Universidade do Minho

Escola de Engenharia

Departamento de Informática

Manutenção de Caches para Sistemas OLAP

Pedro Carvalho Marques

Dissertação de Mestrado

2010

Manutenção de Caches para Sistemas OLAP

Pedro Carvalho Marques

Dissertação apresentada à Universidade do Minho para obtenção do grau de Mestre em Informática, elaborada sob orientação do Professor Doutor Orlando Manuel de Oliveira Belo.

2010

Aos meus pais, irmão e todos os amigos que tanto me ajudaram neste processo

Agradecimentos

Os agradecimentos que endereço nesta fase são conscientemente poucos, tanto que foi o apoio que senti ao longo do processo de elaboração desta tese, mas todos os outros não foram esquecidos, ficam "cá dentro".

Antes de mais, aos meus pais e irmão que tanto me apoiaram, com tantas palavras sensatas e tantas outras críticas - construtivas sempre, não fossem eles quem mais queria o meu sucesso. De seguida, aos amigos, e às amigas, de tantas horas de conversa, de dias animados, e outros menos… porque é isso que nos faz aprender… evoluir! Ainda, e como não poderia deixar de ser, a toda a restante família, que sempre se mostrou confiante e segura que este trabalho, este percurso, chegaria a bom porto.

Fica também um agradecimento especial ao meu orientador e, acima de tudo, incentivador nos momentos certos e ao longo de todo um ano em que sempre esteve presente quando foi necessário.

Por último quero com grande sentimento agradecer a quem, apesar de não estar presente, marcou a minha vida, a minha maneira de ser e me mostrou como era o mundo, incentivando-me a continuar… mesmo quando o vento não soprava…

Resumo

Manutenção de Caches para Sistemas OLAP

Nos dias que correm a utilização de sistemas multidimensionais de dados faz parte do quotidiano de qualquer organização de média ou de grande dimensão. Este tipo de sistema, cuja principal finalidade consiste em auxiliar os seus utilizadores nas suas actividades de tomada de decisão, tem por principais características a flexibilidade na exploração de dados e a celeridade na disponibilização de informação. Apesar de todos os mecanismos já existentes, que contribuem para este objectivo, é, por vezes, muito difícil manter os níveis de desempenho desejados pelos utilizadores. Como consequência disto, foram estudadas outras formas para reduzir a carga imposta ao servidor multidimensional de dados – o servidor OLAP. Um destes mecanismos é a criação de *caches* que armazenam informação já consultada e que, aquando de um pedido, o satisfazem sem terem a necessidade de consultar a sua fonte. Devido à natureza dos utilizadores dos sistemas OLAP, é possível determinar com bastante precisão os seus padrões de acesso e de exploração, isto é, quais as consultas efectuadas por um determinado utilizador ao longo de um dado período de tempo. Levando esta análise para um pouco mais à frente é, ainda, possível prever com antecedência qual a sequência exacta de consultas que será efectuada por um determinado utilizador quando este iniciar uma qualquer sessão de exploração de dados. Após esta fase de previsão, consegue-se decidir quais as *queries* que deverão ser pré-materializadas, armazenando-as numa *cache*, de forma a servir o maior número possível de pedidos do utilizador a partir desta. A técnica proposta nesta dissertação centra-se na problemática que gira em torno da simplificação do número *queries* que deverão ser pré-materializadas. O objectivo final desta técnica consiste em manter um balanço positivo entre o tempo dispendido a realizar esta pré-materialização e o que seria gasto caso tudo fosse calculado apenas quando requisitado.

Abstract

Maintaining OLAP Cache Systems

Nowadays, the use of Multidimensional Data Systems has become a part of everyday actions in medium and large companies. This type of system, which concerns mainly in aiding it's users in the process of decision making, has a very large flexibility in data exploration and speed of response to queries. Despite all the existing techniques, it is sometimes, very hard to maintain such high levels of performance users demand. With the purpose of tackling these performance losses, other techniques where developed, wich try to reduce central data servers load. One of such mechanisms is the creation of OLAP caches that maintain previous queries and serve them upon subsequent requests without having to ask the central server. Due to OLAP Systems organization, it is possible to identify the characteristics of its users and its exploration patterns – what queries will a user submit during a session. It is, however, possible to go one step further, and to predict exactly wich data will be requested by a specific user and, more important, the sequence of those requests. This is called the prediction phase and is followed by the pre-materialization of views that correspond to the user's requests in the future. These views are then stored in the cache and served to the user in the appropriate time. The proposed technique's main goal consists in maintaining a positive ration between the time spent to predict and materialize the views, and the time that would be spent if no prediction had been done.

Índice

Índice de Figuras

Índice de Tabelas

Índice de Algoritmos

Capítulo 1

Introdução

1.1 Sistemas de Processamento Analítico

Com o rápido aumento da utilização dos Sistemas de Suporte à Decisão por parte das organizações, a tecnologia *On-Line Analytical Processing* (OLAP) tem sido, de algum tempo a esta parte, alvo de particular atenção das equipas de investigação e desenvolvimento bem como dos criadores de software de grande porte em todo o mundo. Tendo como principal aliciante para as organizações, estas tecnologias pretendem fornecer uma vantagem competitiva sobre os demais a quem delas se socorra para realizar análises dos seus índices de desempenho. Incluído num Sistema de Suporte à Decisão encontra-se, assim, em posição de destaque, um sistema OLAP, que permite uma análise segundo as mais variadas perspectivas dos dados da organização de forma a que as decisões sejam tomadas com base em estatísticas e não, como tantas vezes acontece, com base na intuição dos decisores. Estas análises permitem que os dados sejam relacionados de forma não trivial e que a perspectiva de análise seja alterada sempre que necessário devido à forma como os dados se encontram armazenados, isto é, a análise é realizada, essencialmente, através da manipulação de estruturas de dados multidimensionais. Estas estruturas, também conhecidas como hiper-cubos, possuem mecanismos de agregação dos dados que, apesar de assentarem frequentemente sobre sistemas de bases de dados relacionais, estão optimizados para realizarem cruzamentos de informação (*joins*) de forma eficiente. Um dos mecanismos que permite esta

eficiência na manipulação dos dados consiste na sua pré-materialização em vistas materializadas (tal como as existentes nas bases de dados relacionais operacionais) que, num caso ideal, poderão permitir a resposta imediata de qualquer questão que seja posta ao sistema. Como este caso ideal é praticamente impossível de atingir, nomeadamente devido às limitações de memória dos sistemas, foram desenvolvidas técnicas de optimização das vistas a serem materializadas que ainda hoje são alvo de estudo e investigação.

Outra das formas encontradas para optimizar o desempenho deste tipo de sistemas foi a introdução de mecanismos de *caching*. As técnicas de *caching*, desde há muito tempo aplicadas à WWW e com resultados muito positivos, foram vistas como uma possibilidade de acelerar o processo de resposta às *queries* efectuadas pelos utilizadores de OLAP. Quando aplicadas à *World Wide Web* (WWW), as técnicas de *caching* apenas pretendiam manter informação estática para que esta, quando requisitada repetidamente por um utilizador, fosse fornecida pela *cache* e não pela fonte primária de dados, obtendo-se assim duas vantagens em simultâneo: uma diminuição do tempo de resposta ao utilizador e uma diminuição do número de acessos ao servidor de onde a informação é proveniente. Esta diminuição do número de acessos ao servidor resulta numa maior disponibilidade deste para servir pedidos que não estejam guardados em nenhuma *cache* e, consequentemente, num aumento do desempenho global do sistema. No caso dos sistemas OLAP, a informação a manter em *cache* é de natureza dinâmica, ainda que não seja actualizada de forma muito frequente (possivelmente apenas uma vez em cada ciclo de refrescamento do cubo) e, num caso óptimo, apenas de forma incremental, o tipo de desafio gerado por estas condicionantes revelou ser de resolução algo complexa, embora se encontre, já neste momento, num patamar que lhe permite ser um dos factores fundamentais e determinantes no bom desempenho de qualquer sistema OLAP.

Com o aumento da maturidade deste tipo de sistemas, rapidamente se complicaram os mecanismos de *caching* a si associados. Os primeiros sistemas, tal como em WWW, centravam-se na óptica do utilizador individual que era o único a beneficiar da sua própria *cache* (arquitectura de *client-side caching*). A evolução deu-se após a constatação do facto de este tipo de serviços serem por regra utilizados dentro de uma determinada organização pelo que não existe necessidade de competição dos seus utilizadores por recursos. Neste sentido, os sistemas de *caching* desenvolvidos posteriormente seguiam uma vertente de partilha de *caches* entre os utilizadores,

quer esta se encontrasse do lado do servidor quer do lado dos clientes. Esta partilha pode ser efectuada de várias formas, desde sistemas *peer-to-peer* [Kalnis et al. 2002] que partilham as *caches* dos utilizadores, a complexos sistemas de *Olap Cache Servers* [Kalnis & Papadias 2001] que mantém comunicação entre si bem como com os utilizadores em geral como forma de manter a partilha de informação constante.

Uma outra abordagem a este problema, surgiu após consideração da natureza, tanto dos dados mantidos neste tipo de sistemas, como dos seus utilizadores. No que aos dados diz respeito, estes são mantidos de uma forma organizada, segundo hierarquias e recorrendo a mecanismos que facilitam a sua consulta. Os utilizadores deste tipo de sistemas, por seu lado, são ainda hoje, geralmente, altos cargos das organizações e com interesses muito particulares a nível dos dados que consultam. Por este motivo é possível, com alguma exactidão, determinar quais as consultas que serão realizadas por um determinado utilizador aquando do seu próximo início de sessão. Com o objectivo principal de diminuir a carga de acessos ao servidor, e após definidos estes padrões de acesso, procede-se à pré-materialização das vistas correspondentes às respostas às *queries* que serão realizadas pelo utilizador. Estas vistas, após materializadas, serão adicionadas à *cache* e respondidas directamente a partir desta, sempre que requisitadas. Um problema que se põe com este tipo de abordagem é o facto de não ser comportável a materialização de um número demasiado elevado de vistas. De forma a minorar este problema, é possível restringir o número de vistas a serem materializadas através de simples "filtros de qualidade" pelas quais estas passarão, seleccionando apenas as melhores, ou mais benéficas. A técnica proposta e, consequentemente, os testes realizados visam determinar qual o desempenho de um sistema de *caching* baseado em padrões de acesso dos utilizadores, após simplificação do número de vistas a ser materializado.

1.2 Motivações para o Estudo

De forma geral, o objectivo principal de um sistema de *caching*, qualquer que seja a tecnologia à qual este se encontra aplicado, é o de reduzir o tempo total de acesso aos dados por parte dos seus utilizadores. Estes sistemas consistem em manter a informação que obedeça a um determinado critério escolhido, em locais cujo acesso se revele mais célere do que o do servidor central, retirando assim carga deste servidor resultando numa maior disponibilidade da sua parte. Tomando como exemplo um *browser* Web convencional, quando uma página é acedida, a

informação é frequentemente mantida em *cache* na máquina onde se encontra o *browser* para que, num posterior acesso à mesma informação, esta possa ser disponibilizada sem necessidade de acesso à fonte primária de informação. Este tipo de sistema levanta, como seria de esperar, algumas questões, como por exemplo, qual a informação a ser mantida em *cache*, ao fim de quanto tempo esta informação deverá ser renovada por já não se revelar correcta ou actual, etc. Todas estas questões e muitas outras são, ainda hoje, alvo de estudo devido à sua importância fulcral no bom funcionamento de qualquer sistema de *caching*.

Relativamente aos sistemas OLAP, os mecanismos de *caching* revelam ser de particular importância devido à grande quantidade de informação que disponibilizam, mas também de uma grande dificuldade, não só de implementação mas sobretudo de optimização devido à natureza dinâmica dos dados em questão. Contrariamente aos mecanismos de *caching* de um servidor *Web* convencional, que apenas mantêm informação estática, nos sistemas OLAP na informação mantida pode ser utilizada para derivar outra informação através de agregações o que aumenta bastante o nível de complexidade da resolução de todos estes problemas. Neste contexto surgem algumas dúvidas relacionadas com a melhor forma de compor a informação existente, derivando assim a informação pretendida. Este tipo de operação poderá, por exemplo, envolver a pesquisa de informação através de vários elementos de uma rede (*caching* distribuído) o que aumenta de forma significativa a complexidade dos algoritmos. Sendo esta apenas uma questão muito específica de toda a problemática de estudo desta matéria é, no entanto, representativa da dificuldade de implementação deste tipo de mecanismos. Estas questões, entre outras, encontram-se no cerne da motivação do estudo deste tipo de mecanismos sendo o objectivo último deste trabalho clarificar algumas destas situações.

1.3 *Caching* em sistemas OLAP

Num sistema OLAP, as vantagens da introdução de mecanismos de *caching* prendem-se sobretudo com dois factores fundamentais. Em primeiro lugar, pelo facto de as *queries* passarem a ser respondidas directamente a partir da *cache*, o que faz com que, como sabemos, o número de acessos ao servidor central (servidor OLAP) diminua. Como consequência, a disponibilidade deste para responder a *queries* que lhe sejam dirigidas aumenta, bem como o seu desempenho. Por outro lado, observa-se um outro grande benefício da aplicação destas técnicas devido ao facto de

as *caches* estarem frequentemente mais próximas dos utilizadores (em termos de latência de rede). Assim sendo, o desempenho global do sistema aumenta, numa percentagem bastante significativa, através da redução do tráfego de rede nos nodos mais próximos do servidor central. Por outro lado, a aplicação deste tipo de mecanismos envolve um grande esforço, quer por parte de quem o implementa, mas também, e sobretudo, por parte de quem estuda todas as características do sistema para que a implementação seja a mais optimizada possível para este.

De acordo com cada sistema OLAP e as suas características, a implementação do sistema de *cache* correspondente poderá variar desde os algoritmos a serem utilizados ao *hardware* necessário ao seu suporte. Por este motivo, é necessário que a análise prévia do sistema seja o mais exaustiva possível, para que toda a implementação do sistema de *cache* seja a mais adequada e não sejam necessárias grandes reestruturações na sua arquitectura num futuro próximo.

1.4 Organização do documento

Este documento encontra-se organizado segundo um esquema de diferenciação de conteúdos por capítulos. Em adição a este primeiro capítulo, podemos encontrar outros cinco, nomeadamente:

- Capítulo 2: Metodologias e técnicas de *caching* – neste capítulo são descritas algumas metodologias utilizadas para gestão dos espaços e memória reservados a manter dados em *cache*, bem como as suas vantagens e desvantagens.
- Capítulo 3: Arquitectura de Sistemas de *Caching* – são, neste ponto, descritos alguns sistemas de *caching* que vão para além dos típicos sistemas de *client-side caching* e envolvem, na grande maioria dos casos, um grande número de outras máquinas em adição às dos clientes que realizam consultas sobre os dados.
- Capítulo 4: A Técnica Proposta – propõe-se, aqui, uma técnica de *caching* baseada em previsão de padrões de acesso aos dados. São explicadas neste capítulo as diversas motivações, vantagens, desvantagens e considerações necessárias a implementação de um sistema com estas características.
- Capítulo 5: Os Casos de estudo – os dois casos de estudo apresentados no Capítulo 5 pretendem retirar conclusões quanto ao desempenho da técnica proposta no capítulo anterior através de uma série de testes e correspondente apresentação dos resultados.

- Capítulo 6: Conclusões e Trabalho Futuro – neste capítulo são retiradas conclusões sobre a viabilidade da técnica proposta no Capítulo 4 e é indicado um rumo para o trabalho a desenvolver futuramente.

Capítulo 2

Metodologias e Técnicas de *Caching*

2.1 Métricas de medição de desempenho

Quando se pretende manter informação em *cache*, os dados são armazenados num espaço de memória especificamente reservado para este efeito. Este espaço é denominado por *view pool*, ou *pool* de vistas. Quando existe informação a ser adicionada a esta *pool*, caso exista espaço disponível, ela é imediatamente adicionada, caso contrário, existem decisões a serem tomadas. Se, por um lado, uma das alternativas passa, simplesmente, por não adicionar a nova informação, por outro, é possível remover informação já existente de forma a criar espaço suficiente na *pool* para adicionar os novos dados. Este tipo de decisões é da responsabilidade do algoritmo de gestão de *caches* implementado em cada situação e é, em grande parte, decisivo no desempenho global de todo o sistema.

O sistema de gestão de *caches* é um factor decisivo no desempenho global de um sistema de *caching*. A principal função deste tipo de sistemas prende-se com a capacidade de decidir qual a informação a ser mantida ou removida da *pool* de vistas por forma a permitir a adição de novos dados. Desta forma, e com o objectivo de medir o desempenho de um sistema deste tipo, existem duas métricas fundamentais, o *Hit Ratio* e o *Byte Hit Ratio.*

2.1.1 Hit Ratio

Esta métrica de medição do desempenho de um sistema de gestão de *cache* pretende saber, de todos os pedidos efectuados pelo(s) utilizador(es), quais os que foram respondidos directamente a partir da *cache*. Desta forma, é possível definir o *Hit Ratio* de um determinado algoritmo como a percentagem de pedidos que foram satisfeitos a partir da cache, ou matematicamente:

$$Hit\ Ratio = \frac{\#pedidos\ satisfeitos\ da\ cache}{\#total\ de\ pedidos}$$

2.1.2 Byte Hit Ratio

Analogamente à *Hit Ratio*, esta é uma outra forma de medir o desempenho de um sistema de gestão de *caches*. Contrariamente à métrica anterior, esta pretende ter em conta, não só quantos pedidos foram satisfeitos a partir da *cache*, mas também qual a quantidade de informação que foi servida desta forma. Se, numa *cache*, forem mantidos muitos dados de pequena dimensão, é natural que a percentagem de pedidos directamente respondidos a partir desta, seja grande embora seja possível que o número de *bytes* da informação satisfeita desta forma possa não o ser. De forma inversa, é possível que pouca informação mantida mas em grande quantidade resulte no cenário inverso. Os algoritmos de gestão de *caches* apresentados de seguida pretendem encontrar uma solução de compromisso entre estas duas situações sendo que a situação ideal se encontrará, dependendo dos casos, num ponto intermédio entre estas duas. De forma semelhante à métrica anterior, esta define-se segundo a seguinte fórmula:

$$Byte\ Hit\ Ratio = \frac{\#Bytes\ satisfeitos\ da\ cache}{\#Bytes\ total}$$

2.2 Algoritmos de gestão de caches

Quando um pedido é endereçado a uma *cache*, podem verificar-se duas situações distintas:

1. A informação pretendida encontra-se presente na *cache* – *cache hit* – pelo que será devolvida a quem tiver formulado o pedido.
2. A informação pretendida não se encontra na *cache* – *cache miss* – e, portanto não poderá ser devolvida a quem a tenha requisitado.

Caso se verifique a primeira situação, o processo decorrerá de forma simples, já que apenas será necessário devolver a informação encontrada a quem a tenha requisitado e, eventualmente, em casos mais complexos, actualizar eventuais parâmetros associados a essa informação (*timestamp* do último acesso, frequência de acessos, etc.). Na segunda situação, em que a informação não se encontra mantida em *cache*, o processo de procura de informação poderá desenrolar-se de várias formas. Imagine-se a o cenário em que um utilizador acede à sua *cache* local como forma preferencial de obtenção de informação e, caso isto não seja suficiente, acederá directamente à fonte de dados. Ao receber a informação pretendida, existe a necessidade de decidir se a informação recebida deverá ser adicionada à *cache* local, ou apenas utilizada. Este tipo de decisões afecta directamente o conteúdo de uma *cache* e, consequentemente, o seu desempenho. Desta forma, é possível através das consequências destas decisões avaliar positiva ou negativamente um algoritmo de gestão de *cache*, recorrendo às métricas anteriormente apresentadas. Nas próximas secções serão apresentados alguns algoritmos que realizam esta gestão e se realçará algumas das suas vantagens e desvantagens.

2.2.1 First In First Out (FIFO)

Sendo um dos mais simples algoritmos de gestão de *caches*, esta solução retira, caso não exista espaço disponível para manter uma determinada informação em *cache*, a informação inserida há mais tempo. A implementação deste algoritmo assenta numa estrutura do tipo *queue.* Em que os elementos são sempre inseridos no início e retirados do fim da estrutura, obtendo-se assim o comportamento esperado. Este tipo de estrutura resulta bem num tempo constante de acesso à *cache*, independentemente do seu tamanho. Como principal vantagem deste algoritmo destaca-se a sua simplicidade de implementação. Como desvantagem poder-se-á notar o facto de este não revelar um esforço por manter em *cache* a informação mais benéfica para o utilizador, o que, como se sabe, deteriora o seu desempenho.

2.2.2 Second Chance

Surgindo como uma evolução do algoritmo FIFO, esta solução opta por adicionar um *bit* de controlo à informação mantida em *cache*. Este *bit* tem o objectivo de fornecer uma "segunda oportunidade" à informação antes de esta ser removida da *cache*. Quando uma nova informação é adicionada, o seu *bit* de controlo é marcado como activo. Caso esta atinja o ponto em que se torna

a primeira escolha para remoção, o algoritmo verifica se o *bit* se encontra activo ou não. Caso o *bit* se encontre activo, em lugar de esta informação ser removida directamente da cache, o *bit* é desactivado e esta é reinserida no início da fila (tal como se de uma nova entrada se tratasse), dando, assim, uma segunda hipótese à informação antes de a remover. Se, por outro lado, este *bit* se encontrar desactivado a informação é imediatamente removida da *cache*. Note-se que, caso a informação seja referenciada posteriormente à desactivação do *bit* de controlo, este será reactivado e a informação reinserida na posição cimeira da lista. Assim à custa de uma ligeira alteração a nível da implementação do algoritmo, é conseguida uma solução que origina melhores desempenhos do que o algoritmo anterior. No que diz respeito à complexidade de implementação, esta mantém-se ainda muito simples. Tal como no algoritmo FIFO, é de ressalvar o facto de este algoritmo não ter em conta as necessidades específicas do utilizador.

2.2.3 CLOCK

Também baseado na alternativa FIFO, e com uma forma de funcionamento em tudo semelhante a *Second-chance*, este algoritmo assenta a sua estrutura numa lista circular. Como dados adicionais à informação armazenada, mantém-se um apontador para a posição ocupada pela primeira opção de remoção, bem como um *bit* de controlo (tal como na solução anterior). Caso seja necessário adicionar uma nova informação, consultar-se-á o *bit* de controlo referente aos dados armazenados na posição apontada como primeira opção de remoção. Caso este *bit* se encontre desactivado, a informação será removida e a nova informação adicionada. Caso contrário, será desactivado, a posição do apontador incrementada e o processo repetido até que se dê a substituição de alguma informação pela nova. A nível de desempenho, este algoritmo revela ser mais célere do que o FIFO ou até mesmo que o "*Second Chance*".

2.2.4 Least Recently Used

De simples implementação, o algoritmo *Least Recently Use* (LRU) [Mookerjee & Tan 2002] consiste em manter a informação numa lista, na qual todas as informações são adicionadas à "cabeça" desta. Quando um elemento da lista é acedido, este retorna à posição inicial em que foi inserido na lista (primeira posição). Sendo necessário remover elementos desta lista, estes serão retirados do fim da mesma, o que faz com que o elemento removido seja o elemento presente na lista que foi

acedido há mais tempo. A simplicidade de implementação deste algoritmo, bem como a sua velocidade de acesso e a sua adaptabilidade a novos padrões de acesso são claramente as suas principais vantagens. Por outro lado, este algoritmo não possui a capacidade de identificação do benefício da informação a manter em *cache* pelo que toda e qualquer informação será inserida nesta, podendo assim reduzir o benefício da *cache*.

2.2.5 Least Frequently Used

A ideia do algoritmo *Least Frequently Used* (LFU) é a de manter informação adicional associada aos dados mantidos em *cache*. Esta informação passa por saber a frequência de acesso de cada um dos elementos da *cache* com o objectivo de remover desta apenas aquele(s) que revele(m) ter sido menos requisitado(s) recentemente. Esta versão de LFU, também conhecida por "LFU em cache" é uma das duas formas que temos para a sua implementação. Outra forma de interpretar este algoritmo prevê a manutenção de informação relativa à frequência de acesso de todos os dados existentes, e não só os presentes em *cache*, sendo conhecida por "LFU perfeito". Desta forma, é possível uma maior adaptação a padrões de acesso a longo prazo, sendo por sua vez a primeira versão do algoritmo mais indicada para capturar padrões de acesso a curto prazo. Em regra geral, verificam-se desempenhos superiores por parte da versão "LFU perfeito", embora o seu *overhead* de acesso à informação seja superior. Isto deve-se ao facto de, a cada acesso, ser necessário actualizar a informação relativa à frequência de pedidos dessa mesma informação. Como principal vantagem de ambas as versões do algoritmo destaca-se o facto de possuir a capacidade de capturar os referidos padrões de acesso aos dados. Por outro lado, destaca-se como factor negativo a fraca escalabilidade deste algoritmo devido, sobretudo, ao *overhead* de manter a informação relativa aos acessos aos dados actualizada.

2.2.6 Simple time-based expiration

Este algoritmo, remove dados da sua *cache* com base apenas num período de tempo estipulado para a sua validade. Este facto resulta numa redução do *overhead* de manutenção de informação auxiliar associada aos dados, mas, também, numa redução da assertividade relativamente aos dados a serem removidos da *cache*. Quando adicionados à *cache*, é associado aos dados um período de validade definido segundo um qualquer critério adoptado e, passado esse período,

estes são removidos da *cache*. A principal vantagem deste algoritmo está relacionada com a sua facilidade de implementação e manutenção (inexistente) da informação auxiliar mantida.

2.2.7 Sliding time-based expiration

Esta abordagem, uma evolução do algoritmo anterior, prevê a remoção dos dados da *cache* no final de um período de tempo medido em função do último acesso registado a essa informação. Se, por um lado, este comportamento se pode assemelhar ao do algoritmo LRU (pois o período de validade é actualizado em caso de acessos recorrentes ao ficheiro), por outro revela uma tentativa de gestão de *cache* algo simplista. Um ponto vantajoso deste algoritmo é a sua fácil implementação e a sua boa escalabilidade, uma vez que a informação auxiliar a manter é pouca e (maioritariamente) estática.

2.2.8 Lowest Benefit First

Este algoritmo [Kalnis et. al. 2002], baseia-se na atribuição de uma métrica de custo, *W*, inicialmente de valor igual ao seu benefício, a cada *chunk* mantido em *cache*. Este valor é decrementado cada vez que um *chunk* é avaliado para admissão à *cache* e o seu valor inicial restaurado sempre que este *chunk* seja acedido novamente. A forma de remoção de *chunks* mantidos em *cache*, para libertar espaço para nova informação, passa por ordenar a lista de *chunks* existentes, por ordem ascendente de W, e remover os primeiros cujo somatório do espaço permita adicionar o novo *chunk* à cache. Como optimização do algoritmo, para evitar a necessidade de reordenar a lista de *chunks* existentes na *cache*, cada vez que seja necessário libertar espaço, poderá ser utilizado o mecanismo *CLOCK* [Deshpande et al. 1998].

2.2.9 Greedy Dual

Este algoritmo [Young 1991] trata o caso em que as páginas em *cache* têm o mesmo tamanho, mas diferentes custos de *fetching* de outra localização. Para isso, o algoritmo associa um valor H a cada página em *cache* (p). Inicialmente, quando uma página é trazida para a *cache*, o valor de H é inicializado com o valor do custo de trazer esta página de uma outra localização. Quando é necessário efectuar uma substituição, a página com o menor valor de H (H_{min}) é removida e, de seguida, todas as páginas que permaneçam em memória vêem o valor de H (a si associado)

reduzido em H_{min}. Quando uma página é acedida, o seu valor de H é reposto no valo inicial (custo de trazer a página para a *cache*). Deste modo, as primeiras páginas a serem removidas da *cache* são as que possuem baixo custo de "transporte" para a *cache* ou, em alternativa, as que, mesmo possuindo um valor deste custo bastante grande, não são acedidas durante um longo período de tempo. A fórmula utilizada para cálculo do valor de H inicial é:

$$H = cost(p)$$

2.2.10 Greedy-Dual-Size

Apresentado por Pey Cao e Sandy Irani [Cao & Irani 1997] o algoritmo *Greedy-Dual-Size* tem como principal objectivo aumentar o *hit ratio* de uma *proxy*. Como tal, a melhor forma de o fazer é implementar um mecanismo que remova da *cache* ficheiros grandes (especialmente os que não são acedidos ao longo de um grande período de tempo) em detrimento de ficheiros pequenos. A noção intuitiva é que pelo facto de possuirmos em *cache* muitos ficheiros (ainda que pequenos) poderemos satisfazer mais pedidos do que se possuirmos menos ficheiros de tamanho maior. O mecanismo utilizado é a atribuição de um valor inversamente proporcional ao tamanho do ficheiro como custo inicial, o que favorece os documentos mais pequenos e remove mais facilmente os maiores. Chama-se a esta variante do algoritmo *Greedy-Dual-Size(1)* ou *GD-Size(1)*. A fórmula para calcular o valor de H inicial de uma página é:

$$H = \frac{1}{size(p)}$$

Uma grande desvantagem deste algoritmo é, como consequência do alto *hit ratio*, uma notória diminuição do valor de *byte hit ratio*. Uma sua alternativa, que mantém um maior equilíbrio entre os valores de *hit ratio* e de *byte hit ratio*, tem o nome de *Greedy-Dual-Size(packets)* ou GD-*Size(packets)* e varia em relação ao algoritmo anterior na fórmula utilizada no cálculo do valor inicial de H que passa a ser:

$$H = 2 + \frac{size}{536}$$

representando, assim, o valor médio estimado de pacotes enviados e recebidos necessários para satisfazer um *cache miss* de um determinado documento. Desta forma conseguem-se ambos os objectivos – um alto valor de *hit ratio* e de *byte hit ratio*.

Ambos os algoritmos (*GD-Size(1)* e *GD-Size(packets)*) possuem o mesmo mecanismo intrínseco de "envelhecimento" dos dados presentes na sua *cache*, pelo que, por maior que seja o valor inicial atribuído, caso estes não sejam acedidos posteriormente acabarão por ser removidos. Desta forma, consegue-se evitar um acontecimento conhecido por poluição da *cache*: um documento que possui um grande valor de benefício inicial mesmo que não seja acedido posteriormente, caso não exista um mecanismo de envelhecimento dos dados, poderá nunca mais ser removido da *cache* tornando-se obsoleto e ocupando espaço desnecessário. Como consequência das suas próprias características, o algoritmo *GD-Size(1)* pretende diminuir o *miss ratio*. Já o algoritmo *GD-Size(packets)* pretende por sua vez diminuir o tráfego na rede resultante dos *cache miss* que possam ocorrer.

2.2.11 Greedy-Dual-Frequency

Também denominado por *GD-Frequency*, este algoritmo surge como uma tentativa de introduzir no algoritmo *Greedy-Dual* um mecanismo de controlo de frequência de acesso aos ficheiros e resulta na definição das seguintes fórmulas de cálculo do valor de H e de prioridade (Pr):

$$H = frequency \times cost$$

$$\Pr(f) = Clock + Fr(f) \times Cost(f)$$

2.2.12 Greedy-Dual-Size-Frequency

Uma das principais desvantagens dos algoritmos *Greedy-Dual* e *Greedy-Dual-Size* é o facto de estes não terem em conta o número de vezes que um documento foi acedido no passado. Isto pode ser facilmente implementado incluindo nos cálculos realizados um valor que represente a frequência com a qual estas informações são acedidas, de forma a decidir qual a informação a ser removida da *cache*. Ao algoritmo semelhante aos anteriores que incorpora esta informação dá-se o

nome de *Greedy-Dual-Size-Frequency* (ou *GD-Size-Frequency*) e atribui o valor de H através da seguinte fórmula [Cherkasova 1998]:

$$H = frequency \times \frac{cost}{size}$$

A forma de manter a informação relativa à ordenação dos objectos em *cache* de acordo com o seu valor de prioridade é através de uma *queue* de prioridades, e o valor correspondente da prioridade é calculado da seguinte forma:

$$\Pr(f) = Clock + Fr(f) \times \frac{Cost(f)}{Size(f)}$$

em que:

- *Clock* é um mecanismo que inicia o seu valor em 0 e é actualizado cada vez que alguma informação é removida da *cache*, tomando o valor da prioridade deste ficheiro, isto é, Clock = $Pr_{evicted}$.
- Fr(f) representa a frequência com que o ficheiro f é acedido, e calcula-se adicionando uma unidade ao seu anterior valor cada vez que o ficheiro f é acedido ($Fr(f) = Fr(f) + 1$) e , cada vez que o ficheiro f, quando requisitado, não se encontrar em *cache* o valor de $Fr(f)$ toma o valor 1.
- *Size*(f) representa o tamanho do ficheiro f
- *Cost*(f) designa o custo de trazer f para a cache.

Uma explicação breve do funcionamento deste algoritmo pode ser encontrada no **Error! Reference source not found.**.

```
Se ficheiro em cache então

       Incrementar frequência de acesso ao ficheiro – Fr(f) = Fr(f) + 1
       Recalcular valor de prioridade – Pr(f)
       Reordenar queue de prioridades

Senão
       Obter ficheiro (f) do servidor
       Inicializar valor de Fr(f) = 1
       Calcular Pr(f)
       Inserir f na queue de acordo com prioridade calculada
       Calcular espaço utilizado em cache (Used)
       Se Used > Total
              Calcular conjunto de vistas a remover da cache (Remove)
              Se f não em Remove então
                     Remover elementos de Remove da cache
                     Adicionar f a cache
              Senão
                     Não adicionar f a cache
              Fim se
       Senao
              Adicionar f a cache
       Fim se
Fim se
```

Algoritmo 1 - O algoritmo 2.2.12 Greedy-Dual-Size-Frequency (proposta de [Cherkasova 1998])

Neste algoritmo (Algoritmo 1) são de salientar algumas propriedades tais como o facto de serem beneficiados (através da fórmula de cálculo de prioridade) ficheiros de tamanho reduzido o que aumenta o *hit ratio* (diminuindo assim o valor de *miss ratio*). O mecanismo *CLOCK* garante ainda, devido ao frequente aumento do seu valor de cálculo, que documentos raramente acedidos, apesar de poderem possuir um valor alto de prioridade serão eventualmente removidos da *cache* evitando assim o fenómeno de poluição da *cache*.

2.2.13 O Algoritmo de Belady

Este algoritmo, meramente teórico, prevê a remoção da *cache*, apenas da informação que será, no futuro, a última a ser requisitada. Servindo como ponto máximo no despenho que um algoritmo de manutenção de *caches* poderá atingir, este serve como ponto de referência para o teste a outros algoritmos permitindo assim medir a margem de progressão destes (quanto maior for o desfasamento entre os seus desempenhos e os de referência maior a margem de progressão do algoritmo a ser testado).

2.3 As Várias Opções de Caching

Os algoritmos anteriormente apresentados, apesar de diferentes a nível da sua implementação, pretendem atingir um mesmo objectivo – gerir a memória destinada à *cache.* Estas diferenças entre as abordagens ao problema de *caching* resultam, também, em diferentes características das soluções encontradas. De entre estas características destacam-se a simplicidade de implementação, a escalabilidade e a capacidade de ter em conta as necessidades específicas dos utilizadores do sistema, etc. Na tabela que se segue (Tabela 1) encontram-se resumidas algumas destas características que definem qual a abordagem e consequente desempenho de cada um dos algoritmos de *caching* apresentados anteriormente.

Características / Algoritmo	Estrutura de dados utilizada	Mantém informação auxiliar	Implementação simples	Escalabilidade	Adaptabilidade a padrões de pesquisa
First In First Out	Queue	Não	Sim	Boa	Não
Second Chance	Queue	Sim	Sim	Boa	Não
CLOCK	Lista circular	Sim	Sim	Boa	Não
Least Recently Used	Lista	Não	Sim	Boa	Sim
Least Frequently Used (em cache)	---	Sim	Não	Fraca	Sim
Simple time-based expiration	---	Sim	Sim	Boa	Não
Sliding time-based expiration	---	Sim	Sim	Boa	Sim
Least Benefit First	---	Sim	Não	Boa	Sim
Greedy Dual	---	Sim	Não	Razoável	Sim
Greedy-Dual-Size	---	Sim	Não	Razoável	Sim
Greedy-Dual-Frequency	---	Sim	Não	Fraca	Sim
Greedy-Dual-Size-Frequency	---	Sim	Não	Fraca	Sim
Belady	---	---	Não	---	Sim

Tabela 1 - Características dos algoritmos de gestão de caches

Capítulo 3

Arquitecturas para Sistemas de Caching

3.1 Suporte para Caching

Todos os algoritmos apresentados até ao momento têm como objectivo gerir directamente qual a informação a adicionar ou a remover dos espaços de memória reservados à cache. De forma semelhante, ainda que com outro grau de abstracção, existem outros que abordam a problemática da localização dos mecanismos de *caching*. Se, por um lado, se pode realizar *caching* apenas do lado do cliente, é também possível criar mecanismos que beneficiem todos os utilizadores de uma comunidade. Este tipo de sistemas, dos mais simples aos mais complexos, envolve, entre outros, redes *peer-to-peer*, *caching* baseado em *chunks*, etc. Algumas das questões que se pretendem ver implementadas com um sistema deste género, vão desde a gestão das caches individuais de cada um dos seus utilizadores, até à gestão de uma *cache* central (caso esta exista). Uma questão frequente prende-se com a forma de gestão das *caches* individuais de cada um dos utilizadores do sistema e sobre como realizar a comunicação entre estes de forma a que a *cache* de cada um possa satisfazer pedidos dos restantes. Numa tentativa de responder a estas e outras questões, apresentam-se de seguida alguns destes sistemas, especialmente direccionados para o ambiente multidimensional.

3.2 Utilizando Servidores Proxy distribuídos

Por norma, cada departamento dentro de uma organização possui o seu próprio interesse na informação presente num sistema OLAP. Se existir um *data warehouse* central, que sirva toda a organização, este estará encarregue de garantir a consistência e coerência dos dados de forma vertical na empresa (esquema centralizado). Se, por outro lado, cada departamento estiver encarregue de gerir a sua própria informação, de forma independente, que será posteriormente integrada com a informação dos restantes departamentos, encontramo-nos perante um sistema cuja gestão de informação se dá de forma descentralizada ou distribuída.

Como mecanismo de gestão deste tipo de sistemas, podem ser utilizados OLAP *Cache-Servers* (OCS), servidores em tudo semelhantes a *proxies* convencionais, mas com a capacidade de construírem dinamicamente as respostas às *queries* efectuadas, isto é, possuem poder de cálculo sobre os dados OLAP que possuem na sua *cache*. Outra característica importante dos OCS é a possibilidade de renovarem os seus dados incrementalmente, isto é, sem a necessidade de removerem páginas inteiras para adicionarem novas, adicionando apenas as novas informações que recebem. Um conceito importante na utilização de um OCS é o de *active caching* [Cao et al. 1999] que permite que os servidores *proxy* (neste caso OCS) tenham a capacidade de construir respostas dinamicamente através do armazenamento de uma *applet* junto com os dados mantidos em *cache* que permite manipulá-los.

3.2.1 Arquitectura do sistema

Nestes sistemas [Kalnis & Papadias 2001] distinguem-se, tal como na maioria dos sistemas *proxy* existentes, três camadas de dados: o *data warehouse*, os OCS, e a camada do utilizador final. Isto permite que um utilizador nunca questione o *data warehouse* directamente, mas sim os OCS que, cooperando entre si, tentarão devolver a resposta sem necessidade de aceder ao servidor central. Uma característica da arquitectura de um sistema deste género que poderá variar é o grau de liberdade, ou de controlo, com que um OCS toma as suas decisões, nomeadamente a nível de planos de execução das *queries* e quais os dados a manter ou remover da cache. A nível deste tipo de decisões encontramos três políticas distintas:

- **Centralizada**. Neste tipo de politica existe uma entidade centralizadora de toda a informação (um servidor central) que define planos de execução para as *queries*, mantém informação relativamente a todos os dados que os OCS possuem nas suas caches e define quais os dados que estes deverão remover e/ou acrescentar.
- **Semi-centralizada**. Aqui, o servidor central apenas define planos de execução para as *queries* através da informação que possui relativamente aos conteúdos da *cache* de cada OCS. Cabe a cada OCS gerir a informação que pretende manter ou remover da sua *cache*.
- **Autónoma**. Nesta política não existe uma unidade central de controlo de informação, neste caso todas as decisões são tomadas de forma independente pelos OCSs.

Devido ao conhecimento de toda a topologia de rede, bem como das ligações que permitem a comunicação entre os diversos OCS, o esquema centralizado resulta normalmente em bons desempenhos a nível do sistema global. Por outro lado, a sua manutenção pode revelar-se algo trabalhosa, sendo assim a sua utilização desaconselhada para grandes redes. Em pequenas redes, a utilização deste tipo de arquitectura permite atingir um bom desempenho e a manutenção da informação a nível do servidor central não se torna demasiado complexa ou trabalhosa. Por outro lado, a política autónoma de gestão dos OCS, por permitir a estes total liberdade relativamente a qual a informação a manter e quais os OCS aos quais se ligará, é um esquema altamente escalável e de muito fácil manutenção por não centralizar informação sobre os OCS em qualquer ponto da rede. Como desvantagem principal deste esquema autónomo, os planos de execução delineados para resposta às *queries* nem sempre garantem ser óptimos embora o seu desempenho seja bastante bom, sobretudo para redes com elevado número de utilizadores e de *caches*. Por seu lado, a política semi-centralizada pretende atingir um equilíbrio entre as duas anteriores permitindo uma maior escalabilidade da rede do que a política centralizada, embora menor do que a autónoma mas planos de execução, em média, melhores do que a gestão autónoma.

Outro campo de relevo na análise de um sistema deste género é a sua disponibilidade. Relativamente às políticas centralizada e semi-centralizada, pelo facto de possuírem um ponto centralizador de toda a informação relativa à topologia de rede bem como da informação presente em cada OCS, a disponibilidade deste sistema será seriamente condicionada pela disponibilidade do ponto central da rede. Se, por algum motivo, ocorrer uma falha que resulte na

inoperacionalidade deste ponto, todo o sistema será afectado, uma vez que os OCS não possuem autonomia a nível das decisões a tomar, o que pode revelar-se um grave problema no sistema. Quanto à política autónoma de gestão de informação e tomadas de decisão, a fiabilidade e disponibilidade do sistema não são, em princípio, afectadas pela disponibilidade individual de cada um dos OCS embora o desempenho possa deteriorar-se pela falta de um destes elementos.

3.2.2 OLAP Cache servers

Um *OLAP Cache Server* (OCS), tal como referido anteriormente, é em tudo semelhante a um *proxy* convencional, à excepção de algumas funcionalidades que lhe são exclusivas. Um OCS é, nesta arquitectura, a unidade básica e consiste num local reservado de memória (primária ou secundária) de nome *view pool* na qual são armazenados os dados a manter em *cache*. Fazem ainda parte de um OCS um optimizador de *queries*, que constrói os planos de execução para os vários pedidos, e uma unidade de controlo e admissão, que define as regras que ditam a inserção ou remoção da informação na *cache*.

Ao configurar um OCS é de grande importância definir a granularidade dos dados a manter em *cache*, pois isto definirá, entre outras coisas, o tipo de pedidos que este nodo da rede poderá satisfazer. A nível físico, os dados são armazenados em memória secundária e apenas trazidos para memória primária quando forem requisitados ou, mais exactamente, quando o algoritmo de *fetching* decidir que será a altura mais apropriada, podendo esta não ser, necessariamente, *on-demand,* mas seguindo uma abordagem preditiva [Ramanchandran et al. 2005] ou uma qualquer outra técnica de *fetching* de dados. A forma de armazenamento dos dados no disco poderá seguir, por exemplo, uma abordagem como a proposta em [Deshpande et al. 1998] ou [Roussopoulos et al. 1997].

O Optimizador de *queries*

Como parte integrante de um OCS, o módulo de optimização de *queries* tem como principal função definir onde a informação pretendida por um determinado pedido deverá ser requisitada e de que forma isso deverá acontecer. Devido às diferenças relativamente ao grau de liberdade entre as

diversas políticas de configuração da rede, este optimizador poderá comportar-se de várias formas distintas: autónoma, centralizada e semi-centralizada.

Política autónoma

Na política autónoma cabe a cada OCS definir o plano de execução de uma determinada *query* sem que haja intervenção de um servidor central. [Kalnis & Papadias 2001] propõem o seguinte algoritmo para ser seguido quando um pedido (q) chega a um OCS_i:

- Se o OCS_i possuir na sua cache a resposta ($answer_q$) devolvê-la-á a quem efectuou o pedido. De forma simplificada, podemos afirmar que o custo desta operação é linear com o tamanho da informação devolvida:

$$cost(q) = size(answer_q)$$

- Se o OCS_i não possui a resposta ao pedido que lhe foi dirigido, este pedido será redireccionado para outro OCS_j (a escolha deste OCS poderá seguir uma diversidade de heurísticas). A proposta de [Kalnis & Papadias 2001] é a seguinte:
  - O OCS_i envia a todos os seus vizinhos directos a querie (q)
  - Todos os vizinhos respondem dizendo se podem ou não satisfazer o pedido e, caso possam, qual o custo previsto para o fazer
  - O OCS que anuncie o menor custo para responder ao pedido efectuado será escolhido e o pedido ser-lhe-á então "oficialmente" redireccionado

- Se OCS_i não possuir a resposta exacta a q mas sim outra informação que permita derivar esta resposta, ao contrário do que seria intuitivo, ele redirecciona o pedido para os seus vizinhos directos de forma a saber qual deles poderá responder com um menor custo. Este custo é posteriormente comparado com o custo de derivar localmente a resposta à questão e a solução que apresentar menor custo total será a escolhida.

Políticas centralizada e semi-centralizada

Nas políticas centralizada e semi-centralizada mantém-se num nodo central toda a informação relativa à topologia de rede bem como à informação mantida em *cache* por cada um dos OCS. Assim, é nesse nodo central que são tomadas as decisões relativamente a qual o plano de execução de cada uma das *queries* dirigidas ao sistema. Assim, cabendo a este nodo tomar todas as decisões, não é necessária (nem permitida pela definição desta política), por parte dos OCS, qualquer intervenção, pelo que o módulo de optimização de *queries* não será utilizado.

Políticas de *caching*

Em caso de necessidade de adição de novas informações à cache é a política de *caching* que define qual a forma de o fazer. Caso o espaço existente não seja suficiente, será definido por esta política qual a informação (caso exista) a remover, originando espaço para nova informação. O algoritmo responsável por este tipo de decisão poderá ser um dos vários apresentados anteriormente, embora este varie com a política de liberdade estabelecida para os OCS.

Política autónoma

Na política autónoma poderá ser utilizado qualquer um dos algoritmos de *caching* apresentados anteriormente na secção "Algoritmos de gestão de caches", embora em [Kalnis & Papadias 2001] se proponha a utilização do algoritmo *Lowest Benefit First* com algumas alterações ao nível do cálculo do custo de computação de uma vista.

Políticas centralizada e semi-centralizada

Para as políticas centralizada e semi-centralizada é proposto em [Kalnis & Papadias 2001] um algoritmo que, apesar de considerar o benefício de uma vista e seu custo de computação (estes cálculos são realizados de igual forma na política autónoma), não é o exactamente o mesmo. Neste caso é utilizado o algoritmo *Smallest Penalty First* pois este simplifica o trabalho a ser realizado. Desta forma, apesar de se possuir informação relativa a todos os OCS o custo de realizar estes cálculos no nodo central não é demasiado alto.

3.3 Através de redes *Peer-to-Peer*

A proposta de [Kalnis et al. 2002], tal como todas as outras propostas na área do *caching* distribuído, parte do princípio que se todos os participantes de uma rede partilharem as suas caches pessoais, todos sairão beneficiados. É, nesta proposta, apresentada uma arquitectura de rede que permita esta funcionalidade (*PeerOLAP*). São ainda propostas algumas políticas de renovação e manutenção dos dados em *cache* para um sistema de *caches* distribuídas sendo a abordagem destas sempre centrada no cliente.

As principais razões para a aplicação deste tipo de técnicas a sistemas OLAP, e não a outros, assentam em vários aspectos, nomeadamente: a estrutura regular dos dados, o que simplifica a reutilização de dados armazenados; a dimensão dos resultados, que justifica o tempo que leva à pesquisa destes nos nodos vizinhos; as actualizações no(s) nodo(s) central(ais), que são pouco frequentes e possuem um tempo de vida bastante grande; e a "validade geográfica" muito grande que os dados possuam, isto é, dados de um lado do mundo serão, frequentemente, acedidos por pessoas em locais completamente diferentes do globo. Nesta arquitectura, os dados são, tal como referido em [Deshpande et al. 1998], armazenados sob a forma de *chunks,* sendo que as principais vantagens se prendem com as semelhanças do ponto de vista semântico entre os diversos conjuntos de dados armazenados, o que facilita a sua combinação e manipulação em pós-processamento, bem como uma boa utilização do espaço disponível, uma vez que não existe sobreposição dos dados. Contrariamente a [Kalnis & Papadias 2001], em que se recorre à utilização de *OLAP Cache Servers* como forma de manter os dados em *cache* partilhados por toda a rede, e que possuem a capacidade de manipulação dos dados armazenados, na arquitectura *PeerOLAP* não existe uma camada intermédia responsável apenas pelo armazenamento de dados em *cache*, cabendo aos utilizadores finais gerir uma *cache* distribuída (geralmente de grande dimensão). Outra diferença entre estes tipos de arquitecturas prende-se com o facto de a rede *PeerOLAP* ser dinâmica o que potencia um melhor aproveitamento dos recursos em cada momento.

O sistema *Piazza* [Gribble et al. 2001] é um sistema pensado para efectuar a gestão dos dados num sistema *Peer-to-Peer*, sendo que o seu principal foco se prende com a gestão da localização dos dados de forma a permitir uma melhoria dos tempos de resposta das *queries*. Nesta abordagem, o sistema está separado em termos lógicos em diversos grupos de cooperação que depois informam os seus nodos constituintes sobre quais os conjuntos de vistas materializadas possuídos pelo grupo. Existem algumas semelhanças entre este sistema e a proposta *PeerOLAP*, entre elas o facto de ambos utilizarem um mecanismo que informa os restantes nodos relativamente aos conteúdos da cache de um *peer* embora no *PeerOLAP* a fina granularidade dos dados seja um problema para este mecanismo pois obriga a uma intensa troca de mensagens.

Como *Peer-to-Peer* que é, o *PeerOLAP* é baseado em *BestPeer* [Ng et al. 2002], um sistema com o objectivo de servir de base ao desenvolvimento de novas aplicações P2P e que disponibiliza as seguintes funcionalidades:

- Funcionalidades de agentes móveis.
- Partilha de dados com uma granularidade muito fina, bem como poder computacional
- Configuração dinâmica da arquitectura de rede de forma a garantir que, em cada momento, um *peer* se encontra ligado directamente aos *peers* que lhe disponibilizam o serviço que mais lhe convém.
- Incorporação de um conjunto de servidores de resolução dinâmica de nomes independente da localização (LIGLO) que permitem reconhecer *peers* que possuem conexões flutuantes.

No que diz respeito à sua arquitectura, o sistema *PeerOLAP* consiste num conjunto de utilizadores finais que acedem a *data warehouses* e formulam *queries*. Cada um destes utilizadores possui uma *cache* própria e implementa um conjunto de mecanismos que lhe permitem indicar aos restantes membros da rede os conteúdos desta, bem como das suas capacidades computacionais. Outros utilizadores que se liguem a este poderão questioná-lo acerca de dados, tal como fariam num *data warehouse,* podendo este responder-lhes caso possua a informação pretendida ou reencaminhar o pedido para os seus vizinhos directos. Os resultados, quando encontrados, quer num caso quer noutro, são enviados directamente para o utilizador que formulou a questão em primeira instância.

Tal com em outras propostas que seguem uma arquitectura descentralizada, um desafio que se apresenta a este sistema é a necessidade de criação de um mecanismo que evite a propagação descontrolada de mensagens, o que iria, como sabemos, congestionar a rede, deteriorando os seus desempenhos. A solução encontrada passa pela definição de um número máximo de "saltos" que uma mensagem poderá dar até perder a sua validade, o que é até intuitivo, pois uma mensagem ao fim de um determinado número de saltos, mesmo que possa encontrar um *peer* que possua a resposta pretendida, será muito dificilmente fornecida no melhor período de tempo possível. Outro problema que se levanta é quando uma mensagem se encontra a ser reenviada para outros utilizadores, ainda antes de atingir esse número máximo de saltos e o único local para onde esta pode ser reenviada é o *data warehouse.* Neste caso, a mensagem não é retransmitida

para este nodo central pois isto poderia levar à repetição de mensagens enviadas ao *data warehouse* o que quebra todo o objectivo deste tipo de sistemas de *caching*. É ainda possível que uma mensagem, quando o número de saltos definido for demasiado alto, entre em *loop.* Com o objectivo de diminuir este problema, existe um mecanismo que testa se a mensagem já foi processada anteriormente e, em caso afirmativo, rejeita-a. Alternativamente reencaminha a mensagem de forma normal. Quanto ao utilizador que iniciou o processo de descoberta, como não possui informação relativamente a quais os utilizadores que lhe irão responder ou, sequer, sabe se a resposta se encontra nestes, espera até obter a resposta completa ou, em alternativa, até um tempo máximo de espera ser atingido, altura em que redirecciona directamente para o *data warehouse* o pedido da informação ainda em falta. Quando se encontrar na posse de todos os dados, o utilizador que iniciou o pedido de informação decide se estes deverão ser guardados na sua *cache* local ou ignorados, com base na atribuição de uma métrica de benefício. Existe ainda a possibilidade destes dados, caso tenham sido enviados directamente do *data warehouse,* serem armazenados na *cache* dos vizinhos deste utilizador.

Devido à natureza dinâmica destas redes - qualquer utilizador possui liberdade de se ligar ou desligar desta quando lhe for mais conveniente -, é necessário que exista um mecanismo que controle a informação dos utilizadores activos, nomeadamente acerca de quais os *data warehouses* aos quais este se liga, qual a sua localização física, velocidade de ligação à rede, etc. Este é o papel dos servidores LIGLO. Quando um utilizador se pretende ligar à rede *PeerOLAP*, questiona um destes servidores relativamente a potenciais "vizinhos" com os quais possa estabelecer ligação. É tarefa do utilizador, de forma individual, decidir a que outros utilizadores se vai ligar, sendo o fornecimento destas informações a única tarefa dos servidores LIGLO, que são, nesta arquitectura, o único elemento que faz com que esta não seja totalmente descentralizada, embora possam ser utilizados alguns mecanismos alternativos caso este seja um objectivo rígido da implementação a efectuar.

Como seria de esperar, nada garante que o conjunto inicial de "vizinhos" de um utilizador seja óptimo, ou ainda que nenhum destes se irá manter na rede durante um longo período de tempo. Existem, para este efeito, mecanismos que permitem a manutenção de ligações efectuadas (criação de novas ligações, eliminação de ligações actuais que não se revelem benéficas, etc.), mantendo-se constante o objectivo da diminuição do tempo de resposta das *queries* efectuadas. É

ainda definido, a nível do sistema, um número máximo de vizinhos aos quais um utilizador se poderá ligar, evitando assim um grande número de mensagem na rede que diminuiriam o rendimento desta.

Modelos de custos

Os modelos de custo apresentados em [Kalnis et al. 2002] englobam, entre outras métricas, o custo de operações de cálculo sobre um nodo, num determinado cliente, e o custo de transferência de *chunks* entre nodos. Desta forma, é possível a um nodo questionar todos os seus vizinhos sobre qual o tempo que lhe levam a fornecerem os dados necessários e escolher o mais vantajoso. Destes destaca-se o modelo que permite calcular o custo total de responder a um pedido de um *chunk* c, a partir do nodo P, utilizando dados provenientes de uma *query* anterior mantida na cache Q:

$$T(c, Q \rightarrow P) = S(c, Q) + N(c, Q \rightarrow P)$$

em que $S(c, Q)$ corresponde ao custo de computação do *chunk* c no nodo Q, e $N(c, Q \rightarrow P)$ representa o custo de transferência do *chunk* c de Q para P.

Processamento de *queries*

Esta análise parte do princípio de que um *data warehouse* é um sistema apenas de leitura, pelo que os seus utilizadores não poderão emitir pedidos para a sua actualização, apenas poderão requisitar informação nele armazenada. Caso o *data warehouse* seja actualizado, este deverá informar todos os seus clientes ou, em alternativa, poder-se-á recorrer à utilização da noção de "tempo de vida" da informação fornecida por este. As duas alternativas, apresentadas de seguida variam no que concerne ao processamento de *queries*, sobretudo no esforço dispendido na construção do plano de execução de um determinado pedido.

Eager Query Processing (EQP)

Assumindo que um utilizador *P* emite um pedido *q*, este algoritmo terá o seguinte comportamento [Kalnis et al. 2002]:

1. A *query* é decomposta em *chunks* com a mesma granularidade que a vista representativa. Seja C_{all} o conjunto de todos os *chunks* necessários.
2. *P* verifica que informação existe na sua cache local. Seja C_{local} a informação encontrada na cache local de *P e* C_{miss} a informação restante em falta.
3. *P* envia uma mensagem aos seus vizinhos $Q_1,\ldots,Q_n$ pedindo o conjunto C_{miss}. Se Q_i possuir um subconjunto de C_{miss} então estima o custo $T(c_i, Q \rightarrow P)$ para cada um dos *chunks* e envia estas informações a *P.* Caso Q_i não possua qualquer informação não responde. Em qualquer dos casos, Q_i propaga o pedido de C_{miss} para todos os seus vizinhos recursivamente até o número máximo de "saltos" ser atingido.
4. *P* recebe respostas por um período de tempo determinado (*t*) depois do qual presume que não serão de esperar mais resultados.
5. Seja C_{peer} o subconjunto de C_{miss} que foi encontrado na rede PeerOLAP. *P* constrói um plano de execução para C_{peer} de forma "greedy". Um *chunk* c_i aleatoriamente escolhido de C_{peer} e é atribuído a Q_i, sendo Q_i o vizinho que poderá fornecer c_i com o menor custo. De seguida um *chunk* c_j é seleccionado do subconjunto restante de C_{peer}. Seja Q_j o vizinho que fornece c_j com o menor custo. Se Q_i também possui este *chunk*, o algoritmo verifica se o custo total $T(\{c_i, c_j\}, Q_i)$ de obter ambos os *chunks* de Q_i é menor do que $T(c_i, Q_i) + T(c_j, Q_j)$ caso no qual o *chunk* c_j é também obtido de Q_i. O processo desenrola-se da mesma forma para os restantes subconjuntos de C_{peer}. Note-se que obter vários *chunks* simultaneamente de um mesmo vizinho poderá resultar num custo inferior pois o custo de inicializar a ligação é partilhado entre eles.
6. *P* incializa as ligações directas com os diversos vizinhos indicados pelo plano de execução desenhado e requisita os *chunks* correspondentes. Os *chunks* que não tenham sido entretanto eliminados da cache dos vizinhos são, então, enviados a *P.* Seja $C_{evicted}$ o conjunto de *chunks* eliminados das caches dos vizinhos.
7. O conjunto de *chunks* C_{DW} que ainda estão em falta é: $C_{DW} = C_{miss} - (C_{peer} - C_{evicted})$, conjunto este que será obtido por *P* directamente do *DataWarehouse.*
8. *P* calcula a resposta e retorna-a para o utilizador. Os novos *chunks* são enviados para o gestor de cache e qualquer adaptação na arquitectura da rede será então realizada.

Este tipo de abordagem visa obter uma grande quantidade de *chunks* através da rede *PeerOLAP* sem necessidade de recorrer ao *data warehouse* mas tem como principal desvantagem o grande número de mensagens que troca, muitas delas completamente desnecessárias (ou mesmo ambíguas) não compensando, não raras vezes, o esforço dispendido nesta procura. Se tivermos ainda em consideração que alguns dos *chunks* poderão ser obtidos através de cálculos aplicados aos *chunks* já obtidos, a complexidade da abordagem apresentada aumentaria em muito, embora este modelo esteja preparado para lidar com este tipo de complexidade. Uma alternativa a este algoritmo é uma pesquisa menos "intensiva" de dados, relaxando assim o critério de pesquisa e diminuindo em grande escala o número de mensagens trocadas entre os vizinhos.

Lazy Query Processing (LQP)

Em relação à política anterior, esta distingue-se sobretudo no esforço que é dispendido na fase de construção do plano de execução da *query,* mais especificamente no ponto 3 do algoritmo anterior, uma vez que tenta reduzir o número de vizinhos visitados. Nesta segunda abordagem (mais conservadora em termos de consumo de recursos) uma mensagem reencaminhada por um utilizador será reenviada apenas para os seus vizinhos mais benéficos. Em adição, se um vizinho Q_i puder devolver um determinado *chunk* c_i fá-lo-á e este será removido das mensagens a serem reencaminhadas para os seus vizinhos. Como consequência desta alteração, se Q_i puder devolver todos os *chunks* necessários, a mensagem não será reencaminhada. Desta forma o número de mensagens torna-se $O(k \cdot \square_{max})$ sendo que no algoritmo anterior seria $O(k^{\square max})$.

Políticas de *caching*

De forma a seleccionar quais os *chunks* que deverão ser mantidos ou removidos de uma *cache* é necessário ser-lhes atribuída uma métrica para quantificar o seu benefício. De entre as diversas métricas existentes, em [Kalnis et al. 2002] sugere-se a seguinte:

$$B(c,P) = \frac{T(c, Q \rightarrow P) + a \cdot H(P \rightarrow Q)}{size(c)}$$

Nesta expressão, $H(P \rightarrow Q)$ representa o número de "saltos" entre *P* e *Q* e *a* uma constante que representa um *overhead* pago pelo envio de uma mensagem. Intuitivamente, valores altos de *H()*

indicam que se trata de um valor de difícil acesso pelo que será benéfico mantê-lo localmente - o valor de benefício é normalizado dividindo custo de o obter pelo seu tamanho. Embora esta medida seja semelhante a *ClockBenefit* [Deshpande et al. 1998], a métrica $\frac{|D|}{n}$ não é aplicável pois neste caso permite-se a pré-agregação de dados no *data warehouse*, logo o custo computacional de um *chunk* depende do conjunto de vistas materializadas. Na rede *PeerOLAP* permite-se a replicação de *chunks,* mas esta deverá ser utilizada apenas quando absolutamente necessária, uma vez que consome espaço que poderia ser utilizado com nova informação.

Algoritmo de gestão da cache – *Least Benefit First*

O gestor de cache em cada um dos utilizadores é o algoritmo Least Benefit First, que remove, de cada cache, os elementos que revelem possuir uma métrica de benefício a si associada, com valor mais baixo. Contrariamente a este algoritmo, que gere a cache de cada cliente de forma individual, os algoritmos que se apresentam de seguida têm como objectivo gerir as ligações directas entre *peers* e garantir um aumento progressivo do grau de colaboração entre estes.

Isolated Caching Policy (ICP)

Intuitivamente, o ICP mantém a autonomia de gestão das caches a nível local por parte dos utilizadores, embora pretenda tirar partido dos seus vizinhos de uma forma "*greedy*". Segundo esta política, um utilizador publica os conteúdos da sua cache, mas não tem em conta o número de vezes que estes são acedidos pelos seus vizinhos. Isto é, mesmo que um *chunk* seja acedido por outro utilizador, este não verá o seu peso actualizado positivamente. Como consequência desta pequena alteração ao algoritmo LBF, se um *chunk* não for benéfico ao utilizador que o mantém ele será, eventualmente, removido da cache mesmo que seja benéfico aos seus vizinhos.

Hit Aware Caching Policy (HACP)

Contrariamente ao ICP, o HACP tem em conta os *hits* dos vizinhos aos elementos da sua cache. O algoritmo LBF possuía um mecanismo de colaboração que consistia na actualização do valor do peso de um *chunk,* aquando de um *hit* por parte de um dos seus vizinhos. Nesta política este esforço é levado um passo mais além pois sempre que existe um *hit* na cache de um utilizador, este aumenta o valor do seu peso o que origina um maior nível de cooperação entre os utilizadores.

Voluntary Caching

Mais do que uma política completamente nova de *caching*, este algoritmo, que poderá ser utilizado em conjunção com qualquer um dos apresentados anteriormente, pretende tirar partido do facto de poderem existir, nos utilizadores, recursos não utilizados. Imagine-se que um determinado utilizador *P* possui na sua cache um conjunto de *chunks* de grande valor e que obtém um novo directamente do *data warehouse,* mas não o pode armazenar, pois a sua cache encontra-se totalmente preenchida e o valor do peso dos *chunks,* que seria necessário eliminar, é superior ao do novo *chunk.* Neste caso, *P* questiona os seus vizinhos se alguém pretende armazenar este novo *chunk* e, em caso afirmativo, envia esta informação para o vizinho *Q* que manifestou esse interesse. Em caso de existirem vários vizinhos interessados, o utilizador *P* escolhe aquele que possuir maior relação $B(c,Q) - B(victims,Q)$. Devido aos custos de transferência o benefício de adicionar este novo *chunk c* à cache de *Q* será:

$$B(c,Q) = \frac{T(c, DW \rightarrow P) - T(c, P \rightarrow Q)}{size(c)}$$

em que DW representa, obviamente, o *Data Warehouse*.

Reorganização da rede de *peers*

Um dos objectivos da reorganização dinâmica dos *peers* é a tentativa de permitir o seu agrupamento em vizinhanças que possuam semelhanças nos seus padrões de utilização. Desta forma, aumentam-se as probabilidades de um vizinho poder satisfazer um seu pedido directamente através dos seus vizinhos mais directos, sem necessidade de recorrer a outros pontos da rede, potencialmente mais dispendiosos em termos de tempo de resposta. Com este objectivo definido, levanta-se agora o problema acerca de quais os vizinhos que um peer deverá seleccionar, pois o ideal seria manter ligações directas com todos os elementos da rede - infelizmente tal revela-se uma tarefa demasiado dispendiosa. [Bakiras et al. 2005] propõem que se aborde o problema como um caso de especial de *caching*, no qual são mantidas em "cache" as *x* ligações mais benéficas e destas escolhidas as melhores *y* ligações ($y \leq x$), podemos atribuir uma métrica de benefício a cada uma das ligações armazenadas e gerir esta informação como se de uma cache se tratasse, neste caso, recorrendo à política LFU. Por uma questão de estabilidade, o conjunto de vizinhos não

é alterado sempre que se verifica uma alteração na cache LFU mas apenas quando se satisfazem um certo número de pedidos (sendo este número um parâmetro do sistema). Note-se ainda que as ligações estabelecidas são assimétricas [Bakiras et al. 2005].

3.4 Active caching

A técnica conhecida como active *caching*, referida em [Cao et al. 1999] e, posteriormente, em [Loukopoulos et al. 2005], apresenta-se como uma solução ao problema de criação de *caching* de informação dinâmica em proxies Web. Se, por um lado, a realização de *caching* de páginas HTML estáticas é uma prática corrente, por outro, a tentativa de realizar *caching* de *queries* OLAP (e respectivos resultados) em proxies Web não o é. Isto deve-se ao facto destas proxies não estarem preparadas para manter informação dinâmica e, sobretudo, por não possuírem os mecanismos necessários para a tratar. Esta nova abordagem ao problema de *caching* de queries OLAP em ambientes *proxy*, permite resolver este tipo de problemas à custa da associação de *applets* a cada url. Uma *applet* é um pequeno pedaço de código, escrito numa linguagem que garanta a independência de plataforma, possivelmente em JAVA. Estas *applets* têm como objectivo a realização de tarefas tais como, para uma determinada *query*, determinar qual a acção a realizar pelo servidor *proxy*, quer esta seja devolver ao utilizador uma nova página fornecida pela *applet*, quer seja devolver ao utilizador uma página já existente em cache. Com este tipo de variedade de soluções é até possível que a *applet* possa realizar operações sobre informações mantidas em cache, por exemplo, agregações de dados e construção de páginas Web a serem devolvidas ao utilizador.

Quando um *proxy* é contactado por um utilizador, este tem autonomia para decidir não invocar a applet associada ao url requisitado, embora seja forçado a reencaminhar o pedido do utilizador directamente para o servidor central. Isto acontece assim, pois a única restrição que é imposta à *applet* é o facto de esta não poder devolver ao utilizador informação mantida em cache sem invocar a *applet* associada. Caso este "contrato" não fosse cumprido, poderia acontecer que o utilizador estivesse a consultar informação que, apesar de ter sido mantida em cache, já não se encontrar actualizada (ou até mesmo correcta).

Com este tipo de abordagem é possível retirar uma grande quantidade de carga computacional do servidor, pois, embora alguns dos pedidos sejam ainda direccionados para este, a grande maioria é já tratada pelas *applets* de forma independente do nodo central. Desta forma, a utilização de *applets* nos servidores *proxy*, distribui a carga computacional por todos estes, aumentando assim a disponibilidade e fiabilidade de todo o sistema.

Em [Cao et al. 1999] foi definida uma interface à qual deverão obedecer todas as *applets*, bem como um protocolo de actuação das *proxies* envolvidas neste sistema. De entre as normas regentes do protocolo destacam-se as seguintes:

- Caso um documento presente na cache seja requisitado pelo utilizador, o *proxy* optará por invocar a *applet* associada a este pedido ou, em alternativa, reencaminhar o pedido directamente para o servidor central.
- Se, por algum motivo, a execução da *applet* falhar, o pedido será directamente reencaminhado para o servidor central.
- Aquando do sucesso da execução de uma *applet*, o *proxy* deverá, em todos os casos, tomar a acção resultante desta execução.

No que diz respeito à interface a ser implementada pelas *applets*, a interface *ActiveCacheInterface* apenas define um método, de implementação obrigatória a todas as *applets*, com a seguinte assinatura:

```
public abstract int FromCache(String User_HTTP_Request, String
Client_IP_Address, String Client_Name, int Cache_File, int New_File)
```

e os valores de retorno possíveis (e consequentes acções) são:

- 1, o conteúdo de *New_File* deverá ser devolvido ao utilizador como resposta ao seu pedido http.
- 0, o pedido poderá ser respondido com a informação mantida em *Cache_File*.
- -1, o pedido deverá ser reencaminhado para o servidor

Existe ainda uma classe cujos métodos deverão ser invocados pela *applet* para que esta veja os seus pedidos realizados: a classe *ActiveProxy*. Nesta classe são disponibilizadas funcionalidades como o acesso aos ficheiros, a interrogação das caches, a gestão de questões relacionadas com concorrência, bem como o reencaminhamento de pedidos para os servidores centrais. Em [Cao et al. 1999] são apresentados alguns exemplos de *applets* que realizam tarefas como a gestão de permissões em processos de login, a gestão de anúncios dinâmicos em páginas Web (qual deverá ser apresentado, frequências de apresentação, etc), a gestão de informações específicas para cada utilizador, etc.

Estas primeiras *applets* a serem construídas têm como objectivo, não só testar a viabilidade deste paradigma de *caching* mas também aumentar o conhecimento relativo a este, permitindo uma rápida evolução e implementação de novas funcionalidades. Existe ainda uma outra classificação, com duas possibilidades, para este tipo de *applets* – negociadas e não-negociadas. As *applets* não negociadas são aquelas que provêm de servidores não certificados e têm como principal objectivo diminuir a carga de rede na entrada do servidor. As *applets* negociadas, por seu lado, provêm de servidores certificados e pretendem aumentar a disponibilidade e fiabilidade dos serviços prestados pela globalidade do sistema.

Em [Loukopoulos et al. 2005], em detrimento de uma explicação exaustiva da forma de implementação deste tipo de sistemas, os autores cingem-se à análise das motivações e consequências dessa implementação. Uma importante característica da utilização de *active caching* é o facto de esta técnica diminuir a latência de rede correspondente à transmissão de pedidos entre a *Web proxy* e o servidor Web, sendo as páginas construídas na *proxy*, que fica em termos de custo de transmissão, mais próxima do utilizador final. Desta forma, e apesar de se reconhecer que, trabalhando sob carga normal, o servidor Web é mais rápido a construir uma página do que um servidor *proxy* convencional, a construção da página no último permite reduzir consideravelmente a largura de banda gasta em trocas de mensagens entre estes dois pontos da rede. Os modelos de custo considerados têm em conta factores como a latência da rede, os custos relativos ao não encontrar uma determinada informação na cache (frequentemente referidos como *miss penalty*), as métricas de benefício associadas a uma determinada vista, etc.

3.5 Caching utilizando *chunks*

A proposta de [Deshpande et al. 1998] pretende contornar os problemas do *caching* a nível da *query* permitindo que se utilizem dados mantidos em cache para responder parcialmente à questão apresentada, sendo a restante parte possivelmente obtida directamente a partir do servidor. Isto é conseguido através da divisão dos dados em *chunks* que serão armazenados em cache. Como o nível de agregação dos *chunks* é inferior ao das *queries*, estes poderão ser utilizados para calcular resultados parciais para as questões. Através da utilização de mecanismos de mapeamento entre os dados da *query* e o número do *chunk* correspondente, é possível determinar o conjunto de *chunks* necessários ao cálculo da solução de uma questão. Este conjunto é posteriormente dividido em dois conjuntos disjuntos: o conjunto dos *chunks* presentes em *cache* e o conjunto de *chunks* em falta que terão de ser obtidos do servidor. Desta forma é possível reduzir, em alguns casos em grande percentagem, a quantidade de informação que será necessário obter directamente do servidor central de dados, uma vez que a composição dos *chunks* obtidos da cache poderá responder a uma grande parte da questão a responder.

3.5.1 Benefícios de utilizar *chunks* para realização de *caching*

Tal como referido anteriormente, existem diversas vantagens provenientes da utilização de *chunks* como unidade básica de armazenamento da informação a manter em cache. Para além das vantagens, referidas anteriormente, e que se prendem com a possibilidade de combinação dos dados obtidos da cache para responder à questão posta, são ainda de destacar outras vantagens desta técnica de *caching*:

- Granularidade dos dados armazenados. Os dados armazenados desta forma permitem que sejam realizadas operações sobre os dados de forma a calcular os resultados das *queries* pretendidas e, ainda, que os *chunks* mais requisitados sejam, quase garantidamente, mantidos em cache.
- Uniformidade. Devido à uniformidade dos dados a utilização de dados combinados para obter uma resposta é mais simples e evita, ainda, a necessidade de verificação relativamente à sobreposição dos dados armazenados.

- *Closure property of chunks*. Esta propriedade define a possibilidade de utilizar *chunks* em vários níveis diferentes de agregação para obter *chunks* noutro nível de agregação. A utilização de uma organização orientada ao *chunk* no servidor diminui ainda mais o *miss penalty* associado ao facto de um *chunk* não se encontrar em *cache.*
- Não é armazenada informação redundante. Realizando caching ao nível do *chunk* é possível partilhar os *chunks* que possuam porções equivalentes, isto é, eliminar a redundância de dados armazenados o que diminui a quantidade de memória utilizada aumentando assim a memória disponível para a adição de novos dados e assim aumentando o desempenho do sistema a nível global.

3.5.2 Ordenação das dimensões

Para se poder armazenar correctamente os *chunks*, é necessário realizar uma divisão dos valores das dimensões em intervalos. Este processo levará, caso seja efectuado de forma correcta, a um melhor armazenamento dos dados pelo que revela ser da maior importância. Uma boa forma de dividir as dimensões é seguir a lógica da hierarquia de cada uma delas e a criação de um mapeamento estrutural de nome *domain* assegurará, daí em diante, a correspondência correcta entre o valor de uma dimensão e o seu número. Este tipo de conversão poderá representar um papel semelhante ao de uma chave de substituição, frequentemente utilizadas em Data Warehousing ou de um índice de valor para cálculo de *offset* de células em MOLAP [Sarawagi 1997].

Se numa dimensão existirem hierarquias múltiplas deverá ser utilizada apenas uma delas pois a lógica pretendida, provavelmente, se manterá nas restantes. Isto é possível de se realizar, pois, em OLAP, os membros de cada nível são, tipicamente, conhecidos em antecipação e, além disso, manter-se-ão inalterados por períodos suficientemente grandes de tempo [Karayannidis & Sellis 2001]. Uma abordagem frequentemente utilizada é a de definir uma ordem específica para cada um destes membros [Sarawagi 1997], [Roussopoulos et al. 1997], [Markl et al. 1999] e [Vassialidis & Skiadopoulos 2000].

3.5.3 Definição de intervalos para os *chunks*

Os intervalos nos quais serão divididos os *chunks* identificarão as fronteiras das regiões destes. Caso não existam hierarquias definidas para uma dimensão, esta deverá ser dividida de acordo com intervalos uniformes ao longo de todos os seus valores. Caso contrário, a hierarquia deverá ser respeitada no processo. O algoritmo para a criação de intervalos de *chunks,* poderá ser o seguinte:

```
Tamanho_hierarquia = tamanho da hierarquia da dimensão
      Dividir nível 1 em intervalos uniformes
      Para(i=1 até Tamanho_hierarquia - 1)
            Para cada intervalo de chunk no nível 1
                  Seja R = intervalo de valores
mapeados no nível 1
                  Dividir R em intervalos uniformes
```

Algoritmo 2 - Criação de intervalos para os chunks (proposta de [Deshpande et al. 1998])

3.5.4 Organização do ficheiro de *chunks*

Apesar de reorganizados segundo *chunks*, num ficheiro com esta organização a informação é, ainda, armazenada segundo tuplos sendo criado um índice que permita, dado o número de um *chunk* aceder a todos os tuplos que o constituem. Um ficheiro de *chunks* possui duas interfaces: a relacional (que permite a realização de *queries* SQL) e a interface baseada em *chunks* (que permite aceder directa e individualmente a cada um dos *chunks*).

3.5.5 Implementação

Quando se pretende implementar uma técnica de caching baseado em *chunks* são vários os passos que deverão ser seguidos para que esta implementação seja o mais efectiva e correcta possível. Ao longo desse processo, numa primeira fase o processo consistirá num conjunto de considerações que deverão ser tidas em conta para adaptar a técnica à situação específica em causa, deverão ser

tomadas várias decisões que definirão o desempenho global final do sistema implementado. Uma segunda fase, não abordada neste documento passará pela monitorização da utilização do sistema com o objectivo de corrigir possíveis erros ou optimizar processos que se revelem ter sido menos bem implementados.

Tamanho dos *chunks*

Utilizar *chunks* de menor tamanho melhora a granularidade dos dados armazenados pois como qualquer *query* é calculada em função dos *chunks* que envolve e os *chunks* nas fronteiras da *query* possuem alguma informação extra e envolvem cálculos extra, se for possível diminuir o número deste tipo de *chunks* conseguimos diminuir os recursos computacionais utilizados. Isto é particularmente importante no caso de queries altamente agregadas uma vez que cada tuplo resulta da agregação de muitos valores e envolve muito custo de computação. O valor definido para o tamanho dos *chunks* não deverá, no entanto, ser muito pequeno pois isso trará problemas ao nível do número de *chunks* que é necessário utilizar para responder a uma determinada *query* (este número aumenta com a diminuição do tamanho de cada *chunk*).

Análise das queries

Para que valores armazenados de *chunks* possam ser reutilizados, é necessário que se verifiquem as seguintes condições:

1. Níveis de agregação. Os resultados armazenados terão de estar no mesmo ou inferior nível de agregação dos pretendidos.
2. Conjunto de projecções. O conjunto de projecções da *query* deverá ser um subconjunto dos resultados armazenados.
3. Lista de valores seleccionados. Os predicados de selecção podem ser divididos em dois grupos: os de *group-by* e os de não *group-by*. É necessário que os segundos sejam exactamente iguais entre a *query* e os dados armazenados. Quanto aos predicados de *group-by*, estes não necessitam de ser coincidentes pois poderão ser utilizados apenas parte dos dados armazenados. Este tipo de comparação de predicados é um problema NP-completo [Sun et al. 1989] pelo que a literatura recente opta por uma de duas estratégias: ou não considera sequer esta questão [Harinarayan et al. 1996] e [Baralis et al. 1997] ou apenas inclui a possibilidade de igualdade de predicados [Shim et al. 1999].

Caso estas condições se verifiquem, é então possível analisar os *chunks* armazenados para que estes sejam reutilizados na resposta à nova *query* efectuada.

Cálculo do número dos chunks

Relativamente aos predicados de selecção de *group-by* referidos anteriormente, é necessário que estes sejam convertidos numa lista de números de *chunks* para que seja possível verificar a sua presença na cache. Para isto é necessário definir uma função de conversão entre o índice do *chunk* e o número do *chunk* numa determinada dimensão. Esta função chama-se *getChNum()*e representa-se da seguinte forma:

$$RC_i = \left\{ \left(\frac{x}{c_i} \right) | x \in R_i \right\}$$

em que R_i é o conjunto de valores seleccionados na dimensão *i*, RC_i é o conjunto de índices dos *chunks* de da dimensão *i.* Se existirem *k* dimensões sobre as quais são executados *group-by* obteremos *k* conjuntos deste género. De seguida realiza-se um produto cartesiano entre estes conjuntos e para cada valor deste produto utiliza-se a função *getChNum()* para obter o número do *chunk* correspondente. O algoritmo apresenta-se de seguida:

```
CNums=∅
Para s em (RC0xRC1…xRCk)
      Num =getChNum(s)
      CNums =CNums ∪ {num}
```

Em que *CNums* é o conjunto final de *chunks* que é necessário obter da cache.

Algoritmo 3 - Cálculo do conjunto de chunks necessário

Divisão das *queries*

Uma vez calculada a lista de *chunks* necessários esta é dividida em dois conjuntos: os que se encontram presentes na cache e os que terão de ser obtidos directamente do servidor. Para cada

chunk em falta é necessário saber quais os *chunks* da tabela base é necessário agregar para o obter. O número de cada *chunk* é convertido num conjunto de índices de *chunks* (utilizando a função inversa de *getChNum()* o que indica os índices dos *chunks* do nível agregado. É necessário ainda converter estes índices nos índices correspondentes dos *chunks* no nível de agregação mais baixo. Posteriormente, e obtidos todos os *chunks* em falta, poderá ser necessário algum tipo de pós-processamento para obter o resultado final e decidir, ainda, se estes novos *chunks* deverão ser admitidos na cache ou não.

Ficheiro de *chunks* no servidor

Relativamente à implementação de suporte a ficheiros de *chunks* no servidor existem duas alternativas principais. Se se pretender uma implementação completa de suporte a este tipo de ficheiros será necessário definir um novo tipo de ficheiros no servidor permitindo assim que sejam disponibilizadas por este duas interfaces distintas (a interface relacional habitual e uma interface de interacção com ficheiros de *chunks*) o que permitirá que, caso o servidor seja questionado relativamente a este novo tipo de dados eles sejam devolvidos sem necessidade de cálculos adicionais. Em alternativa poderá apenas utilizar-se um mecanismo que dependerá de uma camada intermédia que realize a separação entre os dados relacionais (tuplos) e os *chunks* que poderão depois ser adicionados à cache. Este mecanismo implicará a realização de três fases distintas:

- Adicionar um novo atributo à relação para identificar o número do *chunk.*
- Ordenar o ficheiro segundo o número dos *chunks* para que o *clustering* dos dados seja mais benéfico em termos de desempenho.
- Adicionar um índice aos dados sobre o número dos *chunks* para que o acesso seja privilegiado neste campo.

3.6 Gestão dinâmica de organização de servidores *proxy*

[Bakiras et al. 2005] propõem alguns algoritmos que permitem agrupar dinamicamente *proxies* em "vizinhanças", reagrupando-as sempre que necessário. O problema poderá ser analisado como um problema de *caching* de segundo nível, em que a informação a manter se refere aos melhores

vizinhos de um determinado proxy *v*. Quando v determina que um *proxy* v_i, que não é seu vizinho, lhe traria maiores benefícios, adiciona-o à sua lista de vizinhos, eliminando, caso necessário, outro *proxy* v_j desta lista. Por regra, v_j será o *proxy* menos benéfico para v. É necessário, no entanto, distinguir dois tipos de arquitecturas no que respeita às ligações entre as diversas *proxies*:

- *Caching* unidireccional (ou assimétrico). Um *proxy*, apesar de se ligar a um determinado número de vizinhos para benefício próprio, não obriga a que estes se liguem a si, isto é, pelo facto de um *proxy v* se ligar directamente a outra *w* não significa que *w* possua uma ligação directa a *v* para benefício de *w*. A forma de representar este tipo de ligações poderá ser através de um grafo direccional (um nodo A só se encontra ligado a outro, B, se existir uma aresta direccionada do primeiro para o segundo (Figura 1).
- *Caching* bidireccional (ou simétrico). Nesta arquitectura de ligações entre os *proxies* um *proxy v* só se encontra na vizinhança de *w,* caso exista uma ligação bidireccional entre *v* e *w*, isto é, se ambos os nodos usufruírem da cache um do outro. Frequentemente este tipo de ligações é representado graficamente através de um grafo não orientado (ver Figura 2).

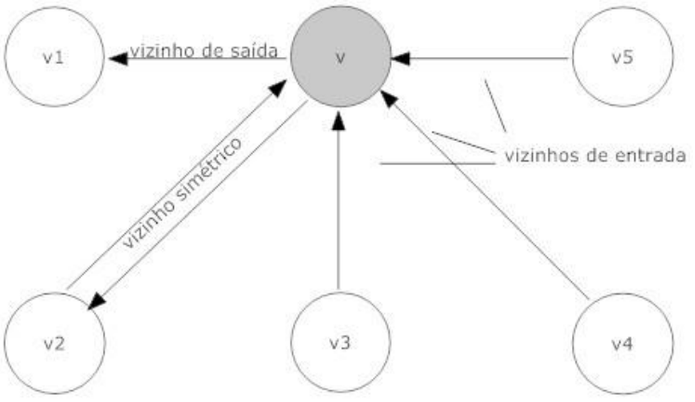


Figura 1 - Ligações unidireccionais entre proxies

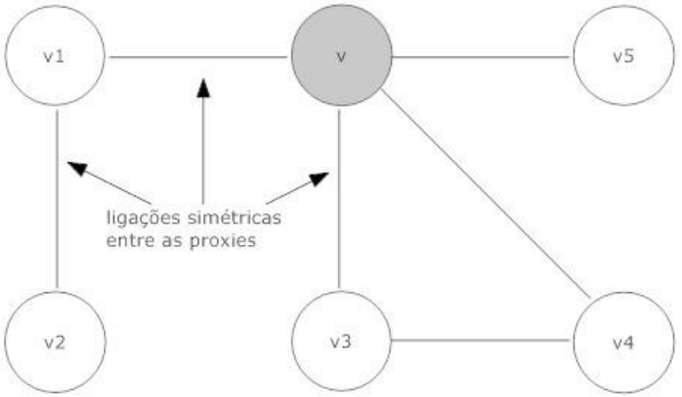


Figura 2 - Ligações bidireccionais entre proxies

3.6.1 *Caching* unidireccional

Numa arquitectura que visa a troca de mensagens entre os nodos que possuam ligações directas entre si, é de vital importância que estas ligações sejam as mais vantajosas possíveis. Tendo em vista a optimização destas ligações a nível de relevância, é proposta em [Bakiras et al. 2005] a técnica que se apresenta de seguida e que pretende delegar nos próprios nodos a responsabilidade de seleccionar as ligações a efectuar com os seus vizinhos. Desta forma, é possível que um nodo reorganize de forma independente o seu grupo de vizinhos para que o desempenho dar resposta às questões que lhe sejam colocadas, seja o melhor possível.

Reorganização da arquitectura do sistema

Quando uma reconfiguração da organização do sistema é necessária, a nova vizinhança de uma *proxy v* é definida como o conjunto *k* de *proxies* que forneceram a *v* o maior número de respostas positivas. É possível que algumas destas *proxies* já se encontrem na vizinhança de v pelo que não será necessário tomar qualquer atitude quanto a estas. Relativamente às restantes, *v* requisitará as suas *chache digests* e mantê-las-á em cache no lugar das *proxies* eliminadas da sua vizinhança. Como condições de renovação da vizinhança são de ressalvar dois casos extremos: quando esta renovação se dá imediatamente após cada resposta positiva, o que fará com que o algoritmo LFU

se comporte tal como o LRU e o caso em que estas renovações se dão apenas ao fim de um período de tempo muito longo o que fará com que esta organização se assemelhe a uma organização estática.

Em [Bakiras et al. 2005] refere-se que, como o objectivo é maximizar o número de respostas positivas, apenas se tem em conta o número destas e não o tamanho das páginas devolvidas por uma resposta o que influenciará, certamente, os resultados dos testes quando aplicados a uma tecnologia específica. Uma forma de contornar este problema será o de atribuir, como frequentemente se faz, uma métrica de benefício associada a uma informação permitindo assim medir com maior exactidão a vantagem ou desvantagem de adicionar (ou remover) uma *proxy* à vizinhança de outra.

3.6.2 *Caching* bidireccional

Caching bidireccional utilizando o algoritmo LRU (B-LRU) é em tudo semelhante ao seu equivalente unidireccional. Sendo a principal diferença relativa à situação em que v_i é adicionado à lista de vizinhos de v, este adicionará também v removendo, se necessário, um outro vizinho v_j para que seja libertado espaço para v. Neste ponto, as *proxies* v e v_i trocam entre si as suas *cache digests* e tornam-se o vizinho mais recente um do outro. A *proxy* removida v_j deverá, por sua vez, eliminar da sua vizinhança v_i de forma a manter a simetria das relações estabelecidas neste tipo de organização. O processo de remoção de v_i por parte de v_j consiste em acrescentar o primeiro a uma lista de "remoções" até que seja encontrado um substituto para este.

Uma característica importante do algoritmo B-LRU é o facto de, aquando da adição de uma *proxy* à vizinhança de outra, ambas iniciarem de imediato a usufruir desta parceria sem a necessidade de a segunda ser forçada a "descobrir" a primeira através do seu próprio processo de exploração. Como desvantagens desta implementação ressaltam duas situações que poderão facilmente ocorrer. A primeira destas situações ocorre quando uma *proxy v* adiciona *w* à sua vizinhança por esta lhe trazer um valor de benefício acrescentado sendo por sua vez, *v* adicionado à vizinhança de *w*. O que sucede nesta situação é que *v* escolheu adicionar a nova *proxy* à sua vizinhança mas o contrário não se verifica pelo que, para adicionar *v* à vizinhança de *w* esta pode ter sido forçada a remover uma *proxy* que lhe trouxesse uma maior vantagem, diminuindo assim a sua eficiência.

Outra desvantagem deste algoritmo é o facto de gerar uma grande quantidade de tráfego pelo facto de as reconfigurações de vizinhança das *proxies* acontecerem com alguma frequência e envolverem uma vasta troca de mensagens entre estas por serem obrigadas a transmitirem aos seus vizinhos todas as actualizações. Esta troca de mensagens poderá ainda gerar um efeito "bola de neve" pois quando uma *proxy* altera a sua vizinhança isto também afectará pelo menos uma *proxy* que dela seja removida e que transmitirá este facto aos seus vizinhos, que poderão, por sua vez, ser afectados e assim sucessivamente.

Existe um algoritmo bidireccional (B-LFU) que permite a diminuição de alguns destes problemas. Tal como o seu equivalente unidireccional, este algoritmo mantém estatísticas acerca do benefício dos seus vizinhos e só quando um destes valores chega a um valor mínimo é despoletado o mecanismo de adaptação da vizinhança às novas necessidades da *proxy*. Quando uma *proxy* v_i é adicionada à vizinhança de v, esta reinicia as suas estatísticas o que faz com que uma próxima actualização só ocorra passadas x respostas positivas. Em caso de remoção, a *proxy* a ser removida (v_j) reinicia o número de respostas positivas por parte da *proxy* que a remove (v) mas não o número total de respostas da sua vizinhança (de forma a não adiar a sua reconfiguração). Para evitar um efeito de propagação observado em B-LRU, v_j não substitui imediatamente v com um novo vizinho mas aguarda até ao fim do período de renovação (quando o contador de respostas positivas fornecidas pela sua vizinhança atingir o valor pré-definido).

Mais do que uma tentativa de substituir os algoritmos unidireccionais, estas soluções bidireccionais pretendem ser uma alternativa válida aos primeiros sendo, no entanto, uma restrição destes. [Bakiras et al. 2005] defende que, como habitualmente, a aplicação prática destes algoritmos será mais vantajosa caso se encontre num ponto intermédio entre as duas soluções – uma solução híbrida possivelmente utilizando um algoritmo unidireccional para *proxies* dentro de uma mesma organização e outro bidireccional para ligação a *proxies* externas.

3.7 *Pre-fetching* dinâmico de dados baseado em padrões de acesso dos utilizadores

Em OLAP, o problema da selecção de vistas a materializar é um problema NP-completo [Gupta 1999] e tem sido bastante estudado. Algumas das abordagens conhecidas [Harinarayan et al. 1996], [Baralis et al. 1997], [Bauer & Lehner 2003] propõem que esta selecção seja realizada de forma estática antes de cada conjunto de *queries* e que os resultados desta acção sejam utilizados para responder a *queries* posteriores. Para contornar o problema que se levanta, com o facto de os padrões de utilização serem dinâmicos, foram desenvolvidas algumas técnicas que exploram este tipo de comportamento [Kotidis & Roussopoulos 2001], [Sapia 2000] e [Yao & An 2003].

Uma evolução da proposta de [Ramanchandran et al. 2005] em relação ao sistema *dynamat* [Kotidis & Roussopoulos 2001] consiste na implementação de um mecanismo que tem em consideração os padrões de utilização dos utilizadores bem como a sua estrutura dinâmica. No seguimento destes trabalhos surgiu a proposta do sistema PROMISE [Sapia 2000] que pretende prever com maior exactidão qual a estrutura de uma *query* baseada no padrão de exploração previsto para o utilizador em questão bem como na *query* actual emitida por este. Uma proposta que pretende responder aos dois problemas fundamentais da selecção dinâmica de vistas – a quantidade de informação necessária para traçar um bom perfil do utilizador e o momento certo para trazer essas vistas para memória (apenas quando são requisitadas ou tentando prever qual a *view* que será requisitada de seguida) – pode ser encontrada em [Ramanchandran et al. 2005].

3.7.1 Algoritmo de *pre-fetching* global

Os padrões de acesso de um utilizador prevêem a probabilidade de este realizar uma *query*, sabendo qual a *query* realizada previamente o que pode, intuitivamente, ser representado através de uma cadeia de Markov [Howard, 1960]. Um dos algoritmos que poderá ser utilizado nesta situação, e que visa a análise de informação estatística previamente armazenada relativamente à probabilidade de uma *query* surgir como sequência directa de outra já efectuada foi apresentado em [Ramanchandran et al. 2005]. Este algoritmo, através da informação sobre a probabilidade de

transição entre dois nós de uma Cadeia de Markov, prevê para uma determinada *query* efectuada, um conjunto de *queries* que poderão ser efectuadas num momento imediatamente posterior.

É, de seguida, apresentado o algoritmo referido, tal como definido em [Ramanchandran et al. 2005].

Seja:

- q uma queue de vértices (vistas da *lattice*)
- v a vista actual
- w[] o array que armazena a informação relativa ao benefício de cada um dos nodos baseado na probabilidade de o atingir a partir do novo actual
- eh[i][j] o array que armazena a probabilidade de ligação entre dois nodos (i e j), ou seja, a probabilidade de, sabendo que nos encontramos no nodo i, o próximo nodo ser§ o nodo j.
- views o conjunto de todas as vistas na *lattice*
- TS o espaço total existente para armazenar as vistas (na *view pool*)
- startNode o nodo que representa o início de uma sessão (ou novo contexto)
- prefetchedViews o conjunto de vistas retornado por este algoritmo
- size[v] o tamanho da vista v

```
Output = vistasPréMaterializadas

Adicionar nodoInicial a lista(q)
Inicializar w[nodoInicial]=1, w[j≠nodoInicial]=0
Enquanto(!vazio(q))
Inicio
       V = primeiro_elemento(q)
       Para todos(b pertencentes a pai(q))
       Inicio
              Se(!condicaoDeParagem)
                     Adicionar b a cauda(q)
              Para todos(c pertencentes a filhos(b))
                     W[b] = eh[c][b] * w[c]
       Fim
Fim
Ordenar descendentemente as vistas baseando-se em w[]
K=0, espaço = 0
```

Algoritmo 4 - Algoritmo de prefetching global
(proposto por [Ramanchandran et al. 2005])

Capítulo 4

A Técnica de *Caching* Proposta

Os sistemas OLAP, pela sua natureza, são sistemas que frequentemente permitem identificar, para cada utilizador ou grupo de utilizadores, a forma como estes acedem aos dados. Se tomarmos como exemplo um administrador de alto nível, muito provavelmente este irá aceder a informações relativas às vendas numa determinada loja ou região em detrimento de informações mais específicas relacionadas, por exemplo, com produtos. Explorando características como esta, é possível definir não só quais as áreas de incidência da análise de um determinado utilizador, mas também qual a sequência específica de pesquisas que este costuma realizar. Dessa forma é possível realizar previsões relativamente a qual a próxima *query* a ser dirigida ao servidor OLAP sabendo qual ou quais as *queries* efectuadas anteriormente.

Uma forma de adquirir este tipo de conhecimento acerca do comportamento de um utilizador passa por analisar os ficheiros de log relativos a sessões anteriores de consulta de dados. Através destes ficheiros é possível, através da aplicação de técnicas de mineração de dados, extrair o conhecimento pretendido, sendo que este será tanto mais assertivo quanto maior for a informação adquirida no passado. Uma das questões que se prende com a utilização deste tipo de conhecimento é a sua assertividade e a vantagem que advém da sua utilização. Ao contrário de outras técnicas de *caching*, esta abordagem não tem como objectivo primordial a diminuição da carga no servidor de dados, mas sim na melhoria no tempo de resposta aos pedidos efectuados pelo utilizador. A forma básica de actuação deste tipo de técnica tem em conta a última *query*

efectuada tentando inferir através desta (e de outra informação histórica de satisfação de *queries*) qual será o próximo pedido do utilizador. Caso seja possível realizar esta previsão, com um grau de certeza suficientemente elevado, a *query* será imediatamente lançada para que, quando o utilizador a realizar, esta esteja já armazenada em memória e seja apenas necessário fornecer os resultados ao utilizador. Outra abordagem possível a este problema passa por ter em conta, não só a última *query* efectuada, mas também um determinado número de *queries* anteriores a essa. Desta forma, mantendo informação relativa à sequência de pedidos efectuados pelo utilizador até ao momento é possível realizar previsões com maior grau de certeza embora seja também necessária uma maior quantidade de informação relativa aos comportamentos do utilizador que permita realizar todas estas previsões.

4.1 Previsão feita com base em regras de associação

A fase de previsão permite, através do recurso a técnicas de *Data Mining*, analisar os logs de *queries* existentes de forma a gerar regras de associação que serão utilizadas posteriormente para prever as *queries* efectuadas por um determinado utilizador num determinado momento. As regras geradas terão a forma:

$$A \rightarrow B\ (sup = \alpha; conf = \beta)$$
$$\alpha\ \in [0,1];\ \beta \in [0,1]$$

O que significa que, quando verificado o acontecimento A, dar-se-á B em (β*100)% dos casos. Pode ainda afirmar-se que o acontecimento A seguido de B se verifica em (α*100)% de todas as operações. Esta técnica quando aplicada a um cenário OLAP poderá resultar num conjunto de regras, como a seguinte:

$$A \rightarrow B\ (sup = 0.3; conf = 0.8)$$

Esta regra poderá ser interpretada como: "quando o utilizador efectuar a *query A*, em 80% dos casos efectuará a *query B* de seguida, sendo que de todas as *queries* efectuadas por ele numa sessão a sequência de *queries* A seguida de B ocorrerá em 30% das ocasiões".

Por outro lado, o antecedente (A) deste tipo de regras não terá forçosamente de ser um único acontecimento., o que significa que A poderá representar um conjunto de *queries*. Desta forma, o que será previsto não será somente baseado numa *query*, mas sim num caminho percorrido pelo utilizador desde o início da sessão. Se a regra for do tipo:

$$A_1, A_2, A_3 \rightarrow B$$

significará que o acontecimento B pode ser previsto como consequência dos acontecimentos A_1, A_2 e A_3, com uma dada probabilidade. O tipo de previsão que estas regras permitem fazer vai muito além da simples previsão "passo-a-passo", como seja o de prever qual a próxima *query* a ser efectuada pelo utilizador. Desta forma, é possível assim prever com um maior grau de antecipação quais as *queries* que irão ser efectuadas pelo utilizador até ao final da sessão, sabendo-se o rumo que este seguiu na sua análise até ao momento. É ainda possível que os antecedentes das regras obtidas não sejam necessariamente conjuntos de *queries* efectuadas, mas sim outro tipo de condições, como sejam certas alturas do dia ou da semana. Se explorarmos estas possibilidades, associando-as desde já ao ambiente OLAP, podemos definir regras que nos indiquem a frequência com que um utilizador específico realiza uma consulta, previsivelmente à mesma hora, num determinado dia da semana. Podemos pensar, por exemplo, no cenário de um Administrador que, todas as Sextas-Feiras à tarde, antes de sair do local de trabalho, verifica se tudo se encontra numa situação regular no que diz respeito aos indicadores da empresa. Previsivelmente, neste tipo de cenário, os indicadores da empresa analisados por si serão sensivelmente os mesmos e as consequências desta análise pertencerão a um conjunto possivelmente restrito de pesquisas.

A exploração deste tipo de características do sistema de previsão que foi criado revela algumas potencialidades interessantes que poderão ser de grande utilidade na melhoria do desempenho de um sistema OLAP, mas, por outro lado, revela também questões que deverão ser respondidas após deliberação para cada caso específico. Algumas destas questões são, entre outras, as seguintes:

- Qual o número de pesquisas do utilizador, dentro de uma mesma sessão, que deverão ser tidos em conta no momento da previsão?

- Deverão ser aceites todas as regras como válidas ou, por outro lado, deverão ser definidos valores mínimos de suporte e confiança a partir dos quais o esforço de materialização das vistas correspondentes é recompensado?
- Deveremos materializar imediatamente todas as *queries* que se prevêem vir a ser necessárias ou apenas um subconjunto destas.

4.2 Refrescamento de padrões de acesso

Quando são analisados os ficheiros de log com o objectivo de extrair padrões de navegação dos utilizadores, este conhecimento é armazenado para que o seu acesso seja simples e rápido quando se pretender realizar previsões baseadas nessa informação. Sabendo que as técnicas de *Data Mining* aplicadas à informação contida nos ficheiros de log conseguem gerar previsões tanto mais correctas quanto maior for a informação relativa a acções passadas, surge uma questão que se prende com o período de revalidação dos dados. Se, por um lado, os ficheiros de log forem analisados todos os dias poder-se-á estar a levar em conta factores que não representam um comportamento global, mas sim apenas esporádico. Por outro lado, se a reavaliação dos padrões de utilização surgir em períodos demasiado distanciados no tempo, poder-se-á correr o risco de se estar a realizar previsões sobre uma realidade que apesar de válida no passado já não o é no presente. Adicionalmente, uma outra problemática emerge, esta agora relacionada com qual a informação a ser utilizada quando se pretender obter os padrões de acesso de um determinado utilizador. Assim, dever-se-á ter em conta toda a informação disponível, relativamente aos acessos desse utilizador ou dever-se-á apenas ter em conta, por exemplo, a informação da semana imediatamente anterior à data da análise?

A tomada de posição relativamente a esta problemática condicionará os padrões de acesso gerados, isto é, caso seja utilizada toda a informação existente relativamente a este utilizador poder-se-ão estar a descurar hábitos de utilização recentemente adquiridos, uma vez que estes ficarão "diluídos" entre toda a informação analisada. Caso seja utilizada apenas a informação relativa a um passado recente, é possível que estejam a ser extraídos padrões de utilização que representem apenas necessidades demasiado recentes e não o comportamento a longo prazo do utilizador em causa.

Este tipo de decisões de implementação, nomeadamente saber qual a informação a ter em conta bem como saber durante quanto tempo se deverá considerar essa informação como válida, influenciará, quer positiva quer negativamente, todo o desempenho do sistema em causa, pelo que deverão apenas ser tomadas depois de uma cuidada análise do problema específico em causa.

4.3 Número de movimentos do utilizador a ter em conta

Na previsão de padrões de acesso aos dados por parte dos utilizadores, frequentemente é necessário tomar decisões relativamente aos critérios desta escolha. De entre estes destaca-se o número de movimentos do utilizador, dentro da mesma sessão, que deverão ser tidos em conta no momento da previsão.

Uma das formas de realizar esta previsão consiste em representar as sequências de *queries* que se prevê virem a ser realizadas pelo utilizador numa Cadeia de *Markov* [Howard, 1960]. Esta estrutura, semelhante a um grafo em que (neste caso específico) os arcos representam a probabilidade de transição entre os diferentes estados (*queries* OLAP) facilita de grande forma uma análise visual do problema, essencial em termos de consciencialização das características específicas deste.

Por outras palavras, é possível realizar previsão sobre uma cadeia de *Markov* apenas por observação e análise desta. É possível, por exemplo, afirmar que (no caso do exemplo apresentado na Figura 3), se o utilizador se encontrar no estado S_1, existem 90% de probabilidades de este realizar a *querie* S_3. Este tipo de análise resulta da mera observação dos dados que são extraídos dos ficheiros de log correspondentes a utilizações prévias do sistema por este utilizador.

É, no entanto, possível efectuar uma análise mais cuidada dos dados (através de algoritmos de *data mining* que gerem regras de associação entre os dados) tendo em conta mais do que um passo na interacção do utilizador com os dados. Desta forma é possível afirmar que, por exemplo, quando um utilizador atinge o estado S_2 vindo de S_1, a probabilidade de o próximo estado atingido ser S_8 não é de 10% como sugerido pelo gráfico, mas sim de 30%. A principal diferença deste tipo de abordagem é o facto de se pretender retirar dos modelos, não só informação relativa às acções

do utilizador para o futuro, mas também, basear esta informação no princípio que defende que as acções de uma pessoa no futuro dependem daquilo que foi efectuado no passado. É, no entanto, intratável materializar todas as vistas correspondentes às *queries* efectuadas pelos utilizadores.

A forma proposta de minorar este problema passa pela utilização de técnicas de *Data Mining* que gerem as regras de associação entre os dados. A técnica de geração tem como objectivo encontrar correlações em dados aparentemente não relacionados. Um exemplo clássico da aplicação de regras de associação vem da área do retalho em que se pretende saber, frequentemente, quais os produtos que os seus clientes compram em conjugação uns com os outros, isto é, quem comprar o produto A, terá uma determinada probabilidade de adquirir em conjunto o produto B. No caso específico da previsão que pretendemos realizar, este é em tudo análoga à anteriormente apresentada no sentido em que pretendemos saber qual a probabilidade de, sabendo que um utilizador realizou a *query* A, este vir a realizar a *query* B de seguida. Sendo ainda mais específico, o que pretendemos saber na realidade é, sabendo que o utilizador realizou a *query* A, quais as *queries* que este poderá vir a realizar de seguida e quais as suas probabilidades associadas.

O conjunto de regras de associação geradas (a partir da análise dos logs do servidor OLAP) possui no entanto um grande problema – o número de pré-materializações que seria necessário efectuar para servir todas as possibilidades de questões do utilizador, directamente a partir da *cache.* Desta forma, a técnica proposta prevê um passo extra que consiste em definir um valor que distinga as regras de associação que possuem um grau de previsão mais acertado das restantes.

Cada regra de associação gerada por um algoritmo de *Data Mining* possui a si associados dois valores – a confiança e o suporte – e é precisamente através destes que será feita a selecção de quais as regras que deverão ser tidas em conta no momento da previsão e quais as que deverão ser ignoradas.
As regras resultantes de todo este processo e, consequente, a cadeia de *Markov* gerada, incluem já todas estas considerações, visto que, pela própria definição de cadeia de *Markov*, a transição entre dois nodos A e B depende apenas de A e não de quais as *queries* realizadas anteriormente.

4.4 Definição do suporte e confiança ideais

Quando tratamos regras de associação, estas possuem métricas referentes à sua importância e grau e fiabilidade. Estas métricas, confiança e suporte, são de vital importância na distinção de regras importantes daquelas que, apesar de válidas, foram obtidas a partir de acontecimentos menos significativos, ou menos frequentes nos dados estudados. A métrica de suporte indica-nos o número de ocorrências de acontecimentos que confirmam a regra, estudados no *dataset*. Por seu lado, a confiança apresenta-se como uma medida que indica a força da regra, isto é, uma probabilidade condicional de, dado o antecedente da regra, o consequente se vir a verificar. A forma de cálculo da confiança de uma regra é dada pela seguinte regra:

$$conf\,(A \rightarrow B) = \sup(AC)\,/\sup(A)$$

isto é, a confiança da regra é dada pelo quociente entre o número de vezes que os dois acontecimentos ocorreram de forma consecutiva no *dataset* e o número de vezes que ocorreu o acontecimento correspondente ao antecedente da regra independentemente do contexto em que isso sucedeu.

Um problema da geração deste tipo de regras é a definição dos valores ao suporte e confiança mínimos a partir dos quais as regras devem ser aceites como válidas e benéficas. Estes valores variam de caso para caso e deverão ser definidos após o estudo do cenário específico em causa. A correcta definição deste tipo de valores condiciona o futuro comportamento do sistema, tanto pelo trabalho que será dispendido se se permitir a geração de demasiadas regras como pela perda de regras importantes se os valores de suporte e confiança forem demasiado exigentes para o cenário em causa.

4.5 Exemplo de cadeia de Markov

Em capítulos anteriores deste trabalho foi referido que as regras de associação poderiam ser representadas sob a forma de uma Cadeia de *Markov* como forma de simplificar a compreensão do significado destas. Na figura que se segue é apresentada a Cadeia de *Markov* correspondente a um conjunto de regras que servirão de base ao primeiro conjunto de testes efectuado.

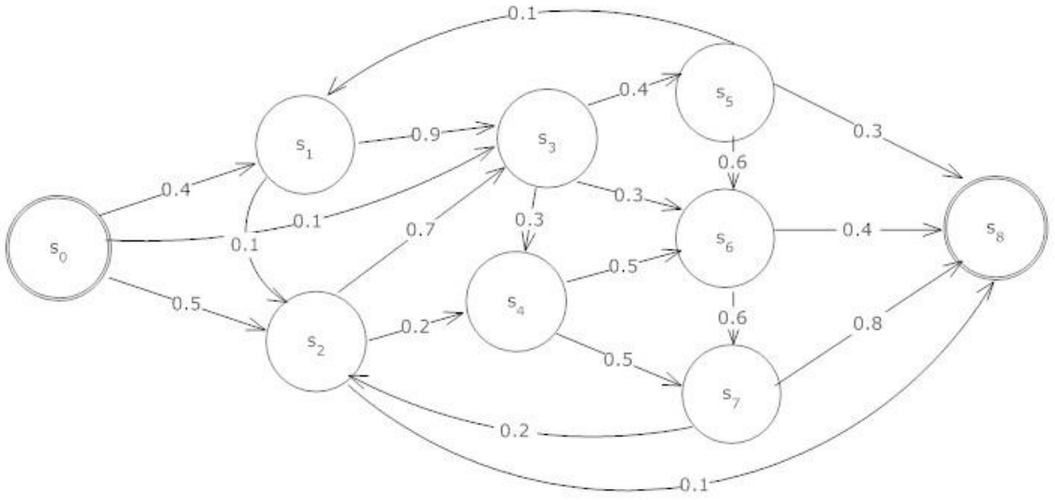


Figura 3 - Exemplo de cadeia de *Markov*

A Figura 3 apresenta a transformação de um conjunto de regras de associação numa representação que permita uma análise mais simples destas. Esta representação consiste em, primeiramente, identificar um estado inicial S_0 (início de sessão por parte do utilizador) e um estado final S_8 (fim de sessão). Numa segunda fase, as regras existentes dão origem aos restantes elementos da Cadeia de *Markov*. Analisando a cadeia apresentada, é possível saber que, após ser realizada a *query* S_1 que será em 40% dos casos a primeira *query* a ser efectuada (este valor é indicado na legenda da transição de S_0 para S_1), será em 90% dos casos realizada a *query* S_3. A regra que deu origem a esta transição seria do tipo $S_1 \rightarrow S_3$ $(sup = \ldots; conf = 0.9)$. De forma análoga decorre a restante composição da Cadeia de Markov que representa o conjunto de *queries* a analisar.

4.6 Que vistas materializar?

Uma forma de abordar o problema de *caching* preditivo passa por, após ser iniciada uma sessão por parte de um utilizador verificar se é possível prever quais as *queries* que serão efectuadas por este. Caso seja possível efectuar essa previsão pode optar-se por uma de duas abordagens:

1. Pré-materializar as vistas que se prevê virem a ser necessárias na sessão iniciada.

2. Efectuar apenas o pedido que se prevê vir a ser o primeiro a ser efectuado pelo utilizador e, apenas quando se confirmar que esta previsão foi acertada, efectuar o que se prevê ser o próximo pedido e assim sucessivamente.

Destacam-se, de imediato, diferenças fulcrais nos requisitos, desempenho e implementação destas duas políticas de *caching* sendo que a primeira se enquadra numa temática de *iceberg cubing* e a segunda se aproxima mais de uma técnica de *fetching on-demand* como a realizada pelos sistemas operativos actuais. As diferenças entre ambas as abordagens são numerosas e incluem desde o espaço necessário, às consequências de uma previsão não acertada. A opção por materializar as vistas que se prevê virem a ser utilizadas é uma técnica bastante eficiente no sentido em que quando posteriormente estas forem necessárias, todo o processamento a elas associado foi já realizado. Desta forma é possível balancear a carga no servidor materializando as vistas (sempre que possível) nos momentos em que o servidor estiver com uma menor carga de processamento.

Com este balanceamento de carga consegue-se obter uma solução de compromisso para o tempo de resposta tanto aos clientes cujas vistas não podem ser materializadas *a priori* como para os restantes. No que diz respeito ao espaço de memória necessário para a implementação destes sistemas, a opção por materializar apenas a vista correspondente à próxima *querie* a ser efectuada será muito menos exigente. Neste caso apenas é necessário, em cada momento, manter em memória uma, possivelmente duas, vistas materializadas. No caso de pretendermos ter todas as vistas simultaneamente em memória, necessitaremos de uma grande quantidade de espaço disponível pois, em OLAP, os resultados das *queries* efectuadas poderão reunir grandes quantidades de informação ocupando assim muito espaço. Se se pretender implementar um sistema de pré-materialização de vistas há que ter em conta factores como qual a política de gestão de cache que vai ser utilizada de forma a manter em cache a informação pretendida caso não exista espaço disponível para toda a informação na *pool* de vistas. A decisão por qual o gestor de cache a utilizar define qual a política de inserção/remoção de vistas na *pool* o que é de fulcral importância no desempenho global dos sistema. Por outro lado, se pretendermos apenas manter em memória a *query* indicada pelo sistema de previsão como sendo a próxima a ser efectuada (não necessariamente apenas essa mas com critérios bastante restritivos), não é necessária essa gestão pois o espaço em memória existente será sempre, previsivelmente, suficiente para o que se

pretende. Desta forma, ao errar uma previsão, o esforço "desperdiçado" na pré-materialização da vista correspondente resume-se a apenas uma vista que, apesar de poder ser algo de significativo, será certamente muito menor do que o associado à materialização de um grande conjunto de vistas.

Analisando outra vertente dos erros de previsão constata-se que, se por um lado errar uma previsão utilizando a técnica mais eficiente em termos de memória se revela menos penalizante em termos de trabalho desperdiçado, por outro lado isso implica que toda a estrutura de previsão deverá ser reavaliada. Isto acontece pois, neste tipo de técnica, a previsão é realizada seguindo um "caminho" num grafo orientado onde uma falha na previsão poderá resultar numa localização do grafo onde não existe possível previsão futura. Se for utilizada a técnica de pré-materialização de diversas vistas em simultâneo permite-se que um erro, apesar de desperdiçar o trabalho de construir uma das vistas, não seja tão grave em previsões futuras. Isto deve-se ao facto de a pesquisa feita pelo utilizador, apesar de não ser a prevista para aquele instante poder estar materializada por ter sido prevista para outra fase da consulta.

4.7 Fases da técnica desenvolvida

A técnica desenvolvida tendo em vista a diminuição do tempo de resposta a uma *query* efectuada pelo utilizador envolve várias fases. Num primeiro momento, é necessário extrair do Data Warehouse, mais especificamente do servidor OLAP, um conjunto de ficheiros de log que permitam conhecer a sequência de *queries* efectuadas num determinado período e por um determinado utilizador. De seguida, estes ficheiros são analisados e são aplicadas, sobre as informações neles presentes, técnicas de *data mining* cujo resultado será um conjunto de regras de associação. As regras de associação obtidas representam as sequências de *queries* encontradas independentemente da frequência com que estas se verificaram. Como forma de diminuir o número de regras existentes, estas são posteriormente analisadas mantendo apenas aquelas que revelem um grau de certeza suficientemente elevado. Este valor, que ditará a aceitação ou remoção de uma regra da lista que será utilizada para efectuar as previsões, é parametrizável e deverá ser ajustado de acordo com as necessidades e características da situação específica em causa. Numa fase final, as regras obtidas após a fase de simplificação serão inseridas no servidor de cache OLAP e utilizadas para efectuar posteriores previsões.

Na Figura 4 encontram-se representadas as fases da técnica desenvolvida previamente apresentadas.

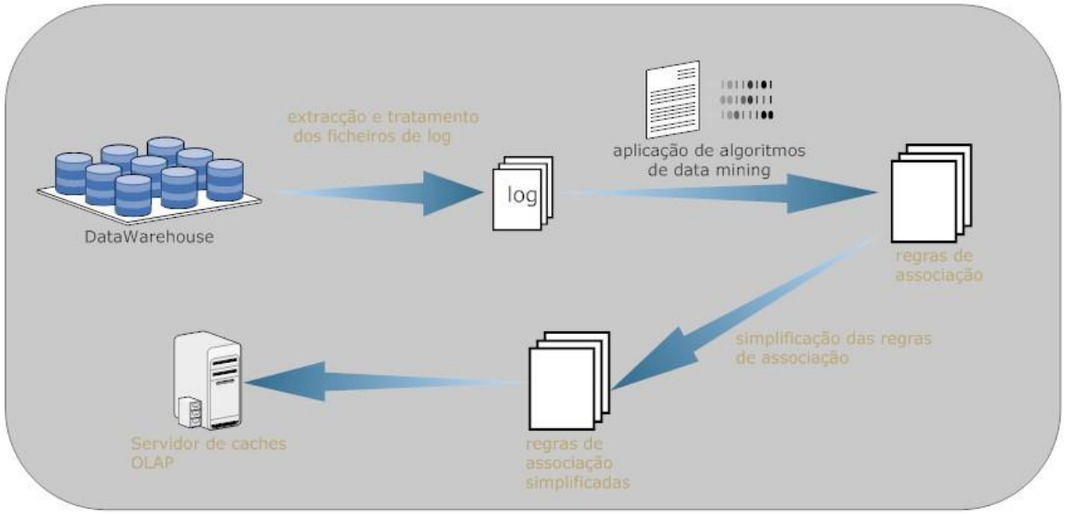


Figura 4 - Representação das diversas fases da técnica desenvolvida

Capítulo 5

Os Casos de Estudo

A técnica desenvolvida e anteriormente explicada, consiste em várias fases. De entre todas estas fases dá-se neste capítulo particular atenção à que envolve a simplificação do conjunto de regras a serem tidas em conta no momento de efectuar a previsão acerca de qual a próxima *query* a ser executada. Dito por outras palavras, é nesta fase que são rejeitadas as regras de associação que não satisfaçam uma condição de assertividade definida como mínima para o problema específico. A importância desta fase da técnica deve-se ao facto de ser nesta que vão ser escolhidas as ferramentas a serem fornecidas ao sistema que posteriormente fará todas as previsões. Fica então claro que se ocorrerem erros neste ponto do processo, estes serão propagados para fases posteriores afectando futuramente o desempenho global do Sistema.

Os casos de estudo foram separados em dois conjuntos distintos. Um primeiro conjunto, que possui um número reduzido de regras de associação a serem simplificadas, pretende, antes de mais, clarificar alguns conceitos relacionados com a simplificação das regras de associação. Além disso, outro objectivo deste primeiro conjunto de testes é o de exemplificar como se processa a simplificação das regras de associação bem como demonstrar a importância desta simplificação e as consequências que esta acção terá sobre a totalidade do sistema. Numa segunda fase, é apresentado um conjunto substancialmente maior de regras de associação, retirados de um cenário de utilização real de um servidor OLAP sobre as quais vão ser aplicadas as técnicas de simplificação previamente referidas. Neste segundo conjunto de testes, devido à dimensão dos

dados a ter em conta, são mais notórias as diferenças de parametrização dos valores a ter em conta para classificação das regras como aceites ou não aceites.

5.1 Simplificação do número de vistas a materializar

Quando se pretende realizar *prefetching* recorrendo a um conjunto de regras previamente geradas, é necessário que seja definida uma estratégia que diferencie as regras que deverão ser consideradas como boas, das restantes, que deverão ser ignoradas em termos de processo de previsão. O problema que se põe com este tipo de decisão foi já explicado anteriormente e será, nesta fase, descrito de uma forma mais prática.

Para isso serão analisados alguns conjuntos de regras de associação aos quais serão aplicadas diferentes técnicas de simplificação e testados os desempenhos do sistema decorrentes das diferentes técnicas aplicadas. É espectável que os resultados obtidos permitam retirar algumas conclusões sobre a implicação que estas simplificações terão sobre o desempenho global do sistema embora não seja previsível que se extraiam regras genéricas adaptáveis a todos os sistemas deste género.

Com isto pretende-se que seja mais evidente a dificuldade de identificar soluções óptimas e facilitar a observação dos compromissos necessários a uma decisão que afectará o desempenho global do sistema de uma forma bastante profunda.

5.1.1 Identificação do caminho mais provável

Uma das possibilidades que existe no sentido de reduzir o número de *queries* que deverão ser materializadas previamente à consulta por parte do utilizar baseia-se na simples observação da probabilidade com que estas *queries* ocorrerão. Desta forma, torna-se simples identificar qual o conjunto de *queries* que, partindo da ligação de uma nova sessão, permitirá mais previsivelmente atingir o estado final (final de sessão). Para isto, em cada estado da cadeia, deverá observar-se qual será o estado com maior probabilidade de ser o estado seguinte, até atingir o estado final.

O conjunto de estados percorridos neste processo designa-se por caminho principal, ou mais provável, a ser seguido pelo utilizador na sua próxima interacção com o sistema multidimensional de dados.

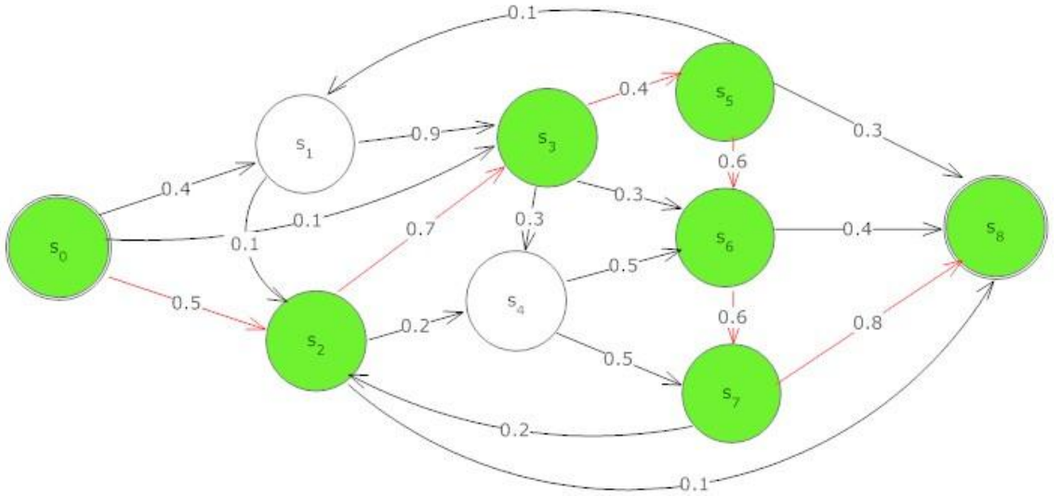


Figura 5 - Identificação do caminho mais provável numa cadeia de *Markov*

No caso específico da cadeia de *Markov* apresentada na Figura 5, a identificação do caminho principal resultaria na materialização dos estados $\{S_2, S_3, S_5, S_6, S_7\}$, uma vez que os estados S_0 e S_8 correspondem, respectivamente, às acções de início e final de sessão.

5.1.2 Identificação de valores mínimos de confiança a considerar

Como referido anteriormente, a simplificação do número de vistas a materializar, não é necessariamente definida pelo caminho mais provável a ser seguido pelo utilizador. É possível, também, definir-se um valor mínimo para a confiança da regra de associação a ser aceite, servindo este critério como forma de validação das regras em termos da sua credibilidade.

A definição de diferentes valores de confiança mínima (minconf) influenciará directamente o número de regras a serem consideradas como verdadeiras, ou úteis. Quanto maior for o valor tido como mínimo para a confiança de uma regra, menor será o número de regras aceites, e vice-versa.

Após a atribuição de um valor a esta variável, é possível diminuir o número de regras a serem aceites. Estas regras deverão ser aquelas que, na representação visual da nova Cadeia de *Markov*,

correspondam a nodos que não possuem arcos de entrada - arcos cujo sentido aponta para o nodo em questão.

Os valores de confiança mínima escolhidos para os testes realizados foram de 0.3, 0.4 e 0.5. A escolha destes valores deveu-se ao facto de as probabilidades de transição entre os estados da Cadeia de *Markov* serem bastante abrangentes (existem valores entre 0.1 e 0.9) e ainda por os valores escolhidos não serem demasiado extremados dentro das possibilidades existentes. Se, por um lado, simplificar a Cadeia de *Markov* utilizando um valor de confiança mínima de 0.1 resultaria numa simplificação pouco relevante, a escolha do valor 0.9 a ser aceite como valor de referência resultaria numa simplificação demasiado abusiva desta Cadeia (o número de nodos resultantes seriam demasiadamente pequeno para os resultados dos testes serem tidos como fiáveis).
Desta forma, e após deliberação, foram escolhidos os valores de confiança mínima de 0.3, 0.4 e 0.5 tal como referido anteriormente.
De seguida apresentam-se os resultados da simplificação da Cadeia de *Markov* apresentada na figura 4, segundo os diversos valores de confiança mínima definidos e os respectivos resultados no que diz respeito à assertividade das previsões efectuadas numa simulação de utilização em cenário real.

Minconf = 0.3

A imagem que se segue (Figura 6) representa a utilização de um valor de confiança mínimo de 0.3 unidades (confiança mínima de 30%). Desta forma, foi já possível remover arcos que possuíssem uma probabilidade associada menor do que o valor requerido. É possível observar que, por exemplo, o arco correspondente à transição entre S_2 e S_8 não foi aceite. Por outro lado, se a estratégia passar, tal como foi descrito anteriormente, por materializar todas as vistas ainda presentes na Cadeia, não será possível com este valor de confiança mínima remover qualquer vista. Devido a este facto, pode considerar-se que apesar da definição de um valor mais restritivo da confiança mínima o benefício será nulo, pois o conjunto de vistas materializadas será precisamente o mesmo.

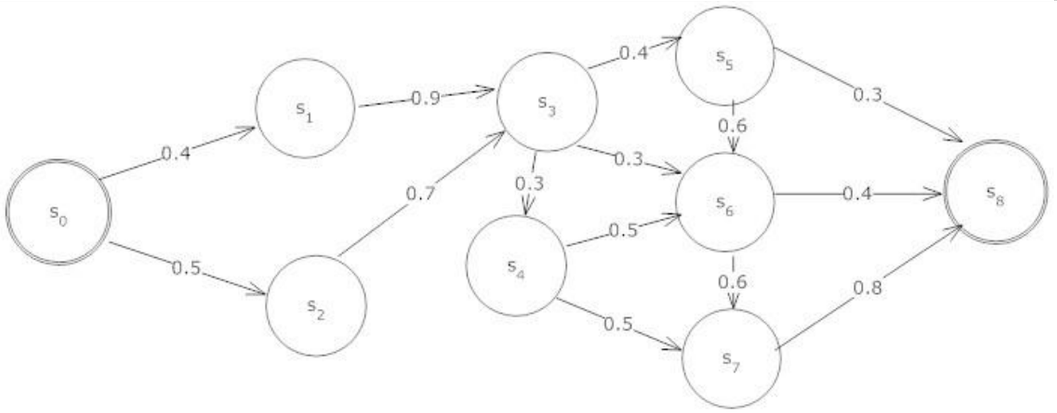


Figura 6 - Exemplo de cadeia de *Markov* simplificada (minconf = 0.3)

Minconf = 0.4

No caso que se segue (**Error! Reference source not found.**) com a definição de um valor mínimo de confiança igual a 0.4, é possível simplificar o conjunto de regras a serem consideradas. Como se pode observar, o nodo que cumpre o requisito de não possuir arcos de entrada é o nodo correspondente à querie S_4. Desta forma interpreta-se que, a probabilidade de a *query* S_4 ser realizada é insuficiente para justificar o custo computacional de materializar a vista correspondente. Neste ponto, impõe-se a discussão de outra problemática associada a esta simplificação do modelo de previsão. Quando se remove um nodo do grafo deve-se, logicamente, remover-se também aqueles nodos que sejam posteriores (temporalmente) em relação a este. Neste caso específico, estes seriam os nodos S_5 e S_6, mas que se mantêm no modelo pois, apesar de não serem previsivelmente atingidos a partir do estado S_4, poderão sê-lo a partir do estado S_5 (continuam a possuir arcos de entrada).

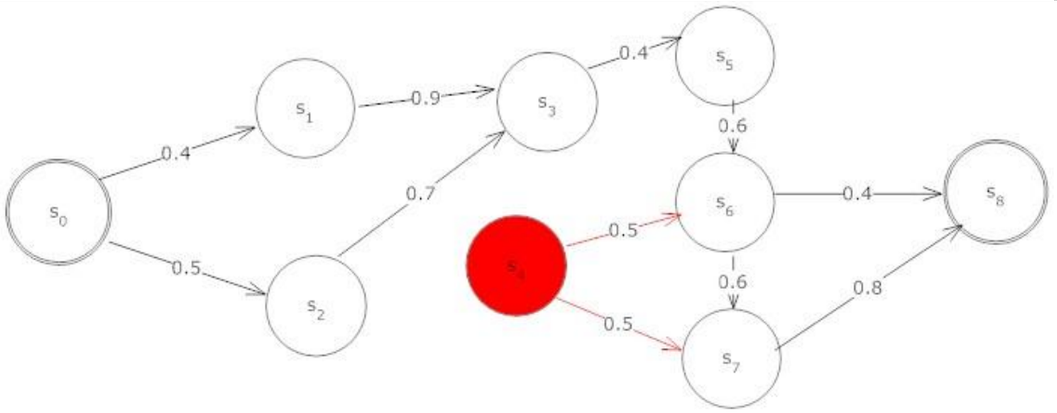


Figura 7 - Exemplo de cadeia de *Markov* simplificada (minconf = 0.4)

Minconf = 0.5

Neste caso apresenta-se a simplificação do modelo de previsão anterior mas com uma exigência ao nível da confiança mínima de 50%. Seguindo a lógica exposta anteriormente, são removidos do modelo os nodos $S_{1,}$ S_4 e S_5 por não possuírem, após a primeira etapa da simplificação, arcos de entrada a si associados. Numa segunda fase da análise, podemos observar que o nodo S_6, após remoção de S_5 perde a única ligação que possuía e que previa a sua ocorrência. Desta forma, é necessário remover este nodo (S_6). Da mesma forma, remover-se-á de seguida o nodo S_7. Como é possível observar pela figura que se segue, a restrição do valor de confiança mínima aceitável a 0.5 unidades resulta numa simplificação bastante considerável do modelo de previsão gerado. Neste caso, apenas serão materializadas as vistas S_0 (estado inicial), S_2, S_3 e S_8 (estado final) – ver Figura 8.

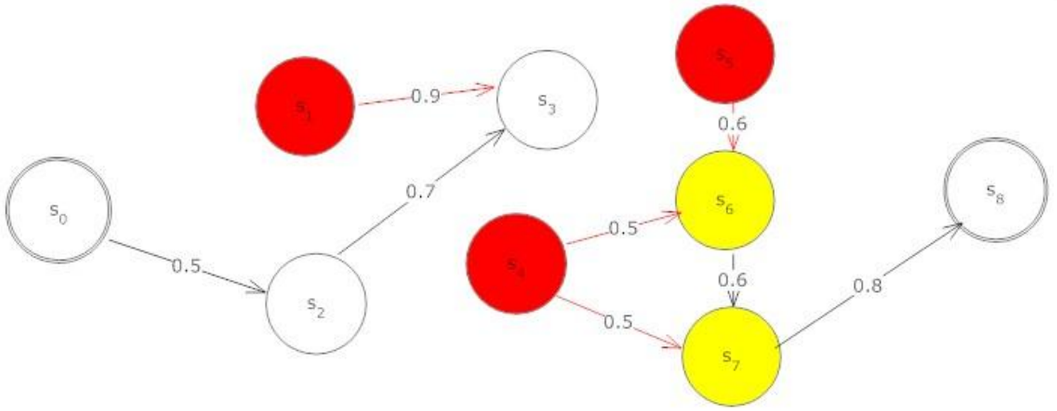


Figura 8 - Exemplo de cadeia de *Markov* simplificada (minconf = 0.5)

5.1.3 Testes de desempenho

Na impossibilidade de realizar testes de desempenho num ambiente real, desenvolveram-se um conjunto de testes com o objectivo de testar algumas características do desempenho do sistema de previsão descrito.

Pelo facto de não ser possível realizar estes testes em ambiente real não foi também possível realizar testes a nível de desempenho das soluções propostas. Foi, ainda assim, possível realizar testes que nos fornecessem indicadores quanto à qualidade de previsão do sistema.

Para estes testes foram criados alguns *datasets* dinâmicos, recorrendo a um factor de aleatoriedade que, a partir do grafo apresentado e recorrendo a uma técnica de *reverse engineering*, extrapolaram valores que correspondessem a sessões válidas de interacção com a base de dados. Um problema que se põe com este tipo de abordagem é a forma de criação dos *datasets* sem manipular, de forma intencional, ou não, os resultados obtidos. Neste sentido, a forma de criação dos *datasets* baseou-se no grafo apresentado anteriormente e, de acordo com as probabilidades das transições apresentadas, foram gerados os valores de teste. Posteriormente, a cada passo da interacção, foi gerado um valor aleatório que identificava qual o próximo nodo. Desta forma, sabendo qual o conjunto de vistas que se encontravam pré-materializadas, foi possível medir a percentagem de previsões acertadas (*hit ratio*) e consequente *miss ratio*. Os esquemas de simplificação testados foram os apresentados anteriormente com os valores de

confiança mínima a variar entre os valores 0.3, 0.4 e 0.5. Foi ainda testado o esquema de simplificação que se baseia na materialização das vistas correspondentes ao caminho mais previsível a seguir pelo utilizador. De seguida, para cada organização do modelo de previsão, foram também gerados cinco diferentes conjuntos de teste, com o objectivo de tornar irrelevantes eventuais eventos que não estivessem em conformidade com o modelo apresentado.

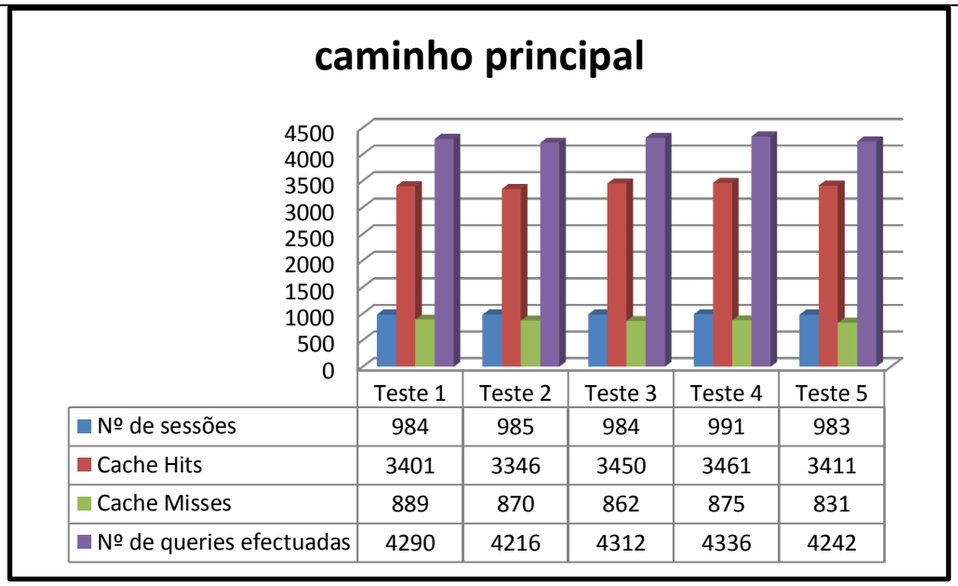
De cada conjunto de testes resultaram os seguintes indicadores:

- Número de sessões, que indica o número de sessões simuladas (travessias do grafo desde o estado inicial ao estado final).
- *Cache hits*, que revela o número de *queries* requisitadas que já existiam na *pool* de vistas resultantes do processo de previsão.
- *Cache misses*, que representa o número de *queries* requisitadas que necessitaram de materialização no momento em que foi efectuada a *query*.
- Número de *queries* efectuadas, que corresponde ao número total de *queries* efectuadas (corresponde ao somatório do número de *cache hits* e *cache misses*).

De seguida, apresentam-se os resultados obtidos nos diversos testes efectuados.

Caminho principal

Na Figura 9 apresentam-se os resultados dos testes correspondentes à simplificação denominada por caminho principal. Nesta simplificação identificam-se os nodos correspondentes, em cada passo, à maior probabilidade de transição identificando, assim, a sequência mais provável de consultas a serem realizadas pelo utilizador em causa.



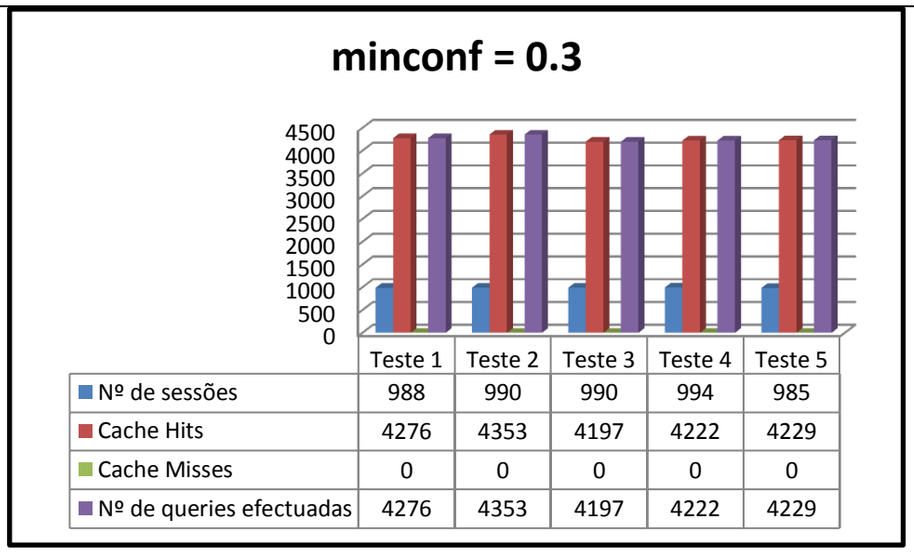
	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
Nº de sessões	984	985	984	991	983
Cache Hits	3401	3346	3450	3461	3411
Cache Misses	889	870	862	875	831
Nº de queries efectuadas	4290	4216	4312	4336	4242

Figura 9 - Resultados dos testes (caminho principal)

Para estes testes foi identificada, conforme apresentado anteriormente, a sequência de *queries* {S_0, S_2, S_3, S_5, S_6, S_7, S_8}. Deste conjunto, todas as *queries* foram materializadas à excepção de S_0 e S_8, já que correspondem aos estados inicial (início de sessão) e final (fim de sessão), respectivamente. De todo o conjunto de *queries* possíveis, apenas não foram pré-materializadas as *queries* correspondentes aos estados S_1 e S_4 por não satisfazerem o critério referido anteriormente. Nos resultados apresentados é possível observar que o número de *cache hits* é bastante superior ao de *cache misses* o que indica uma boa taxa de previsão, ainda que, para isto, contribua o facto de a maioria das vistas ter sido pré-materializada (cerca de 71% das vistas). Neste sentido, podemos inferir que, apesar de a carga computacional associada à materialização das vistas ser alta, esta materialização será, provavelmente, compensada pela alta percentagem de *queries* respondidas directamente através da cache (aproximadamente 80%).

Minconf = 0.3

De seguida (Figura 10) apresentam-se os resultados relativos aos testes efectuados com uma definição do valor mínimo de confiança de 30%. Neste caso, a simplificação das regras não permitiu excluir nenhuma vista pelo que todas elas serão pré-materializadas.



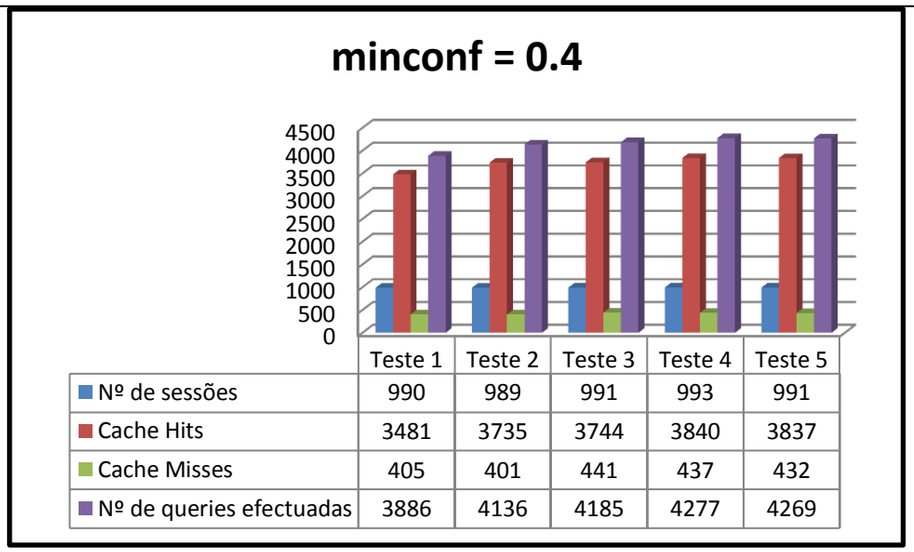
	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
Nº de sessões	988	990	990	994	985
Cache Hits	4276	4353	4197	4222	4229
Cache Misses	0	0	0	0	0
Nº de queries efectuadas	4276	4353	4197	4222	4229

Figura 10 - Resultados dos testes (minconf = 0.3)

Ao observarmos os resultados recolhidos nos testes efectuados é de salientar a percentagem de pedidos que foram servidos a partir da cache (100%). Isto acontece pelo simples facto de todas as vistas terem sido pré-materializadas - conforme explicado anteriormente. Desta forma, ainda que com benefícios evidentes em termos de tempo de resposta aos pedidos realizados, é necessário ressalvar o facto de a carga de materialização ser bastante elevada, uma vez que é necessário fazer a materialização de todas as vistas resultantes do processo de previsão.

Minconf = 0.4

Os resultados que se apresentam de seguida (Figura 11) correspondem aos testes efectuados utilizando o valor de 40% para a confiança mínima admissível para uma regra de previsão. As *queries* materializadas para este conjunto de testes foram as seguintes: S_1, S_2, S_3, S_5, S_6 e S_7, ficando apenas por materializar a *query* S_4 que, tal como descrito anteriormente, possuía regras a si associadas que não satisfaziam os critério mínimos de confiança, sendo assim retiradas do modelo de previsão.



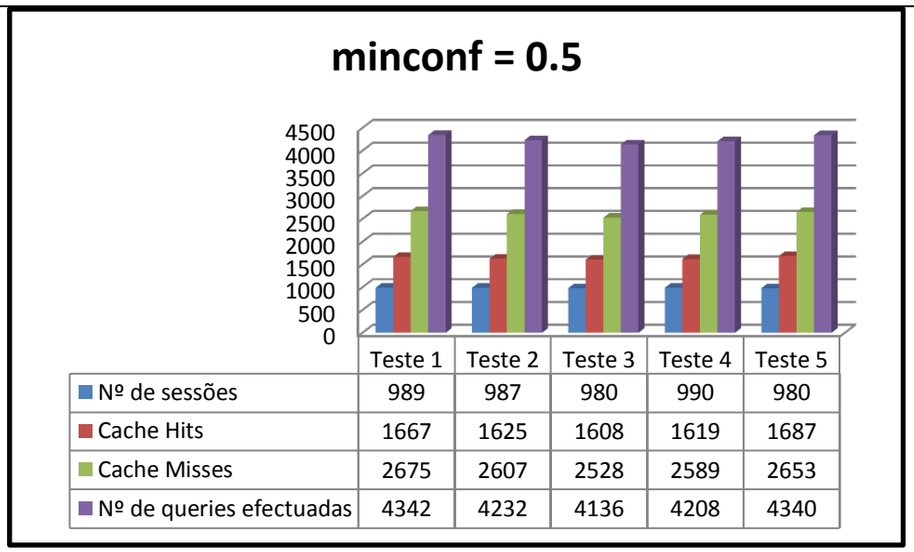
	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
Nº de sessões	990	989	991	993	991
Cache Hits	3481	3735	3744	3840	3837
Cache Misses	405	401	441	437	432
Nº de queries efectuadas	3886	4136	4185	4277	4269

Figura 11 - Resultados dos testes (minconf = 0.4)

Dos resultados apresentados destaca-se o facto de a percentagem de *cache hits* ser bastante elevada (aprox. 90%). De ressalvar que este tipo de resultados advém do facto de a taxa de materialização das vistas ser, também ela, muito elevada (cerca de 86%).

Minconf = 0.5

Por último, apresentam-se os resultados dos testes efectuados utilizando um valor de confiança mínima de 50%, isto é, só as regras suportadas por esse valor de confiança é que serão aceites. Neste cenário, a simplificação resultante das restrições impostas revela já uma diminuição considerável no número de vistas a materializar na cache. Como referido anteriormente, o conjunto de vistas materializadas corresponde apenas às *queries* S_2 e S_3, sendo todos os restantes excluídos aquando do processo de simplificação do conjunto de regras de associação geradas no processo de previsão.



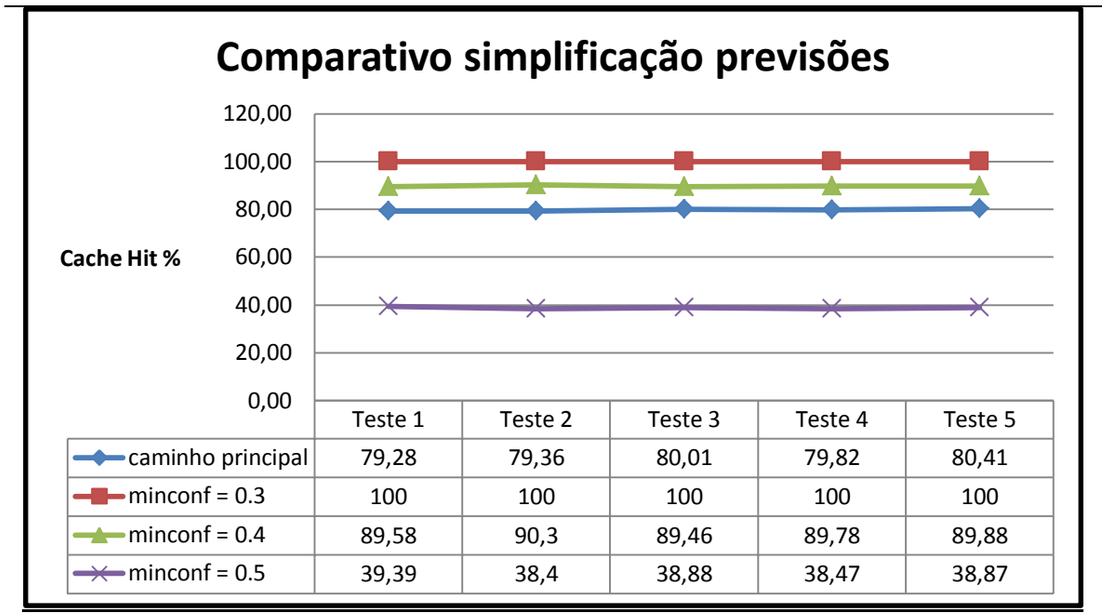
	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
Nº de sessões	989	987	980	990	980
Cache Hits	1667	1625	1608	1619	1687
Cache Misses	2675	2607	2528	2589	2653
Nº de queries efectuadas	4342	4232	4136	4208	4340

Figura 12 - Resultados dos testes (minconf = 0.5)

Neste caso, a percentagem de *cache hits* diminui, em relação aos restantes testes, para valores próximos dos 39% (ver Figura 12). Isto deve-se ao facto de que apenas cerca de 28% das vistas serem materializadas. Como consequência deste processo de simplificação, foi reduzida a carga imposta ao servidor relativamente ao número de vistas a materializar aquando do início de sessão por parte de um utilizador, em prejuízo do tempo médio de resposta aos pedidos deste.

Comparação global

Após reunir todos os resultados dos testes realizados foram comparados estes resultados entre si. Os resultados desse processo de comparação encontram-se sumariados na Figura 13 e que, em detrimento do número de *cache hits*, apresenta, para uma comparação mais justa, as percentagens deste valor atingidas em cada um dos testes.



	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
caminho principal	79,28	79,36	80,01	79,82	80,41
minconf = 0.3	100	100	100	100	100
minconf = 0.4	89,58	90,3	89,46	89,78	89,88
minconf = 0.5	39,39	38,4	38,88	38,47	38,87

Figura 13 - Comparativo resultados dos testes

Analisando o gráfico apresentado (Figura 13), bem como os valores que o suportam - tabela anexa ao gráfico - é possível identificar imediatamente qual o teste que originou melhores resultados, mesmo que seja apenas baseado na métrica de cache hits. Tal como já oportunamente explicado, os resultados atingidos nos testes relativos a um valor de confiança mínima de 30% devem-se ao facto de, apesar de imposto um valor de restrição a nível de regras, no caso específico em estudo, este valor não ter contribuído para a remoção de qualquer uma das vistas da zona de pré-materialização. Deste modo, se por um lado os resultados atingidos são os melhores possíveis, por outro, o número de vistas que é necessário materializar para os atingir poderá revelar-se um problema no que diz respeito ao espaço necessário para armazenar todas estas materializações. Num cenário real, com um conjunto de vistas muito maior do que o testado, a carga computacional necessária para a materialização de todas as vistas que permitam atingir valores de *cache hit* próximos dos 100% pode tornar-se intratável. As dificuldades de materialização de um grande número de vistas prende-se, quer com a capacidade de processamento necessária, quer com o espaço de memória necessário para armazenar estes dados. Analisando os resultados correspondentes aos restantes testes efectuados, podemos observar que, à excepção dos testes

realizados com o valor de confiança mínima de 50%, os resultados em termos de *cache hits* são bastante elevados, ainda que o sejam também as percentagens de vistas materializadas.

Caso os recursos computacionais, ou a sua disponibilidade, fossem menores, as melhorias de desempenho obtidas com o valor de minconf de 50% seriam ainda bastante relevantes, simplesmente à custa de uma solução de compromisso entre as vistas materializadas e os recursos necessários a esta materialização. Para responder a uma *query* OLAP o principal factor que determina o seu tempo de resposta é o tempo dispendido a agregar todos os dados requisitados. Neste sentido, se considerarmos mínimos os tempos de transferência da informação pela rede, que o são, de facto, quando comparados com o tempo necessário à materialização das vistas, podemos afirmar que, mesmo com valores de minconf de 50% nos testes efectuados se obtiveram melhorias de desempenho importantes, na ordem dos 39%. Esta melhoria de desempenho é obtida quando comparada com um sistema OLAP sem qualquer tipo de cache, isto é, um sistema em que todos os pedidos são respondidos directamente pelos respectivos servidores OLAP.

Ganhos da simplificação de regras de previsão associados a probabilidades

Tomando como ponto de comparação um sistema que permita saber, para cada utilizador, qual o conjunto de *queries* que este utilizará em cada sessão mas não qual a sua sequência ou probabilidade associada. Se, num sistema deste género fossem materializados 50% das *queries* previstas, intuitivamente se diria que a probabilidade de estas serem efectivamente pedidas pelo utilizador seria, também, de 50%. Assim sendo, apresenta-se, na Figura que se segue (Figura 14), uma comparação entre um sistema deste género (valores de referência) e o sistema proposto, que realiza a simplificação das regras de associação, tendo em conta as probabilidades de ocorrência destas.

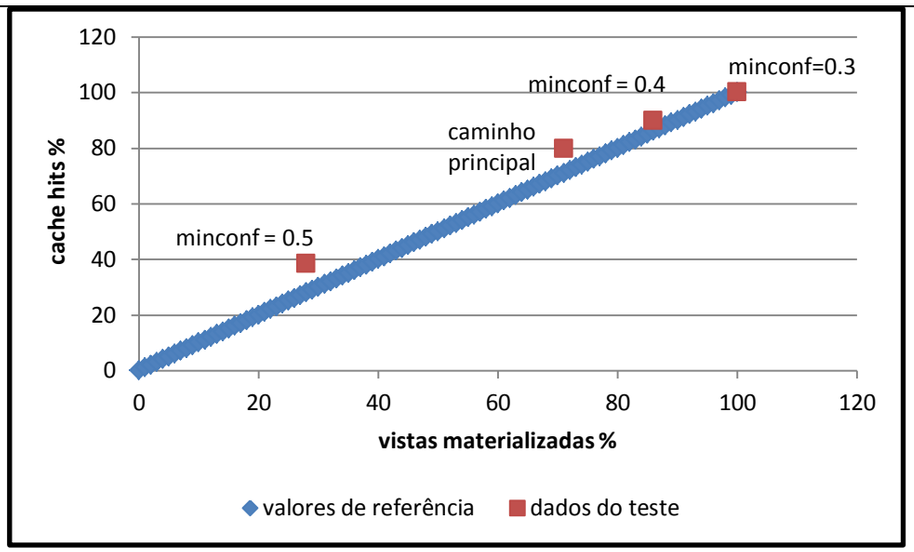


Figura 14 - Valores dos testes vs Valores de referência

Neste gráfico, está representada uma linha que representa os valores de referência anteriormente referidos. Esta linha é equivalente a um gráfico da função $y = x$ pelo que, um ponto que fique "acima" da linha representará, neste caso, uma solução mais vantajosa, uma situação em que com uma mesma percentagem de vistas materializadas se obtêm maiores percentagens de *cache hits*. Como é possível observar pelo gráfico, os pontos correspondentes aos resultados dos testes efectuados encontram-se todos, à excepção do valor de confiança de 30%, acima da referida linha.

5.2 Simplificação do número de vistas a materializar num conjunto de dados real

Tal como realizado para o anterior conjunto de dados, para os dados que se estudam de seguida, foram realizados testes em que foram feitos variar os valores mínimos de confiança aceitável no que diz respeito às regras de associação geradas para estes. O conjunto de dados utilizados para realizar este segundo conjunto de testes foi recolhido de um cenário em que foi simulada uma

utilização real de um sistema OLAP e possui um número total de 59 *queries* distintas realizadas, embora o número de vezes que cada uma delas foi executada seja variável, daí as probabilidades de transição entre as diferentes *queries* sejam, também elas, variáveis.

5.2.1 Cadeia de *Markov* gerada

A aplicação de algoritmos de *data mining* aos dados recolhidos do *Data Warehouse*, resultaram num conjunto de regras de associação que, visualmente, poderão ser representadas pelo seguinte grafo:

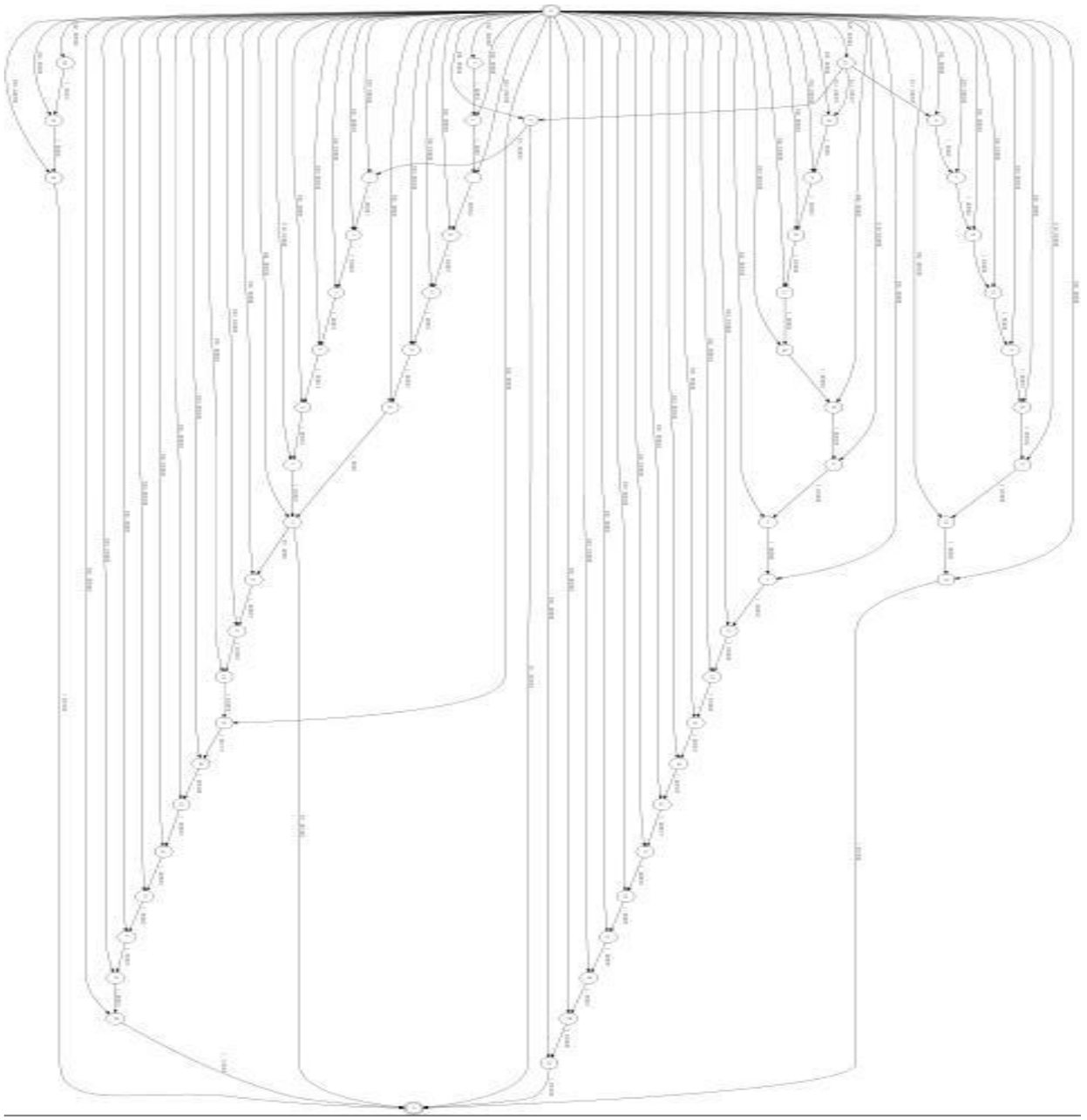


Figura 15 - Representação gráfica do conjunto de regras associadas aos dados reais

Ainda que pouco perceptível, esta imagem permite-nos identificar um número de *queries* possíveis de serem realizadas muito superior ao anteriormente estudado. Por este motivo, certamente a escolha de um critério de materialização das mesmas terá, neste caso, um impacto muito maior no desempenho de todo o sistema.

5.2.2 Identificação dos valores de confiança mínima a considerar

Tal como realizado para o conjunto de dados anterior foram também realizados alguns testes de desempenho (maioritariamente a nível de percentagens de previsões correctamente efectuadas pelo sistema proposto), nos quais foi feita a variação dos valores mínimos de confiança associados à geração de regras de associação.

Pelo facto de a representação visual do grafo não ser facilmente analisável não será, nesta fase de estudo, apresentada para cada um dos valores de confiança mínima, a imagem do grafo resultante da simplificação proposta.

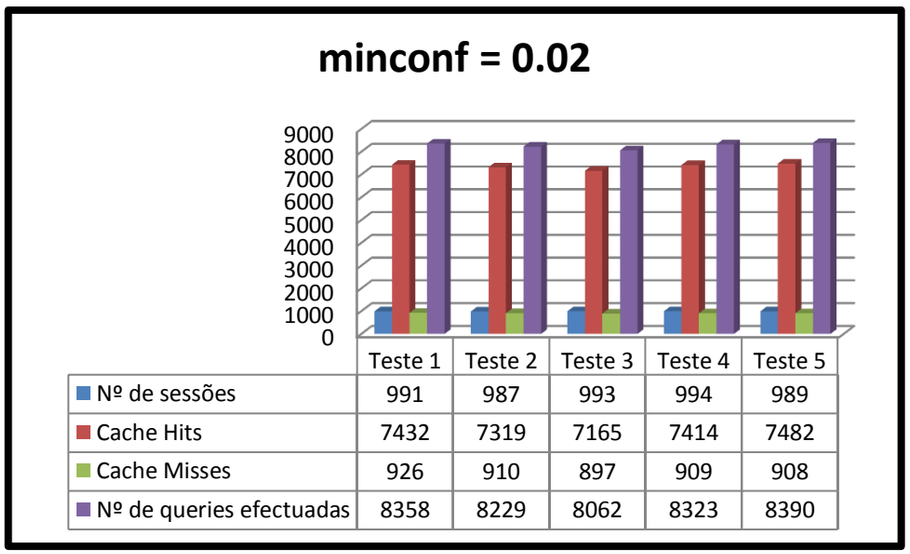
Os valores de confiança mínima testados, neste caso específico, foram adaptados à realidade apresentada por este. Analisando a cadeia de *Markov* obtida, é possível observar que os valores e confiança associados às regras variam entre 0.016 e 1 nos valores específicos {0.016, 0.032, 0.048, 0.5, 1}. Por outro lado, uma rápida observação dos dados recolhidos mostra-nos que, a previsão feita sobre a primeira *query* a ser efectuada imediatamente após o estado inicial, não se encontra bem definida. A probabilidade de esta *query* ser qualquer uma das identificadas na cadeia de *Markov* apresentada é demasiado baixa pelo que a possibilidade de definição de um caminho mais provável a ser seguido pelo utilizador fica de imediato posto de parte. Isto acontece pois, como a probabilidade de erro na primeira previsão efectuada é demasiado alta, todas a previsões que surjam como consequência desta também o serão.

Desta forma, foram definidos valores a testar, de 0.02, 0.4 e 0.6.

5.2.3 Resultados dos testes

Minconf = 0.02

Os resultados dos testes correspondentes à definição de um valor de confiança mínima de 2% encontram-se resumidos na **Error! Reference source not found.**. Para estes testes, e como consequência do valor de confiança mínima utilizado, foram pré-materializadas todas as vistas à excepção daquelas cujo valor de confiança era de 0.016 (1.6%). Assim sendo, foram materializadas 45,9% de todas as vistas que se prevê virem a ser utilizadas para este utilizador específico. Da observação do gráfico, podemos concluir que, apesar de uma redução considerável do número de *queries* a materializar, a percentagem de *cache hits* se mantém bastante elevada (aprox. 89%).

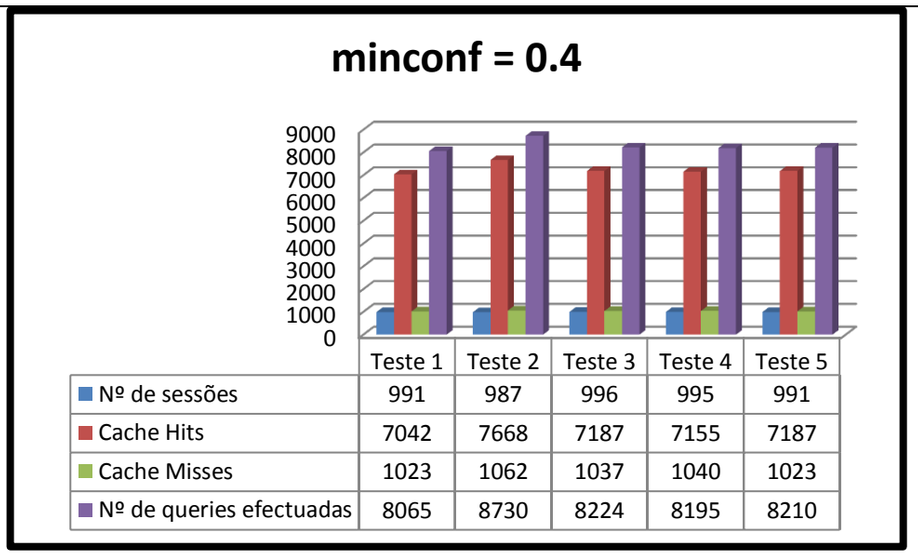


	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
Nº de sessões	991	987	993	994	989
Cache Hits	7432	7319	7165	7414	7482
Cache Misses	926	910	897	909	908
Nº de queries efectuadas	8358	8229	8062	8323	8390

Figura 16 - Resultados dos segundos testes (minconf = 0.02)

Minconf = 0.4

Na Figura que se segue (Figura 17), encontram-se representados os dados relativos aos testes realizados com o valor de confiança mínima de 40%. Neste caso, o número de vistas materializadas foi de 60, isto é, aproximadamente 49%, de todas as previstas.

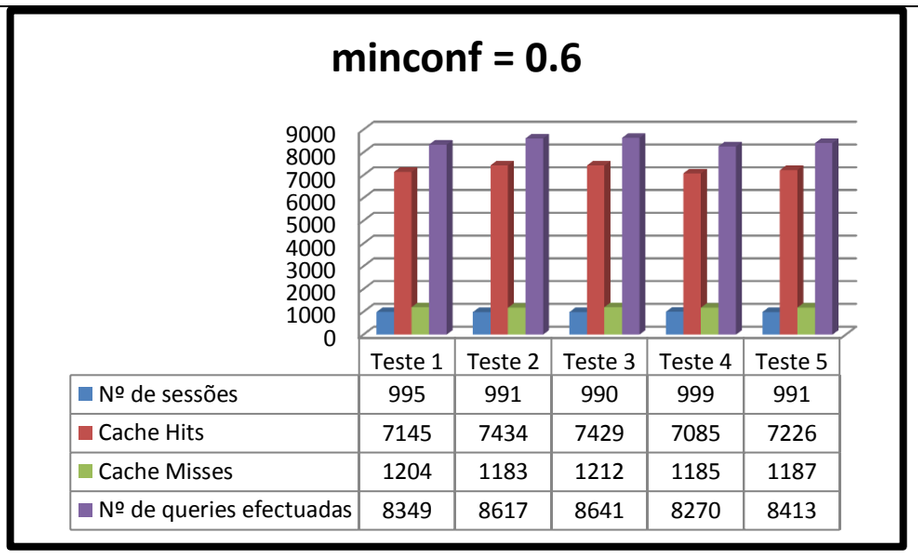


	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
Nº de sessões	991	987	996	995	991
Cache Hits	7042	7668	7187	7155	7187
Cache Misses	1023	1062	1037	1040	1023
Nº de queries efectuadas	8065	8730	8224	8195	8210

Figura 17 - Resultados dos segundos testes (minconf = 0.4)

Minconf = 0.6

Por fim, na Figura 18, apresentam-se os resultados dos testes efectuados com um valor de confiança de 50%. Para este teste, apenas foram materializadas as *queries* correspondentes às regras cujo valor de confiança era de 100% (único valor existente superior a 60%), num total de 56 vistas. Em relação ao total de vistas previstas para materialização, neste caso específico, foram utilizadas aproximadamente 46%.

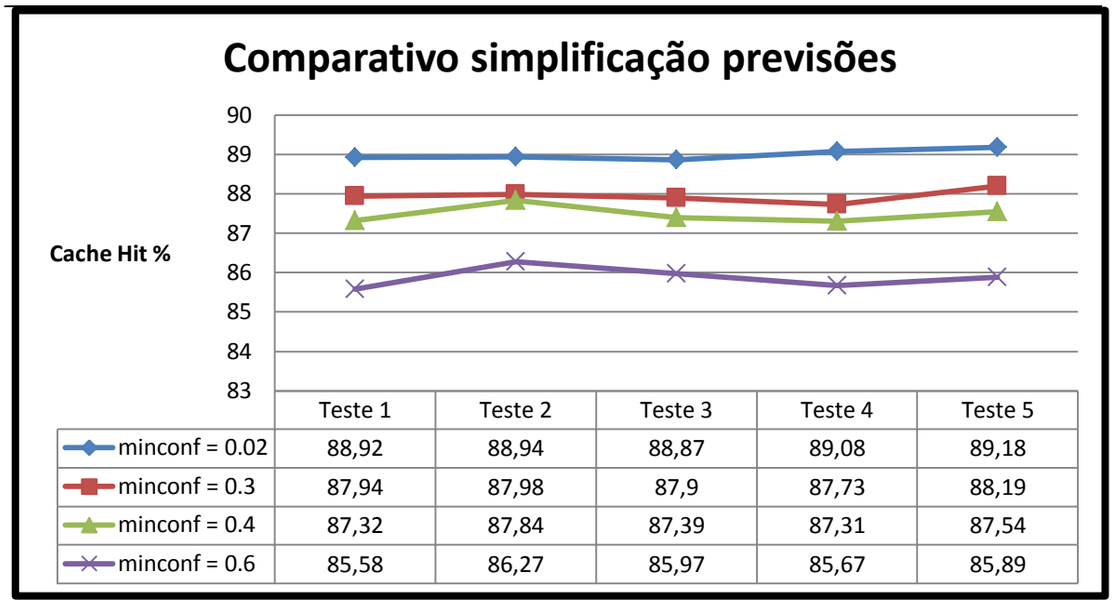


	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
Nº de sessões	995	991	990	999	991
Cache Hits	7145	7434	7429	7085	7226
Cache Misses	1204	1183	1212	1185	1187
Nº de queries efectuadas	8349	8617	8641	8270	8413

Figura 18 - Resultados dos segundos testes (minconf = 0.6)

Comparação Global

De seguida (Figura 19), apresentam-se os valores comparativos entre os testes efectuados para os diferentes valores de confiança mínima.



	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
minconf = 0.02	88,92	88,94	88,87	89,08	89,18
minconf = 0.3	87,94	87,98	87,9	87,73	88,19
minconf = 0.4	87,32	87,84	87,39	87,31	87,54
minconf = 0.6	85,58	86,27	85,97	85,67	85,89

Figura 19 - Comparativo resultados dos segundos testes

Observando a figura 19, conseguimos destacar o facto de, apesar da utilização de um maior número de dados em relação aos primeiros testes efectuados, a utilização de diferentes valores de confiança mínima aceitável, influencia directamente os resultados dos testes. É evidente, após análise dos resultados obtidos, o facto de que uma maior restrição no número de regras consideradas válidas resulta numa diminuição do número de pedidos respondidos directamente da cache. Ainda assim, mesmo com o maior valor de confiança mínima testado (60%), os resultados obtidos indicam que aproximadamente 86% dos pedidos coincidem com valores pré-materializados o que revela ser um enorme ganho pois, neste caso específico, apenas 46% das regras previstas foram materializadas.

Ganhos da simplificação de regras de previsão associados a probabilidades

Na Figura 20 apresentam-se, tal como para o primeiro conjunto de testes, os valores obtidos comparativamente aos valores que seriam espectáveis caso as percentagens de vistas materializadas, o fossem sem ter em conta a probabilidade de serem utilizadas. Isto é, caso

existisse informação relativamente a quais as vistas a serem requisitadas pelo utilizador, mas não qual a probabilidade de cada uma delas acontecer.

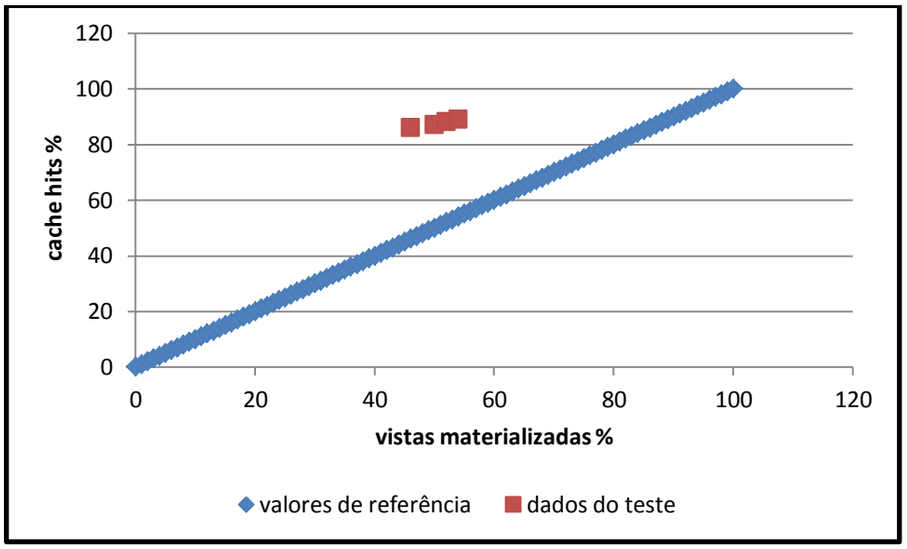


Figura 20 - Valores dos testes vs Valores de referência (segundos testes)

Como é possível observar pelo gráfico anterior, os ganhos relativamente a uma simplificação de regras menos cuidada, são bastante evidentes. Podemos observar, por exemplo, que em relação à materialização de 54% das vistas, no cenário tido como de referência, existiriam aproximadamente 54% dos pedidos respondidos directamente da cache. No caso testado, por sua vez, foi possível responder a aproximadamente 89% de todos os pedidos sem necessidade de aceder ao *Data Warehouse*.

Capítulo 7

Conclusões e Trabalho Futuro

6.1 Conclusões

O grande aumento da utilização de sistemas multidimensionais de dados verificado nos últimos anos provocou um maior interesse ao nível da investigação de novas técnicas para o melhoramento dos sistemas de processamento analítico disponíveis. Dessas técnicas pode-se destacar a implementação e exploração de mecanismos de *caching* para sistemas OLAP, tão sobejamente utilizados noutras áreas, com o objectivo de diminuir a carga à qual se encontra sujeita a fonte central de dados, ou seja, o servidor analítico. Tal como para os sistemas Web, também aqui se optou por, numa fase inicial, implementar este tipo de mecanismos apenas do lado do cliente. Isto fez com que apenas um determinado utilizador beneficiasse dos seus pedidos anteriores. Consequentemente, e como forma de aumentar a vantagem conseguida através da utilização de caches, este tipo de mecanismos passou a centrar-se na partilha de benefícios por todos os utilizadores de um determinado servidor OLAP. Desta forma e através de técnicas de distribuição das caches, quer por meio da partilha das caches individuais dos utilizadores, quer através da criação de sistemas especificamente desenhados para servirem como servidores de cache, foram conseguidas melhorias significativas no desempenho destes sistemas [Bakiras et al. 2003], [Cao et al. 1999], [Kalnis & Papadias 2001], [Kalnis et al. 2002], [Loukopoulos et al. 2005].

No que diz respeito à distribuição de caches, temos que destacar algumas técnicas fundamentais. De entre estas, salientam-se as que se prendem com a criação de ligações directas entre os diversos utilizadores de um determinado servidor OLAP (ou até, possivelmente, de outros). Esta técnica pretende tirar partido de um tipo de arquitectura bastante estudado e amiúde utilizado pelo mais comum dos utilizadores da Web, que são as redes *peer-to-peer*. Através do recurso a este tipo de técnica é possível partilhar os conteúdos da cache de cada um dos utilizadores mas, levando isto ainda um passo adiante, é possível também proceder à partilha de recursos entre

estes. Desta forma é possível não só retirar todos os benefícios da partilha de caches entre os diversos utilizadores, como também da partilha dos recursos entre estes.

Uma outra abordagem possível ao problema da partilha dos recursos de *caching* por todos os utilizadores de um sistema é o recurso a proxies cuja principal funcionalidade é, precisamente, a partilha deste tipo de informação. Este é o caso dos OLAP *Cache Servers*, que mais não são do que conjuntos de proxies, interligadas entre si, optimizadas para realizar funções de *caching* de dados OLAP. Uma das características fundamentais deste tipo de proxies é o facto de poderem armazenar os seus dados de forma a que possibilitem a agregação destes para satisfazer pedidos futuros. Desta forma, é possível que um conjunto de dados mantidos em cache possa ser composto e sejam realizados cálculos sobre estes para responder a um pedido que não corresponda exactamente a algum já mantido em cache. O facto de estes OCS possuírem um optimizador de *queries* permite ainda que sejam utilizadas informações mantidas em diversos pontos da rede. Uma grande vantagem desta funcionalidade é o facto de permitir que não exista, entre os diversos servidores, informação redundante pois esta pode sempre ser requisitada ao nodo que a possua.

Uma outra técnica desenvolvida com o objectivo de retirar dos clientes a obrigação de gerir a própria cache é a introdução da noção de *Active Caching* proposta por [Cao et al. 1999]. Nesta abordagem, associam-se aos dados mantidos em cache pequenos pedaços de código que deverão ser executados quando esta informação seja requisitada. Desta forma, é possível verificar, sempre que necessário, a actualidade dos dados mantidos na cache bem como aumentar a disponibilidade do sistema como um todo, isto porque, em grande parte dos casos, mesmo que o servidor OLAP não se encontre disponível, a utilização das applets permite o cálculo de toda a informação necessária sem necessidade de recurso à fonte primária de dados.

A proposta de [Bakiras et al. 2005] visa fornecer uma contribuição no que diz respeito à problemática da organização de servidores de forma óptima. Recorrendo à utilização de ligações uni e bi-direccionais entre as proxies, é estabelecida uma noção de vizinhança entre estas na qual os nodos pertencentes a uma vizinhança terão idealmente necessidades semelhantes. Mais do que uma nova arquitectura de sistemas de *caching*, a proposta de [Deshpande et al. 1998] propõe que os dados sejam armazenados na cache sob a forma de *chunks*. A principal vantagem desta

abordagem prende-se com a simplicidade de composição deste tipo de informação para responder a pedidos dos utilizadores. Desta forma é ainda possível que, para satisfazer um determinado pedido, sejam compostos dados presentes em cache com outros obtidos directamente a partir do servidor central o que é uma melhoria significativa em relação a outras abordagens estudadas. Em outras propostas estudadas existem apenas duas situações possíveis – a informação pretendida existe em cache (*cache hit*) ou existe apenas no servidor central (*cache miss*). A utilização de *chunks* possibilita a ocorrência de uma situação intermédia na qual apenas parte da informação existe em cache sendo a restante requisitada à fonte central de dados. Isto é conseguido pois a granularidade dos dados mantidos em cache (os *chunks*) é mais refinada do que a frequentemente utilizada (*caching* ao nível da *query*).

Adicionalmente a estas técnicas, a proposta desta tese visa tirar partido da natureza intrínseca da grande maioria dos utilizadores de sistemas OLAP. Esta natureza está maioritariamente relacionada com a posição destes utilizadores dentro das organizações em que se encontram. Ainda hoje, regra geral, a grande maioria dos utilizadores de sistemas OLAP, são membros das organizações com um elevado estatuto dentro da hierarquia organizacional destas. Por esse motivo, as consultas realizadas por estes, têm como objectivo encontrar suporte factual para as decisões que se pretendem tomar. Desta forma, as consultas utilizadas seguem uma tendência específica, ainda que variável de utilizador para utilizador. Sendo possível identificar à *priori* este tipo de tendências, ou padrões, de acesso, podem ser tomadas algumas medidas que melhorem a experiência do utilizador aquando da consulta de dados. De entre estas medidas, a que foi explorada neste trabalho é a proposta de adicionar à cache algumas das *queries* que se prevê virem a ser efectuadas pelo utilizador. Assim, quando estes valores forem necessários já se encontram pré-calculados sendo desta forma satisfeitos de forma muito mais célere.

Um dos problemas que surge neste tipo de abordagem prende-se com o facto de o número de vistas previsto ser demasiado elevado, o que acarreta questões no que diz respeito à viabilidade de pré-materialização destas. Uma solução encontrada para este problema passa pela simplificação do número de vistas a materializar recorrendo à implementação de um "filtro de qualidade" que identifica quais as previsões suficientemente viáveis para serem aceites e quais aquelas que deverão ser ignoradas.

Com este objectivo em vista, foi identificada uma métrica suficientemente identificativa da qualidade e fiabilidade de uma regra – a confiança. Este valor indica a probabilidade de transição entre dois pontos numa cadeia de *Markov* gerada e permite-nos identificar quais as regras cuja capacidade de predição é mais elevada. Todavia, e de forma complementar, uma segunda métrica deveria ter sido analisada: a métrica de suporte. O valor de suporte indica-nos a quantidade de casos no *dataset* que originou as regras de previsão, que conduziram à confirmação da regra em causa. Desta forma, se a transição de A para B apenas tiver ocorrido uma única vez, podemos afirmar que, seja qual for a probabilidade calculada para esta transição, ela terá um suporte baixo, o que indicará que esta transição poderá não se verificar da mesma forma, num próximo cenário. Posteriormente foram testadas simplificações do número de regras, com base em vários valores de confiança mínima associada a estas, que deveria ser aceite. Aplicando, de diversas formas, este critério, foi possível simplificar em diferentes percentagens o número de vistas a serem materializadas (reduzindo assim a carga do servidor num momento específico), embora fosse reduzido também o número de pedidos do cliente satisfeitos directamente a partir da cache.

Os resultados destes testes demonstraram que, nos casos analisados, a simplificação de uma determinada percentagem de regras a serem pré-materializadas não implica que a mesma percentagem de pedidos deixe de ser servida a partir da cache. A dúvida que se impõe, é se este número de *cache hits* é reduzido numa percentagem superior ou inferior à da simplificação das regras. Analisando os dados obtidos pela realização dos testes, tanto recorrendo a dados fictícios (primeiro conjunto de testes), como a dados reais retirados do *Data Warehouse* (segundo conjunto de testes), a técnica de simplificação utilizada revelou-se benéfica em todos os cenários testados.
Numa fase final de análise dos resultados, foi realizada uma comparação entre os valores obtidos através da aplicação desta técnica e os valores que seriam, previsivelmente, obtidos se aplicada uma técnica menos sofisticada de simplificação das regras. Essa técnica, utilizada como valor de referência, correspondia a uma simplificação das regras sem ter em consideração os seus valores de confiança associados. Desta forma, a previsão associada a estes valores de previsão é que a tendência seja para, ao simplificar em determinada percentagem o número de vistas a materializar, a percentagem de valores respondidos directamente a partir da cache decresça em igual medida. Os resultados obtidos, e apresentados na Figura 14 e Figura 20, revelam que a técnica de simplificação das regras utilizada resulta em ganhos no que diz respeito ao quociente entre a percentagem de *queries* removidas da cache e a percentagem de *hit ratio* perdida por essa

intervenção. No que ao primeiro conjunto de testes diz respeito, os ganhos máximos obtidos foram de aproximadamente 10,3%, para valores de confiança mínima de 50%. Por seu lado, o segundo conjunto de testes revelou, em todos os cenários testados, um valor de ganho médio de aproximadamente 35-40%.

Com este conjunto de resultados pode-se afirmar que, apesar das diferenças inerentes à implementação deste tipo de sistema num cenário real, essa aplicação poderia resultar numa melhoria significativa do desempenho nesse cenário. Os dados de um *Data Warehouse* real são em muito maior quantidade o que, se por um lado dificulta o processo de extracção de padrões, por outro permite que os padrões extraídos possuam uma muito maior fiabilidade. Quanto maior for a quantidade de dados disponível para análise, mais fiáveis serão os padrões extraídos a partir destes. Desta forma, é previsível que os resultados obtidos num cenário real sejam bastante satisfatórios.

6.2 Trabalho Futuro

Ao longo dos trabalhos desenvolvidos nesta tese, a dificuldade que se constatou em tentar adquirir um conjunto considerável de dados, foi algo um pouco limitador em termos de testes. Por motivos vários, os dados utilizados tiveram que ser gerados de forma manual. Daí a sua escassez. Com um maior número de casos de teste, preferencialmente provenientes de cenários reais, teria sido possível definir com maior exactidão quais os padrões de acesso dos utilizadores, bem como quais os benefícios emergentes da aplicação das várias técnicas propostas.

Ainda devido à escassez dos dados existentes, bem como por ter sido decidido não integrar no âmbito desta tese a fase do processo em que são analisados os logs do servidor OLAP e de onde serão retiradas as regras de associação que servirão de *input* à técnica desenvolvida, ficará para futuro uma integração destas duas fases do processo descrito. Fica ainda, também, para uma próxima oportunidade, a intenção de implementar um sistema deste género, totalmente funcional, de forma a ter uma integração mais vertical em todo o *data warehouse*, sendo este consequentemente gerido de forma totalmente automatizada. Com isto, pretender-se-á fazer com que um elemento que possa ser francamente vantajoso a nível do desempenho global do sistema de Suporte à Decisão, mais especificamente do servidor OLAP, não seja encarado apenas como

mais uma fonte de manutenção ou um acréscimo de problemas. Desta forma a adesão a este tipo de sistemas poderá aumentar, bem como a sua maturidade, podendo ainda posteriormente serem-lhe acrescentadas novas funcionalidades como, por exemplo, as previsões realizadas terem em conta as pesquisas passadas não só de um utilizador como de toda a comunidade de utilizadores do sistema. Desta forma podem até ser identificados grupos de utilizadores com interesses semelhantes dentro da panóplia de informação que um servidor analítico tem para fornecer. Uma consequência directa deste tipo de análise poderá eventualmente ser a criação de uma cache fisicamente mais próxima desses utilizadores que permita retirar do servidor central alguma carga computacional.

Bibliografia

[Bakiras et al. 2003] Bakiras, S., Loukopoulos T., Ahmad, I., 2003. Dynamic Organization Schemes for Cooperative Proxy Caching. In IPDPS'03 (International Parallel and Distributed Processing Symposium). 2003.

[Bakiras et al. 2005] Bakiras, S., Loukopoulos, T., Papadias, D., Ahmad, I.,2005. Adaptive schemes for distributed Web caching. Journal of Parallel and Distributed Computing, 65, pp 1483-1496.

[Baralis et al. 1997] Baralis, E., Paraboschi, S., Teniente, E., 1997. Materialized Views Seleccion in a Multidimensional Database. In Proceedings of the 23rd International conference on Very Large Databases. Morgan Kaufmann Publishers Inc: San Francisco, CA,USA.

[Bauer & Lehner 2003] Bauer, A., Lehner, W., 2003. On solving the view selection problem in distributed data warehouse architectures.In Proceedings of the 15th International Conference on Scientific and Statistical Database Management. IEEE Computer Socienty: Washington DC, USA.

[Cao & Irani 1997] Cao, P., Irani, S., 1997. Cost-aware WWW proxy caching algorithms. In Proceedings of USENIX Symposium on Internet Technology and Systems. December 1997.

[Cao et al. 1999] Cao, P., Zhang, J., Beach, K., 1999. Active Cache: caching dynamic contents on the Web. In The British Computer Society, The Institution of Electrical Engineers & IOP Publishing Ltd. 1999.

[Cherkasova 1998] Cherkasova, L., 1998. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy.

[Deshpande et al. 1998] Deshpande, P., Ramasamy, K., Shukla, A., Naughton, J., 1998. Caching multidimensional queries using chunks. In ACM SIGMOD. 2005

[Gribble et al. 2001] Gribble, S., Halevy, A., Ives, Z., Rodrig, M., Suciu, D., 2001. What can databases do for Peer-to-Peer. In WebDB.

[Gupta 1999] Gupta, H., 1997. Slection of Views to Materialize in a Data Warehouse. In Proceedings of the 6th International Conference on Database Theory. Springer-Verlag:London, UK.

[Harinarayan et al. 1996] Harinarayan, V., Rajaraman, A., Ullman, J., 1996. Implementing data cubes efficiently. ACM SIGMOD Record, 25(2), pp.205-216.

[Howard, 1960] Howard, R., 1960. Dynamic Programming and Markov Processes. MIT Press.

[Kalnis & Papadias 2001] Kalnis, P., Papadias, D., 2001. Proxy-Server Architectures for OLAP. In ACM SIGMOD. Santa Barbara, California, USA, 21-24 May 2001.

[Kalnis et al. 2002] Kalnis, P., Ng, W., Ooi, B., Papadias, D., Tan, K., 2002. An adaptive peer-to-peer network for distributed caching of OLAP results. In ACM SIGMOD. Madison, Wisconsin, USA, 2002.

[Karayannidis & Sellis 2001] Karayannidis, N., Sellis, T., 2001. SISYPHUS: A chunk-based storage manager for OLAP cubes. In DMDW '2001 (International Workshop on Design and Management of Data Warehouses). Interlaken, Switzerland, June 4, 2001.

[Kotidis & Roussopoulos 2001] Kotidis, Y., Roussopoulos,N., 2001. A case for dynamic view management. In ACM Transactions on Database Systems. ACM: New York, USA.

[Lawrence et al. 2007] Lawrence, M., Dehne, F., Rau-Chaplin, A., 2007. Implementing OLAP Query Fragment Aggregation and Recombination for the OLAP Enabled Grid.

[Lehner et al. 2000] Lehner, W., Albrecht, J., Hümer,W., ANO. Divide and Aggregate: caching multidimensional objects. In Proceedings of the Second Intl. Workshop on Design and Management of Data Warehouses, DMDW 2000, Stockholm, Sweden, June 5-6, 2000

[Loukopoulos et al. 2005] Loukopoulos, T., Kalnis, P., Ahmnad, I., Papadias, D., 2001. Active Caching of On-Line-Analytical-Processing Queries in WWW Proxies. In ICPP'01 (International Conference on Parallel Processing). 2005.

[Markl et al. 1999] Markl, V., Ramsak, F., Bayer, R., Forschungszentrum, B., 1999. Improving OLAP Performance by Multidimensional Hierarchical Clustering.In Proc. of IDEAS'99

[Mookerjee & Tan 2002] Mookerjee, VS., Tan, Y., 2002. Analysis of a least recently used cache management policy for Web browsers. Operations Research.

[Ng et al. 2002] Ng, W.S., Beng,Ng., Ooi, C., Tan, K., 2002. BestPeer: A Self-Configurable Peer-to-Peer System

[Ramanchandran et al. 2005] Ramachandran, K., Shah, B., Raghavan, V., 2005. Dynamic pre-fetching of views based on user-access patterns in an OLAP system. In ACM SIGMOD. 2005

[Roussopoulos et al. 1997] Roussopoulos, N., Kotidis, Y., Roussopoulos, M. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In Proceedings of the 1997 ACM SIGMOD Conference.

[Sapia 2000] Sapia, C., 2000. PROMISE: Predicting Query Behavior to enable predictive caching strategies for OLAP systems. In DAWAK'2000 (Proceedings of the Second International Conference on Data warehousing and Knowledge Discovery). Greenwich, UK, September 2000, Springer LNCS, 2000.

[Sarawagi 1997] Sarawagi, S., 1997. Indexing OLAP data. Bulletin of the Technical Committee on Data Engineering 20, pp.36-43.

[Scheuerman et al. 1996] Scheuerman, P., Shim, J., Vingralek, R., 1996. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In Proceedings of the 22nd VLDB Conference. Mumbay(Bombay), India, 1996.

[Shim et al. 1999]Shim, J., Scheuermann, P., Vingralek, R., 1999.Dynamic Caching in Query Results for Decision Support Systems.In Proceedings of the SSDBM Conference. Springer Verlag, pp.254-263.

[Sun et al. 1989] Sun, X., Kamel, N., Ni, L., 1989. Solving implication problems in database applications. ACM, New York, USA.

[Vassialidis & Skiadopoulos 2000] Vassiliadis, P., Skiadopoulos, S., 2000. Modelling and Optimization issues for Multidimensional Databases. In Proceedings of the 12th International Conference on Advanced Information Systems Engineering. Springer-Verlag, 1999

[Yao & An 2003] Yao, Q., An, A., 2003. Using user access patterns for semantic query caching. In Database and Expert Systems Applications, 14th International Conference. Prague, Czech Republic, September 1-5, 2003, Springer Berlin.

[Young 1991] Young, N., 1991. On-Line Caching as size varies. In Symposium on Discrete Algorithms (Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms). San Francisco, California, United States, January 28 - 30, 1991.