27:45	220401143	132375	0:30:25	223964092	122722
0:03:20	20662558	103325	0:11:30	99564756	144300	0:19:40	178556577	151322	0:27:50	220540906	132065	0:30:30	224044696	122435
0:03:25	21474039	104763	0:11:35	100376559	144432	0:19:45	179366898	151367	0:27:55	220694699	131767	0:30:35	224140696	122149
0:03:30	22271203	106065	0:11:40	101184918	144555	0:19:50	180164714	151402	0:28:00	220823814	131445	0:30:40	224257620	121881
0:03:35	23077895	107351	0:11:45	101993663	144677	0:19:55	180954442	151429	0:28:05	220969713	131141	0:30:45	224392163	121623
0:03:40	23887683	108592	0:11:50	102799958	144793	0:20:00	181750611	151462	0:28:10	221128686	130847	0:30:50	224511448	121359
0:03:45	24695489	109769	0:11:55	103613317	144918	0:20:05	182466031	151429	0:28:15	221256282	130536	0:30:55	224616723	121089
0:03:50	25504799	110902	0:12:00	104414748	145025	0:20:10	183183260	151394	0:28:20	221401403	130238	0:31:00	224722757	120820
0:03:55	26314690	111988	0:12:05	105221893	145138	0:20:15	183989781	151435	0:28:25	221547187	129942	0:31:05	224829677	120554
0:04:00	27128030	113044	0:12:10	106024013	145243	0:20:20	184778146	151460	0:28:30	221651532	129623	0:31:10	224930516	120285
0:04:05	27925227	113991	0:12:15	106838141	145363	0:20:25	185573905	151492	0:28:35	221761933	129312	0:31:15	225026805	120016
0:04:10	28715006	114871	0:12:20	107651213	145479	0:20:30	186374678	151527	0:28:40	221869982	128996	0:31:20	225121052	119747
0:04:15	29523522	115789	0:12:25	108458462	145586	0:20:35	187170156	151558	0:28:45	221970025	128680	0:31:25	225249100	119497
0:04:20	30321891	116633	0:12:30	109263060	145689	0:20:40	187971640	151593	0:28:50	222037236	128347	0:31:30	225391795	119261
0:04:25	31131139	117486	0:12:35	110074510	145799	0:20:45	188771459	151626	0:28:55	222141609	128037	0:31:35	225545981	119023
0:04:30	31940856	118310	0:12:40	110877636	145896	0:20:50	189568027	151657	0:29:00	222258878	127737	0:31:40	225685160	118783
0:04:35	32750280	119102	0:12:45	111692537	146008	0:20:55	190371214	151693	0:29:05	222370557	127435	0:31:45	225816049	118540
0:04:40	33557592	119858	0:12:50	112500894	146109	0:21:00	191174295	151728	0:29:10	222476182	127135	0:31:50	225918500	118283
0:04:45	34363563	120583	0:12:55	113306210	146206	0:21:05	191960460	151750	0:29:15	222565843	126820	0:31:55	226026370	118031
0:04:50	35168411	121280	0:13:00	114104299	146292	0:21:10	192747045	151772	0:29:20	222655858	126511	0:32:00	226129998	117797
0:04:55	35975036	121958	0:13:05	114911364	146387	0:21:15	193542111	151801	0:29:25	222756944	126210	0:32:05	226224542	117521
0:05:00	36766083	122563	0:13:10	115723505	146490	0:21:20	194332989	151825	0:29:30	222870602	125917	0:32:10	226317297	117264
0:05:05	37573371	123200	0:13:15	116536338	146591	0:21:25	195131863	151856	0:29:35	222980520	125624	0:32:15	226460721	117035
0:05:10	38380721	123818	0:13:20	117343198	146683	0:21:30	195932233	151888	0:29:40	223095145	125336	0:32:18	226553773	116914
0:05:15	39178679	124386	0:13:25	118147916	146772	0:21:35	196731864	151919	0:29:45	223205439	125047	0	0	0
0:05:20	39986042	124965	0:13:30	118963172	146872	0:21:40	197404020	151852	0:29:50	223296567	124748	0	0	0
0:05:25	40793962	125529	0:13:35	119757965	146946	0:21:45	198194766	151876	0:29:55	223404584	124461	0	0	0
0:05:30	41594759	126054	0:13:40	120576305	147048	0:21:50	198993251	151906	0:30:00	223498375	124168	0	0	0
0:05:35	42405828	126593	0:13:45	121389216	147143	0:21:55	199765214	151915	0:24:35	212890561	144335	0	0	0
0:05:40	43213499	127107	0:13:50	122209394	147244	0:22:00	200527363	151917	0:24:40	213248360	144089	0	0	0
0:05:45	44018460	127598	0:13:55	123019324	147333	0:22:05	201293520	151922	0:24:45	213587146	143832	0	0	0
0:05:50	44827832	128088	0:14:00	123831155	147422	0:22:10	202045789	151917	0:24:50	213924191	143575	0	0	0
0:05:55	45637485	128565	0:14:05	124637415	147504	0:22:15	202833378	151938	0:24:55	214340279	143374	0	0	0
0:06:00	46435678	128996	0:14:10	125450131	147592	0:22:20	203599646	151943	0:25:00	214599630	143069	0	0	0
0:06:05	47243373	129442	0:14:15	126256649	147672	0:22:25	204331925	151922	0:25:05	214859353	142766	0	0	0

0:06:10	48033671	129829	0:14:20	127065501	147754	0:22:30	205066124	151903	0:25:10	215177412	142504	0	0
0:06:15	48844449	130260	0:14:25	127874088	147834	0:22:35	205794549	151894	0:25:15	215530914	142267	0	0
0:06:20	49651926	130671	0:14:30	128685577	147918	0:22:40	206084602	151535	0:25:20	215736670	141934	0	0
0:06:25	50460980	131075	0:14:35	129495449	147999	0:22:45	206397504	151210	0:25:25	215966273	141619	0	0
0:06:30	51259118	131441	0:14:40	130296447	148068	0:22:50	206641178	150835	0:25:30	216250626	141342	0	0
0:06:35	52069938	131830	0:14:45	131106247	148147	0:22:55	206967910	150525	0:25:35	216537722	141069	0	0
0:06:40	52873866	132192	0:14:50	131907128	148214	0:23:00	207244015	150179	0:25:40	216743181	140745	0	0
0:06:45	53681130	132554	0:14:55	132717745	148292	0:23:05	207436076	149776	0:25:45	216895988	140388	0	0
0:06:50	54489038	132908	0:15:00	133524615	148364	0:23:10	207668173	149404	0:25:50	217041764	140029	0	0
0:06:55	55299663	133260	0:15:05	134328609	148433	0:23:15	207913393	149044	0:25:55	217180454	139668	0	0
0:07:00	56101079	133581	0:15:10	135135150	148503	0:23:20	208197445	148715	0:26:00	217296865	139295	0	0
0:07:05	56898987	133887	0:15:15	135946570	148579	0:23:25	208523255	148418	0:26:05	217427249	138933	0	0
0:07:10	57705680	134206	0:15:20	136759126	148655	0:23:30	208784201	148076	0:26:10	217566476	138579	0	0
0:07:15	58513629	134521	0:15:25	137552956	148710	0:23:35	209021041	147720	0:26:15	217725126	138240	0	0
0:07:20	59304544	134790	0:15:30	138369067	148788	0:23:40	209326664	147416	0:26:20	217854139	137884	0	0
0:07:25	60108708	135083	0:15:35	139183231	148863	0:23:45	209656954	147130	0:26:25	218046926	137571	0	0
0:07:30	60913042	135369	0:15:40	139983653	148922	0:23:50	210005025	146861	0:26:30	218191332	137229	0	0
0:07:35	61718502	135652	0:15:45	140793895	148992	0:23:55	210308881	146559	0:26:35	218342411	136894	0	0
0:07:40	62524705	135930	0:15:50	141610090	149067	0:24:00	210708556	146328	0:26:40	218470361	136546	0	0
0:07:45	63333368	136208	0:15:55	142412944	149127	0:24:05	211073298	146074	0:26:45	218636799	136224	0	0
0:07:50	64135420	136465	0:16:00	143221645	149193	0:24:10	211416795	145807	0:26:50	218784942	135893	0	0
0:07:55	64934707	136711	0:16:05	144031536	149259	0:24:15	211689050	145493	0:26:55	218919843	135556	0	0
0:08:00	65740887	136967	0:16:10	144841060	149324	0:24:20	212020386	145222	0:27:00	219051944	135219	0	0
0:08:05	66542392	137207	0:16:15	145644031	149382	0:24:25	212290614	144911	0:27:05	219181641	134883	0	0
0:08:10	67342663	137441	0:16:20	146453631	149445	0:24:30	212525769	144578	0:27:10	219341906	134568	0	0