Frederico A. C. A. Gonçalves

Otimização de Consultas com Muitas Junções Utilizando Sistemas Multiagente Evolucionários

Belo Horizonte–MG

Setembro/2014

Frederico A. C. A. Gonçalves

Otimização de Consultas com Muitas Junções Utilizando Sistemas Multiagente Evolucionários

Tese submetida à Escola de Engenharia da Universidade Federal de Minas Gerais como requisito para qualificação no Programa de Pós-Graduação em Engenharia Elétrica.

Universidade Federal de Minas Gerais – UFMG

Programa de Pós-Graduação em Engenharia Elétrica – PPGEE

Orientador: Frederico G. Guimarães

Coorientador: Marcone J. F. Souza

Belo Horizonte–MG

Setembro/2014

Frederico A. C. A. Gonçalves
Otimização de Consultas com Muitas Junções Utilizando Sistemas Multiagente Evolucionários/ Frederico A. C. A. Gonçalves. – Belo Horizonte–MG, Setembro/2014-
192 p. : il. (algumas color.) ; 30 cm.

Orientador: Frederico G. Guimarães

Tese – Universidade Federal de Minas Gerais – UFMG
Programa de Pós-Graduação em Engenharia Elétrica – PPGEE, Setembro/2014.

1. Problema de Ordenação de Junções. 2. Otimização de Consultas. 3. Sistemas Multiagente. 4. Algoritmos Evolucionários. 5. Heurísticas. I. Frederico G. Guimarães. II. Universidade Federal de Minas Gerais – UFMG. III. Escola de Engenharia. IV. Otimização de Consultas com Muitas Junções Utilizando Sistemas Multiagente Evolucionários
CDU 62:621.3

Frederico A. C. A. Gonçalves

Otimização de Consultas com Muitas Junções Utilizando Sistemas Multiagente Evolucionários

Tese submetida à Escola de Engenharia da Universidade Federal de Minas Gerais como requisito para qualificação no Programa de Pós-Graduação em Engenharia Elétrica.

Trabalho aprovado. Belo Horizonte–MG, 22 de agosto de 2014:

Frederico G. Guimarães
Orientador

Marcone J. F. Souza
Co-orientador

Marta Lima de Queiros Mattoso
Convidado 1

Alan Robert Resende de Freitas
Convidado 2

André Paim Lemos
Convidado 3

Belo Horizonte–MG
Setembro/2014

Este trabalho é dedicado especialmente aos meus queridos filhos Pandora e Aquiles.

Agradecimentos

Agradeço ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais pela oportunidade. Aos Professores Frederico Gadelha e Marcone Jamilson Freitas Souza, pelos ensinamentos e pela orientação, essencial no desenvolvimento deste trabalho. Agradeço também ao Núcleo de Tecnologia da Informação da Universidade Federal de Ouro Preto. Aos amigos de Conselheiro Lafaiete, Ouro Preto e Belo Horizonte. Aos meus pais e meu irmão, pelo apoio. À minha querida esposa Dani, por todo amor, paciência e incentivo. Agradeço principalmente aos meus filhos Pandora e Aquiles, por existirem e tornarem tudo possível. Encerro deixando a seguinte mensagem: Meus amados filhos, sempre, tudo por vocês e pra vocês!

Resumo

Este trabalho apresenta um sistema multiagente evolucionário para a otimização de consultas em Sistemas de Banco de Dados Relacionais (SGBDR) em ambientes não-distribuídos. A otimização de consultas lida com um problema comumente conhecido como problema de ordenação de junções, o qual tem impacto direto no desempenho de tais sistemas. A técnica proposta foi programada no núcleo de otimização do SGBDR H2. A seção experimental foi projetada em acordo com um planejamento fatorial de efeitos fixos e a análise dos resultados apoiou-se no Teste de Permutações para Análise de Variância. A metodologia experimental empregou o *TPC-DS* (*Transaction Processing Performance Council - The New Decision Support Benchmark Standard*) e um outro *benchmark* sintético. Os testes foram divididos em três experimentos distintos: calibração do algoritmo, validação com um método exaustivo no *TPC-DS* e uma comparação geral utilizando o *benchmark* sintético com vários SGBDRs: H2, *Apache Derby*, HSQLDB e *PostgreSQL*. Os resultados mostraram que o sistema multiagente evolucionário proposto foi capaz gerar soluções associadas a custos menores e tempos de execução mais rápidos na grande maioria dos casos.

Palavras-chaves: Problema de Ordenação de Junções. Otimização de Consultas. Sistemas Multiagente. Algoritmos Evolucionários. Heurísticas.

Abstract

This work presents an evolutionary multi-agent system applied to the query optimization phase of Relational Database Management Systems (RDBMS) in a non-distributed environment. The query optimization phase deals with a known problem called query join ordering, which has a direct impact on the performance of such systems. The proposed approach was programmed in the optimization core of the H2 Database Engine. The experimental section was designed according to a factorial design of fixed effects and the analysis based on the Permutations Test for an Analysis of Variance Design. The evaluation methodologies are based on TPC-DS (Transaction Processing Performance Council - The New Decision Support Benchmark Standard) and a synthetic benchmark. The tests were divided into three different experiments: calibration of the algorithm, validation with an exhaustive method on TPC-DS and a general comparison on a synthetic benchmark with different database systems, namely Apache Derby, HSQLDB and PostgreSQL. The results show that the proposed evolutionary multi-agent system was able to generate solutions associated with lower cost plans and faster execution times in the majority of the cases.

Key-words: Join Ordering Problem. Query Optimization. Multi-agent System. Evolutionary Algorithm. Heuristics.

Lista de ilustrações

Lista de tabelas

Lista de algoritmos

Sumário

1 Introdução

Em um mundo cada vez mais informatizado e conectado, o armazenamento e a recuperação de informações desempenham um papel fundamental em qualquer segmento de negócio. Neste contexto, pode-se dizer que um Sistema Gerenciador de Banco de Dados (SGBD) representa uma solução viável e eficiente na manipulação de informações em geral. Os SGBDs pertencem a uma classe complexa de programas, cujas funções essenciais podem ser resumidas como segue: armazenamento dos dados de forma segura e eficiente, mecanismo para recuperação posterior dos dados e controle das alterações realizadas pelos usuários do sistema. Mais ainda, os SGBDs são empregados em diversas áreas, sendo algumas delas:

- Instituições de educação;
- Instituições Financeiras;
- Sistemas de comércio eletrônico;
- Pesquisas científicas;
- Sistemas de apoio para tomada de decisões;
- Aplicações multimídia;
- Armazém de dados.

De acordo com 2008c GARCIA-MOLINA; ULLMAN; WIDOM;, um SGBD pode ter as seguintes funcionalidades destacadas:

1. **Armazenamento Persistente** - Tal como os Sistemas de Arquivos em Sistemas Operacionais, o SGBD permite o armazenamento de uma grande quantidade de informação. Além disso, ele permite que os dados sejam definidos de forma flexível e recuperados eficientemente;
2. **Interface de Programação** - O SGBD define uma interface que permite facilmente o acesso e modificação dos dados armazenados. Utilizando-se de uma poderosa linguagem de consulta, várias tarefas de manipulação dos dados podem ser executadas de forma simples;
3. **Controle de Transação** - Os Sistemas de Banco de Dados permitem que vários usuários/aplicações acessem simultaneamente as informações armazenadas. O

sistema provê um mecanismo de controle que impede um mau funcionamento em função dos vários acessos simultâneos.

Vários modelos de banco de dados foram propostos ao longo dos anos, sendo o modelo relacional (CODD, 1970) o de maior destaque (ELMASRI; NAVATHE, 2010a). Contudo, vale ressaltar a grande importância que se tem dado a ambientes nos quais são produzidos volumes expressivos de informações, podendo superar os *petabytes* diários. Tal ambiente pode ser composto por fontes heterogêneas de dados e normalmente supera a capacidade que pode ser processada por sistemas tradicionais baseados no modelo relacional. Neste contexto, surge o termo *Big Data*, que pode ser definido com base nos seguintes pontos: grande volume de dados (acima dos *terabytes*), velocidade no que diz respeito ao crescimento dos dados e ao tratamento de tais informações para atender a uma demanda, e por último, a variedade no formato dos dados (formato estruturado, semiestruturado, etc). Assim, com vistas a atender necessidades inerentes ao *Big Data*, novos sistemas do tipo *NoSQL* (CATTELL, 2011) têm se destacado na comunidade. Sistemas *NoSQL* podem ser divididos em vários grupos, dentre eles: Armazenamento Chave/Valor (SADALAGE; FOWLER, 2012), Orientados a Documento (SADALAGE; FOWLER, 2012), Orientados a Colunas (HOLLOWAY; DEWITT, 2008) e RDF (KLYNE; CARROLL, 2004).

Neste trabalho optou-se por trabalhar com o modelo relacional, e portanto, o problema foco foi tratado no âmbito de um Sistema Gerenciador de Banco de Dados Relacional (SGBDR) tradicional (GARCIA-MOLINA; ULLMAN; WIDOM, 2008c; ELMASRI; NAVATHE, 2010a). Embora seja inquestionável a atenção recebida por soluções relacionadas ao *Big Data*, o autor entende que o SGBDR ainda é uma solução mais popular que as outras, sendo a mesma mais utilizada no mercado corporativo em várias situações. Desta maneira, uma contribuição neste campo teria um benefício mais amplo. Além disso, sistemas *NoSQL* objetivam primariamente pelo desempenho e muitas vezes desconhecem o significado do dado armazenado, ou seja, é necessária a escrita de componentes customizados para tratar a informação retornada, o que poderia impactar a seção experimental no que diz respeito à complexidade do código responsável pela análise dos sistemas envolvidos e a padronização de avaliação dos resultados. Mais ainda, é válido notar que o SGBDR tradicional adotado como base para a implementação das contribuições do presente trabalho foi o H2[1].

Ainda em relação a escolha pelo SGBDR tradicional, ressalta-se que o problema de otimização tratado aqui está intimamente ligado à operação relacional de junção (GARCIA-MOLINA; ULLMAN; WIDOM, 2008c) (o problema citado será detalhado no Capítulo 2). Deste modo, alguns dos principais projetos NoSQL na comunidade foram descartados por não suportar tal operação, sendo alguns deles: Cas-

[1] *H2 Database*: http://www.h2database.com/

sandra[2], *HBase*[3] e *MongoDB*[4]. Mais ainda, alguns SGBDs orientados a colunas compatíveis com um SGBDR também foram descartados devido à indisponibilidade do código fonte, o que impossibilitaria à implementação do algoritmo proposto no núcleo de otimização do sistema em questão, dentre eles: *IBM DB2*[5], *Microsoft SQL Server*[6], Vertica[7] (versão comercial do descontinuado *C-store*[8]).

A Figura 1 apresenta um esquema simples para o fluxo percorrido por uma requisição até seu resultado. No processamento de uma consulta são executados uma série de passos que permitem ao SGBDR retornar o resultado corretamente e eficientemente para o solicitante. Ressalta-se que durante este processamento, o problema foco deste trabalho é otimizado pelo SGBDR.

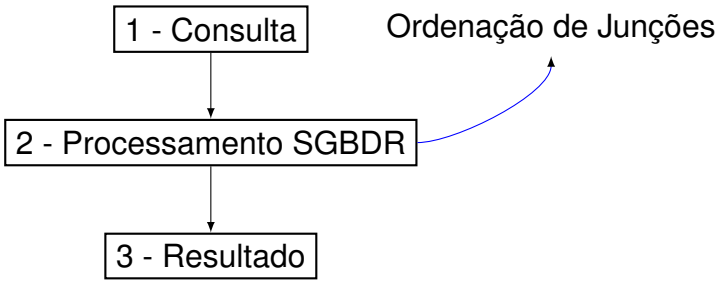


Figura 1 – Esquema Processamento

Como pode ser visto na Figura 2, uma consulta submetida ao SGBDR pode conter a solicitação de tuplas de várias relações presentes no banco de dados. Tais consultas podem conter uma série de filtros, isto é, deverão obedecer a determinados condições de seleção (ex.: retornar apenas os alunos do Departamento de Computação), filtros de conexão entre as relações (ex.: retornar a junção entre os alunos e as suas notas), etc. Uma consulta pode ser definida por meio de várias operações de álgebra relacional. Como todas as operações geram uma relação como resultado, é possível combinar várias operações relacionais em uma única expressão algébrica. Os seguintes operadores relacionais podem ser destacados: interseção ($\cap$), união ($\cup$), diferença ($\setminus$), junção ($\bowtie$), etc.

É importante observar que uma expressão de álgebra relacional pode ser representada por diferentes expressões equivalentes, isto é, os operadores relacionais possuem propriedades que permitem a uma expressão ser reescrita para outra equivalente com alterações na ordem em que os operadores são aplicados à consulta.

[2] Cassandra: http://www.datastax.com/documentation/cassandra/2.0/cassandra/cql.html
[3] *HBase*: http://hbase.apache.org/book/joins.html
[4] *MongoDB*: http://docs.mongodb.org/manual/faq/fundamentals/
[5] *IBM DB2*: http://www.ibm.com/software/data/db2/
[6] *SQL Server*: https://www.microsoft.com/pt-br/server-cloud/products/sql-server/
[7] Vertica: http://www.vertica.com/
[8] *C-Store*: http://db.lcs.mit.edu/projects/cstore/

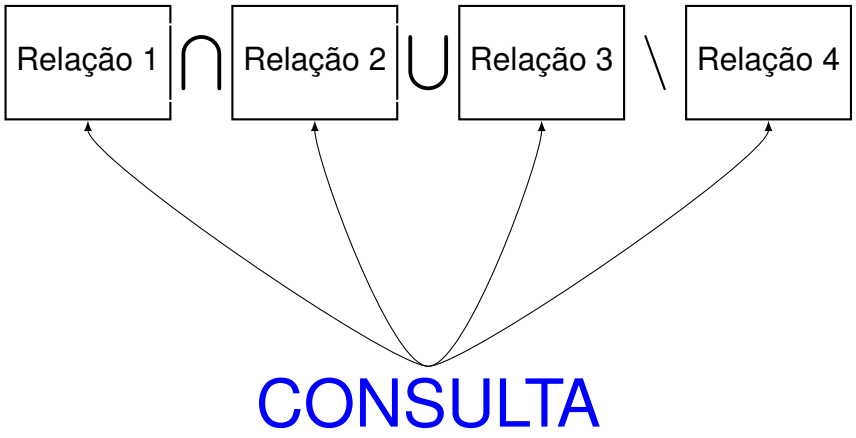


Figura 2 – Consulta

Analogamente, durante o processamento de uma consulta (passo 2 da Figura 1), o SGBDR pode trabalhar com várias soluções equivalentes para as consultas. Mais ainda, dentre outros aspectos, tais soluções podem se diferenciar na ordem de execução dos operadores relacionais, o que pode influenciar diretamente no custo de execução das consultas. Portanto, é necessária a definição de um plano que minimize o custo final de execução da consulta. Tratando-se especificamente de consultas compostas apenas por operações de junção, aquela que mais consome tempo no processamento de consultas (ELMASRI; NAVATHE, 2010a), a tarefa de ordenação das operações de junção pode ser visualizada como um problema de otimização combinatória, comumente conhecido como Problema de Ordenação de Junções.

O Problema de Ordenação de Junções possui semelhanças com o Problema do Caixeiro Viajante (*Travelling Salesman Problem* – TSP). Assim como no TSP (CORMEN et al., 2001) e de acordo com 1984 IBARAKI; KAMEDA;, o Problema de Ordenação de Junções pertence à classe de problemas NP-Completos. No capítulo 2, o problema em questão é descrito em mais detalhes.

Em problemas complexos como o de ordenação de junções, nem sempre é viável a obtenção da melhor solução global. Sabe-se que no problema em questão, metodologias exatas ou exaustivas podem ser uma opção dependendo da complexidade da consulta avaliada, isto é, a utilização da metodologia depende do número de junções envolvidas. Além dos métodos exatos, existem aqueles que utilizam heurísticas, ou formas inteligentes de avaliação do espaço de busca. Tais métodos aplicam técnicas que procuram reduzir e direcionar a busca por boas soluções. Uma classe de métodos heurísticos de uso geral é conhecida como meta-heurísticas. Estes métodos fazem uma exploração mais eficiente e ampla do espaço de soluções, tendo mecanismos que evitam a estagnação em ótimos locais que impedem o seu progresso. Dentre as meta-heurísticas conhecidas, podem ser citadas: Recozimento Simulado, Busca

Tabu, Busca em Vizinhança Variável ou GRASP (*Greedy Randomized Adaptive Search Procedure*).

Uma classe de otimização populacional que merece ser destacada é conhecida como Algoritmo Evolucionário (AE). Tal classe caracteriza-se por trabalhar com uma população de soluções candidatas (indivíduos), ao passo que as meta-heurísticas citadas anteriormente trabalham com apenas uma solução por vez. Sabe-se que um AE evolui iterativamente por meio do uso de operadores heurísticos inspirados ou motivados por conceitos de sistemas naturais e princípios do modelo de Darwin. Em um típico AE, a aptidão de um indivíduo depende tão somente de sua qualidade para resolver o problema.

Além das técnicas de otimização citadas anteriormente, uma outra forma de tratar a ordenação de junções é por meio do uso de Sistema Multiagente (SMA). Define-se como SMA, um grupo de agentes inteligentes trabalhando juntos de forma cooperativa ou competitiva para a solução de um problema. Tais sistemas se diferenciam de outros puramente paralelos pela forma de interação entre os agentes. Os agentes em questão possuem características diferenciadas tais como: reatividade, proatividade e habilidades sociais.

Neste trabalho empregou-se a união de duas técnicas de otimização distintas: SMA e AE. Esta união é definida como Sistema Multiagente Evolucionário (SMAE), onde há a realização do processo evolucionário unido ao ambiente multiagente. Desta maneira, os agentes evoluem, reproduzem, sofrem mutações, competem por recursos, comunicam-se com outros agentes, morrem, tomam decisões autônomas e escolhem parceiros para reprodução (DREZEWSKI; OBROCKI; SIWIK, 2010). Por fim, dois mecanismos de evolução do SMAE podem ser citados: evolução cooperativa (onde ocorre cooperação entre os agentes) e evolução competitiva (onde ocorre competição entre os agentes).

O uso de sistemas multiagente unido à metodologia evolucionária é uma forma de se explorar o melhor de cada mundo. SMA permite que o planejamento seja realizado de forma paralela, ou seja, cada agente trabalha paralelamente numa solução para o problema. Além disso, os agentes podem atuar de forma pró-ativa (tomar decisões) ou reativa (reagir a estímulos). Por outro lado, a metodologia evolucionária possibilita que os agentes analisem o espaço de soluções de forma inteligente. Algoritmos evolucionários são comprovadamente eficientes na otimização de vários problemas da literatura (BENNETT; FERRIS; IOANNIDIS, 1991; STEINBRUNN; MOERKOTTE; KEMPER, 1997; RHO; MARCH, 1997; DONG; LIANG, 2007; MULERO, 2007; SEVINC; COSAR, 2011; GOLSHANARA; ROUHANI RANKOOHI; SHAH-HOSSEINI, 2014). Inclusive, podem ser encontrados na literatura alguns trabalhos com implementação de algoritmos evolucionários aplicados à ordenação de junções. Mais ainda,

uma versão inicial deste trabalho (GONÇALVES; GUIMARÃES; SOUZA, 2013) foi publicada no evento *Genetic and Evolutionary Computation Conference* 2013[9]. Por fim, as ideias apresentadas em 2013 GONÇALVES; GUIMARÃES; SOUZA; foram estendidas em 2014 GONÇALVES; GUIMARÃES; SOUZA; e posteriormente aceitas para publicação no periódico *Experts Systems With Applications*[10].

1.1 Motivação

O desenvolvimento de técnicas que possam contribuir para o desempenho de SGBDRs é um grande desafio. O problema de ordenação de junções pode ter grande impacto no tempo de resposta das consultas submetidas a um SGBDR, uma vez que planos com ordenações ruins podem causar uma grande quantidade de operações de *E/S* ou operações de *CPU*. Desta forma, o estudo de algoritmos que possam contribuir para a geração de planos melhores em tempos viáveis é um grande motivador.

Além disso, nos dias atuais, uma grande quantidade de SGBDRs permitem que o código-fonte do projeto seja obtido em um sítio público. Assim, é possível implementar os algoritmos e incluí-los diretamente no núcleo de otimização destes SGBDRs, compará-los com outras técnicas e vê-los funcionando na prática. Assim, elimina-se a necessidade de se implementar componentes para simular o comportamento de funcionalidades necessárias pelo otimizador e focar somente na resolução do problema. Mais ainda, a execução em um SGBDR real, isto é, a otimização de consultas no núcleo de SGBDRs reais é mais atraente e pode refletir melhor os pontos fortes e fracos do algoritmo.

Na revisão bibliográfica deste trabalho, observou-se que o uso de sistemas multiagente aplicado ao problema de ordenação de junções se mostrou muito pouco explorado na literatura. De nosso conhecimento, não há qualquer trabalho proposto que tenha explorado SMAE no âmbito do problema citado. Assim, pode-se destacar como maior motivação deste trabalho, a avaliação de uma nova metodologia aplicada ao problema de ordenação de junções em um SGBDR.

1.2 Contribuições

Destaca-se como principal contribuição deste trabalho, o desenvolvimento de uma nova abordagem não-exaustiva aplicada ao problema de ordenação de junções envolvendo muitas relações em Sistemas Gerenciadores de Banco de Dados Relacionais. Tal algoritmo baseia-se na metodologia híbrida Sistemas Multiagente Evoluci-

[9] http://www.sigevo.org/gecco-2013/

[10] http://www.journals.elsevier.com/expert-systems-with-applications/

onários e foi incluída no núcleo de otimização do SGBDR H2 centralizado (ambiente não-distribuído). É importante ressaltar que os agentes do algoritmo proposto são capazes de interagir e evoluir paralelamente, isto é, o problema em questão é resolvido de forma paralela, o que preenche uma lacuna da literatura, uma vez que o levantamento bibliográfico do Capítulo 4 mostra que a grande maioria dos trabalhos tende a executar a otimização de forma sequencial. Além disso, cabe ressaltar que a afirmação anterior se refere à resolução paralela do problema citado, e não à geração de planos para execução paralela ou a execução do plano gerado de forma paralela (paralelização inter-operação e intra-operação (ÖZSU; VALDURIEZ, 2011)), sendo este um outro campo de pesquisa.

Ainda em relação ao algoritmo proposto, vale frisar o desenvolvimento de um novo operador de cruzamento denominado Pandora-Aquiles *Greedy Crossover* - PAGX e uma estratégia diferenciada para a execução paralela do método de refinamento Descida Completa (*Best Improvement*), a qual se apoia totalmente no ambiente multiagente projetado.

Não menos importante, cita-se também como contribuição, a calibração do algoritmo e avaliação do mesmo apoiada por um planejamento fatorial de efeitos fixos com a análise estatística dos dados baseada em ANOVA Permutacional (Teste de Permutações (ANDERSON; BRAAK, 2003)).

Por fim, destaca-se como outra fonte de contribuição, a implementação de uma metodologia de avaliação experimental que estende ideias de outros trabalhos da literatura. Tal metodologia tem como objetivo a criação de um ambiente de testes mais realista, devido a inclusão de alguns parâmetros para criação do ambiente, os quais se baseiam em distribuições reais de um banco de dados de produção de uma universidade brasileira.

1.3 Organização do Texto

O restante deste trabalho é organizado como segue. No capítulo 2 é feita uma revisão sobre o processamento de consultas em banco de dados relacionais, o que inclui a análise do problema de ordenação de junções. Nesse capítulo, também discorre-se mais detalhadamente acerca do problema de ordenação de junções. O planejamento e execução de consultas do SGBDR *H2* é discutido no capítulo 3. Uma revisão dos trabalhos relacionados é feita no capítulo 4. O algoritmo desenvolvido é apresentado no Capítulo 5. No Capítulo 6 é abordada a metodologia de avaliação utilizada, a calibração do algoritmo desenvolvido, assim como a comparação dos resultados. Por fim, as conclusões, bem como direcionamentos futuros do trabalho são exibidos no Capítulo 7.

2 Processamento de Consultas

Os Sistemas Gerenciadores de Banco de Dados são compostos por vários componentes que trabalham para gerenciar um conjunto de dados mantidos por eles. De acordo com 2010a ELMASRI; NAVATHE;, o fluxo de processamento de uma consulta submetida ao SGBDR pode ser dividido nos passos apresentados na Figura 3.

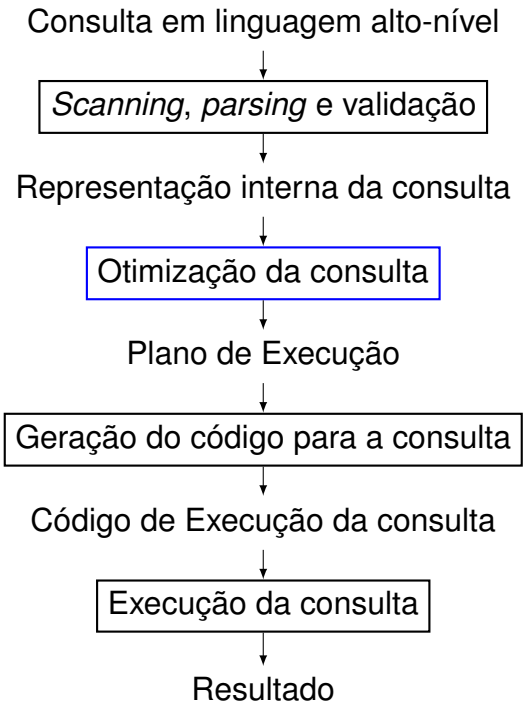


Figura 3 – Passos típicos para o processamento de uma consulta.

Inicialmente, a consulta é submetida em linguagem alto-nível, comumente SQL (*Structured Query Language*), que é a linguagem padrão utilizada pelos SGBDRs relacionais para tal tarefa. A seguir, a consulta é verificada pelo *scanner*, que identifica os *tokens*, tais como palavras-chave da linguagem SQL, colunas, tabelas, etc. O *parser* então realiza uma análise sintática para verificar se a consulta não viola a gramática definida. Essa fase de validação é responsável por fazer algumas verificações semânticas e análises tais como: nomes das colunas das tabelas, nomes das tabelas, etc.

Após a validação da consulta, a mesma é traduzida em uma representação interna reconhecida pelo SGBDR. De acordo com 2010a ELMASRI; NAVATHE;, esta representação pode ser descrita como sendo uma expressão equivalente e estendida de álgebra relacional. A álgebra relacional é uma linguagem de consulta formal procedimental que permite ao usuário solicitar informações à base de dados por meio da especificação dos dados necessários e da forma de como obtê-los. Dois grupos de ope-

rações podem ser usados para dividir as operações de álgebra relacional. Um grupo inclui operações da teoria matemática de conjuntos, tais como: UNIÃO (*UNION*), INTERSEÇÃO (*INTERSECTION*), DIFERENÇA (*SET DIFFERENCE*) e PRODUTO CARTESIANO (*CARTESIAN PRODUCT*). Já no outro grupo, são encontradas operações específicas para bancos de dados relacionais, dentre elas: SELEÇÃO (SELECT), PROJEÇÃO (PROJECT), JUNÇÃO (JOIN) e ORDENAÇÃO (ORDER). A Tabela 1 descreve a função de cada uma das operações citadas.

Operação	Exemplo	Descrição
UNION	$R \cup S$	Operação binária envolvendo duas relações que retorna outra relação contendo as tuplas que estão presentes em ambas as relações de entrada ou em apenas uma das relações e sem repetição.
INTERSECTION	$R \cap S$	Operação binária envolvendo duas relações que retorna outra relação contendo apenas as tuplas que estão presentes em ambas as relações de entrada e sem repetição.
DIFFERENCE	$R - S$	Operação binária envolvendo duas relações que retorna outra relação contendo as tuplas presentes apenas na primeira relação da operação e sem repetição. De acordo com a expressão exemplo, são retornadas as tuplas presentes apenas na relação R.
CARTESIAN PRODUCT	$R \times S$	Operação binária envolvendo duas relações que produz um conjunto de tuplas. O conjunto resultante é formado pela combinação de cada tupla de uma relação com todas as tuplas da outra relação.
SELECT	$\sigma_{<\text{condição de seleção}>}(R)$	Permite selecionar um subconjunto de tuplas de uma relação que satisfaça uma condição de seleção.
PROJECT	$\pi_{<\text{lista de atributos}>}(R)$	Permite projetar atributos específicos de uma relação.
JOIN	$R \bowtie_{<\text{condição de junção}>} S$	Permite combinar duas relações e retornar um conjunto de tuplas destas relações que atendam a uma certa condição de junção.
ORDER	$\tau_{<\text{lista de atributos}>}(R)$	Permite ordenar as tuplas de uma relação de acordo com a lista de um ou mais atributos.

Tabela 1 – Descrição das operações relacionais.

Com respeito à operação de junção ($R \bowtie_{<\text{condição de junção}>} S$), descreve-se uma condição de junção geral da seguinte maneira:

$$< \text{condição} > \text{ AND } < \text{condição} > \text{ AND } \ldots \text{ AND } < \text{condição} > \tag{2.1}$$

Na expressão (2.1), cada termo <condição> é modelado como $r_i \; \theta \; s_i$, sendo r_i um atributo da relação R, s_i um atributo da relação S e θ um operador de comparação geral ($=$, $<$, $\leq$, $>$, $\geq$, $\neq$). Uma operação *JOIN* composta por uma condição de junção formada por θ é conhecida como *THETA JOIN*. Ainda em relação à operação *JOIN*, mais outras duas variações podem ser citadas: *EQUIJOIN* e *NATURAL JOIN*.

Na primeira, a condição de junção geral é composta por termos $<$condição$>$, onde o operador de comparação é sempre o de igualdade ($=$). A variação *NATURAL JOIN* não possui um predicado de junção de igualdade explícito, pois o predicado desta operação é implicitamente gerado a partir dos pares de atributos que possuam o mesmo nome em ambas as relações envolvidas. Assim, o resultado final da operação elimina um dos atributos presentes no par de junção, uma vez que eles possuem valores idênticos e um deles se torna descartável.

Ressalta-se que a operação *JOIN* foi tratada como binária em todas as citações anteriores, ou seja, a operação utilizou apenas duas relações como parâmetro de entrada (*two-way join*). Contudo, esta operação pode ser aplicada a mais de duas relações; neste caso, sendo denominada como *multiway-join*. De acordo com 2010a ELMASRI; NAVATHE;, a operação *multiway-join* aumenta ainda mais a complexidade do planejamento, pois o número de maneiras para se executar tal operação cresce rapidamente (as relações podem ser ordenadas de várias formas distintas para sua posterior análise na execução do operador). Na Seção 2.1 são apresentadas formas de se reduzir o espaço de soluções, sendo uma delas a adoção da representação das soluções limitada ao uso de junções apenas na forma binária.

As expressões de álgebra relacional podem ser categorizadas em dois tipos: atualização (inserção, modificação e remoção de informações) e consulta (seleção de informações). Este trabalho tem como foco as operações de consulta. Dito isso, uma consulta SQL pode ser mapeada para operações da álgebra relacional. É possível combinar as operações relacionais em uma expressão algébrica do tipo *SELECT-FROM-WHERE* (esta expressão também pode conter cláusulas do tipo *ORDER BY* e *GROUP BY-HAVING*). No trecho de código 2.1, é apresentada uma consulta para exemplificar a ideia de blocos. Neste exemplo, são recuperados todos os alunos de uma dada universidade que residem em Minas Gerais. É possível identificar dois blocos na consulta, sendo eles representados por comandos *SELECT* diferentes.

Listing 2.1 – Exemplo - Blocos Consulta SQL.

```
SELECT nome, cidade FROM tabela\_aluno WHERE cidade IN ( SELECT
    cidade FROM tabela\_uf WHERE uf='MG' )
```

A tradução do segundo bloco do Código 2.1 para uma expressão em álgebra relacional é exibida na Figura 4.

$$\pi_{cidade}(\sigma_{uf='MG'}(\text{tabela\_uf}))$$

Figura 4 – Exemplo - Expressão Relacional.

Uma possível representação para expressões de álgebra relacional é conhecida como árvore de consulta (*query tree*). Esta representação é dada por uma estrutura de árvore onde os nós internos são operadores de álgebra relacional e os nós folhas por relações de entrada. A Figura 5 exibe a expressão de álgebra relacional anterior representada como uma árvore de consulta.

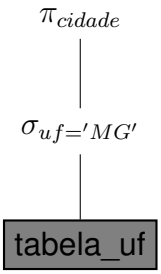


Figura 5 – Exemplo - Árvore de Consulta.

Como descrito em 2008c GARCIA-MOLINA; ULLMAN; WIDOM;, a representação inicial reconhecida pelo SGBDR gerada após o primeiro passo do planejamento (*Scanning*, *parsing* e validação), será tratada como plano de consulta lógico. Com base nisso, o planejamento procede repassando o plano de consulta lógico ao componente de otimização, que é responsável por otimizar este plano e selecionar a forma de acesso das relações, seja por meio de índices, acessos sequenciais, etc. Além disso, também é definida a ordem de execução de algumas operações, dentre elas: junção, interseção, etc. O produto final desta otimização é chamado de plano de consulta físico.

Após definir o plano de execução, o mesmo pode ser interpretado diretamente pelo processador de consultas ou o componente de geração de código pode pré-compilar e produzir um código para o plano gerado que possa ser executado posteriormente de forma nativa (ex.: *stored procedure* ou função escritas em C). Em ambas as opções, o processador de consultas será responsável por controlar e executar as ações definidas pelo plano de execução. Por fim, o resultado é retornado ao solicitante. A Figura 6 ilustra esse processo.

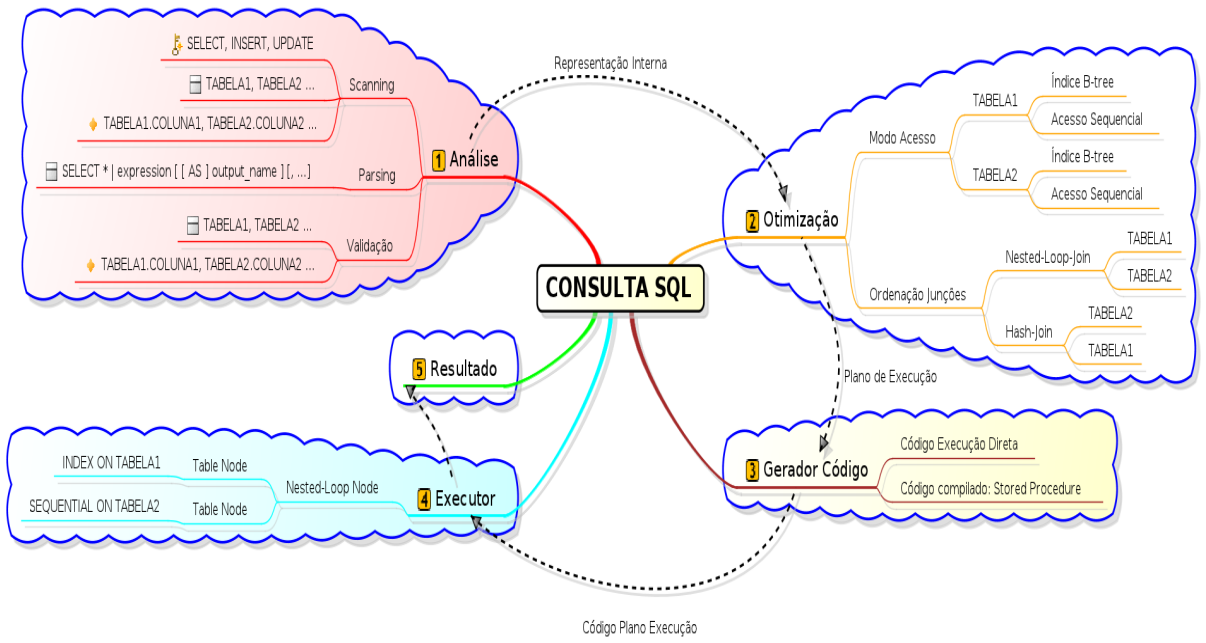


Figura 6 – Planejamento e processamento de uma consulta.

O otimizador da consultas é o componente no qual se concentra a atenção deste trabalho. A Seção 2.1 apresenta uma discussão mais abrangente sobre as tarefas realizadas por este componente. Um detalhamento sobre a linguagem SQL está disponível em 2010b ELMASRI; NAVATHE;2010c ELMASRI; NAVATHE;2008d GARCIA-MOLINA; ULLMAN; WIDOM;2008a GARCIA-MOLINA; ULLMAN; WIDOM;2008b GARCIA-MOLINA; ULLMAN; WIDOM;. Mais informações sobre o modelo relacional podem ser encontradas em 2010d ELMASRI; NAVATHE;2008e GARCIA-MOLINA; ULLMAN; WIDOM;. A álgebra relacional é coberta com detalhes em 2010e ELMASRI; NAVATHE;2008f GARCIA-MOLINA; ULLMAN; WIDOM;.

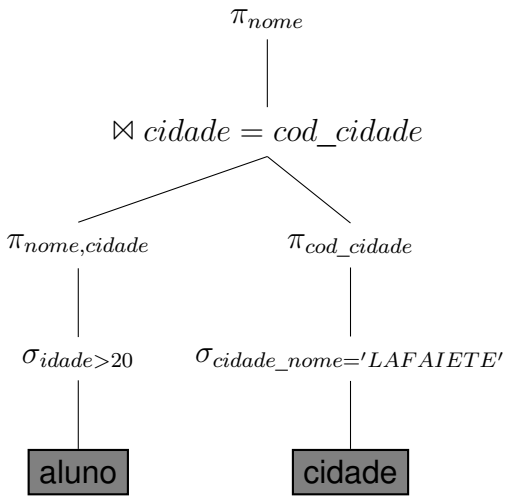
2.1 Otimizador de Consultas

O otimizador de consultas desempenha um papel fundamental em SGBDRs. Ele tem a responsabilidade de definir como e quando as operações inerentes a uma determinada consulta deverão ser executadas. O trabalho realizado pelo otimizador tem por objetivo gerar um plano de execução físico, o qual será executado posteriormente. Neste plano são especificados detalhes sobre a forma de acesso nas relações, a ordem de execução das operações, a forma como o resultado das operações será repassado aos demais nós de processamento, quais métodos serão utilizados para executar as operações da álgebra relacional. Conforme dito na Seção anterior, existem vários operadores da álgebra relacional que podem compor uma expressão relacional gerada a partir de uma consulta SQL. Contudo, este trabalho tem como foco

expressões de álgebra relacional do tipo *SELECT-PROJECT-JOIN* (Figura 7), ou seja, expressões compostas apenas por operações: *SELECT*, *PROJECT* e *JOIN*. Diante disto, serão discutidos na Subseção 2.1.1 possíveis implementações ou operadores físicos comumente utilizadas na prática para as operações de seleção, projeção e junção.

SELECT nome **FROM** aluno, cidade
WHERE cidade=cod_cidade AND
cidade_nome='LAFAIETE' AND
idade $>$ 20

(a) Consulta *SELECT-PROJECT-JOIN*



(b) Consulta *SELECT-PROJECT-JOIN*

Figura 7 – Representação Álgebra Relacional.

2.1.1 Algoritmos para Operações Relacionais

A seleção ($\sigma_{<\text{condição de seleção}>}(R)$) é uma das operações mais básicas da álgebra relacional. Ela consiste em varrer uma relação em busca de tuplas que atendam a uma determinada condição. Duas abordagens são comumente utilizadas na seleção de tuplas de uma relação, são elas:

- *Table-Scan* ou *File-Scan*: Uma relação pode ser organizada em vários blocos armazenados em um ou vários arquivos na memória secundária. Nesta abordagem, os blocos podem ser lidos um a um e as tuplas presentes nestes blocos que atenderem à condição de seleção, podem ser selecionadas. Esta pesquisa tende a ser mais cara computacionalmente.
- *Index-Scan*: Nos casos em que a condição de seleção envolver um ou mais atributos presentes em um índice (árvore-B, árvore-AVL, tabela *hash*, etc), uma pesquisa pelas tuplas pode ser realizada no índice em questão para selecionar

as devidas tuplas. Em determinadas situações, a pesquisa por índice pode resultar em um menor esforço computacional. Ainda em relação aos índices, existe a possibilidade de o arquivo físico de armazenamento das tuplas de uma relação ser organizado de acordo com um determinado índice (*clustering index*). Por consequência, a pesquisa nessa relação pode tirar proveito desta organização e realizar uma pesquisa com base na ordem dos dados.

Em 2010a ELMASRI; NAVATHE; são apresentados uma série de métodos básicos ou formas de acesso utilizados para implementar a operação de seleção:

- **Pesquisa Linear** - Recupera todas as tuplas da relação e testa cada uma delas com a condição de seleção;
- **Pesquisa Binária** - Dada uma relação organizada de forma ordenada segundo algum atributo chave, este método permite que uma pesquisa binária relativa ao atributo chave seja realizada em condições de igualdade, o que resulta em um esforço computacional menor que a pesquisa linear;
- **Pesquisa por atributo chave e uma tupla de retorno** - Se a condição de seleção envolver uma expressão de igualdade em um atributo chave de um índice primário, tal índice pode ser utilizado para recuperar apenas uma tupla. Uma outra opção é a utilização de uma tabela *hash* para recuperação da tupla. Neste caso, considera-se novamente uma condição de seleção como uma igualdade contendo o atributo chave armazenado na tabela *hash*, e assim, a chave *hash* (*hash key*) da tabela pode ser utilizada para retornar apenas uma tupla;
- **Pesquisa por atributo chave e múltiplas tuplas de retorno** - A pesquisa por atributo chave envolvendo uma ou mais expressões de desigualdade ($<$,$\leq$,$>$,$\geq$) também pode utilizar índices primários para recuperar várias tuplas;
- **Pesquisa em índice agrupado e múltiplas tuplas de retorno** - Neste tipo de pesquisa, a condição de seleção envolve uma igualdade com um atributo não-chave e um índice agrupado para este atributo. Assim, é possível utilizar o índice para recuperar todas as tuplas que satisfaçam a igualdade do predicado;
- **Pesquisa em índice secundário (árvore-B)** - O índice secundário pode ser aplicado em pesquisas com igualdades ou desigualdades. Desta maneira, o mesmo pode ser utilizado para recuperar uma única tupla trabalhando com atributos chave (valores únicos) ou múltiplas tuplas para atributos não chave.

A junção ($R \bowtie_{<\text{condição de junção}>} S$) é uma das operações mais custosas no processamento de consultas (ELMASRI; NAVATHE, 2010a). Ela permite combinar as tuplas de duas relações que satisfaçam a uma determinada condição de junção, o que

compõe o resultado da junção entre as relações (ELMASRI; NAVATHE, 2010a). Na literatura são encontrados vários algoritmos para esta operação, dentre eles:

- ***Nested-Loop Join*** - O algoritmo *nested-loop join* trabalha com laços de repetição aninhados. Dadas duas relações R (laço externo) e S (laço interno), o método analisará para cada tupla em R todas as tuplas de S, sendo selecionadas as tuplas que atenderem à condição de junção. Este algoritmo pode ser classificado como um método de força bruta e não possui limitações sobre o fato de as relações caberem ou não na memória principal. Existem variações que podem trabalhar com uma tupla por vez, ou de forma mais otimizada com blocos de tuplas. O uso de índices como forma de acesso nas relações de entrada pode contribuir para evitar uma leitura linear das relações durante avaliação da condição de junção;

- ***Merge-Sort Join*** - Este método depende da ordem das tuplas em relação aos atributos de junção nas relações de entrada. Durante a fase de otimização, os SGBDRs podem trabalhar com a opção de ordenação explícita dos dados, e então aplicar o método sobre eles. Nos casos em que as formas de acesso escolhidas para as relações de entrada proverem implicitamente uma ordem para tuplas (índices ordenados, registros ordenadas fisicamente, etc) nos atributos de junção, este algoritmo se torna uma opção extremamente eficiente para realizar a junção das relações. As tuplas de cada relação são lidas concorrentemente, sendo o apontador para tupla atual de cada relação incrementado na medida em que os valores mudam. A ordem prévia das tuplas nos atributos em questão permite ao método incrementar de forma segura o apontador das tuplas ou até mesmo parar a execução do método sem a necessidade de terminar a leitura dos dados;

- ***Hash Join*** - A implementação do operador de junção baseada em *hashing* para expressões de igualdade (EQUIJOIN), também é uma opção bastante eficiente em determinados casos. O funcionamento do método pode ser dividido em duas fases: partição (*partition*) e junção (*join*, *probing*), respectivamente. Durante a primeira fase, as relações R e S são distribuídas em N partições (*hash buckets*) $R_1, R_2 \ldots R_N$ e $S_1, S_2 \ldots S_N$. Observa-se que a mesma função de *hash* ou particionamento deve ser utilizada para a partição das duas relações com respeito aos atributos da condição de junção em questão. Consequentemente, tem-se a correspondência entre as partições R_i e S_i para algum i em relação à condição de junção. Durante a segunda fase, são necessárias N iterações para a junção entre os respectivos *hash buckets*. É comum trabalhar com partições de tamanho definido, e calcular o número de partições das relações em acordo

com a quantidade de memória disponível para o processamento. Assim, uma partição pode ter que armazenar informações além do previsto, implicando na necessidade de escrita de informações para a memória secundária e posterior recuperação desta informação. Em relação à função de *hashing*, espera-se que a mesma possa distribuir os dados de forma mais uniforme entre as partições, evitando que algumas tenham muito mais informações que outras.

Finalmente, a projeção de atributos de uma relação ($\pi_{<\text{lista de atributos}>}(R)$) permite extrair informações específicas de uma relação. O resultado da projeção define que não pode haver repetição de tuplas. Desta maneira, a projeção que utilize atributos chave de uma relação é direta, pois a repetição de chaves não é permitida em SGBDRs. Entretanto, quando a projeção trata atributos não chave, as mesmas técnicas de ordenação e *hashing* apresentadas anteriormente podem ser aplicadas. No caso da ordenação, basta ordenar as tuplas de acordo com a lista de atributos selecionados e numa segunda leitura eliminar as repetições. Já a técnica de *hashing*, trabalha com o particionamento da relação. As tuplas distribuídas para a mesma partição podem ser comparadas para evitar repetições.

2.1.2 Estimando o Custo dos Operadores

Durante o processo de otimização, várias soluções equivalentes com diferentes combinações de operações podem ser avaliadas. Cabe ao otimizador decidir qual plano escolher; portanto, é necessário estimar o custo de um plano de execução e selecionar aquele dito mais barato. Várias métricas podem ser utilizadas para medir o custo de um plano, as quais podem ser classificadas em vários grupos. Em 2010a ELMASRI; NAVATHE;, tais métricas são distribuídas como segue:

1. **Acesso à Memória Secundária** - Esta métrica está associada ao custo de se ler ou escrever informações entre a memória secundária e principal. O custo de *E/S* pode variar em consequência das formas de acesso disponíveis para uma dada relação (verificar abordagens apresentadas anteriormente para o operador de seleção). Além disso, pode-se citar um outro fator que relaciona-se às páginas de registros de uma relação estarem armazenados contiguamente (alguns SGBDRs permitem que as páginas contendo as tuplas das relações sejam agrupadas contiguamente[1]), e assim, evitar custos adicionais relacionados ao acesso à disco (tempo de *seek* em discos magnéticos, etc);

2. **Gravação na Memória Secundária** - Durante a execução de uma consulta, às vezes é necessário gravar o resultado das operações na memória secundária.

[1] O SGBDR H2 possui uma operação chamada *SHUTDOWN DEFRAG*, que desliga o SGBDR e organiza as páginas das tuplas de todas as relações de forma contígua.

Este custo pode ser devido à materialização do resultado de alguma operação de junção, gravação de resultados intermediários durante uma operação de ordenação que não pode ser realizada em sua totalidade na memória principal destinada à execução da consulta, construção uma tabela *hash* que não pode ser totalmente armazenada na memória principal destinada a tal execução, etc;

3. **Operações de CPU** - Este item se relaciona ao processamento de tuplas na memória principal ou custo de *CPU*. Tais operações dizem respeito à pesquisa por tuplas, ordenação de tuplas, operações inerentes aos valores dos campos das tuplas, mesclar tuplas devido a uma operação de junção, construir uma tabela *hash* em função de uma operação de junção ou função de agregação, etc;

4. **Uso de Memória Principal** - Sabe-se que a memória principal é finita; por consequência, os SGBDRs tendem a adotar uma estratégia de destinar uma quantidade limitada de memória principal para a execução de uma dada consulta[2]. Desta maneira, esta métrica pode influenciar diretamente na escolha dos métodos de junção, opções de ordenação, armazenamento de resultados intermediários na memória secundária, etc;

5. **Rede** - Em bancos de dados distribuídos, a transmissão de informações entre os nós de processamento também influencia o custo do plano gerado.

Planos de execução mais baratos podem resultar em um número menor de ações para a execução das consultas. Com relação às métricas apresentadas, pode-se observar uma maior atenção ao custo relacionado às operações de *E/S*. Segundo 2008c GARCIA-MOLINA; ULLMAN; WIDOM;, as operações de *E/S* influenciam dominantemente no tempo gasto para a realização de uma consulta. Contudo, vale notar os progressos alcançados por outros componentes, outrora considerados caros ou escassos, dentre eles: Memória *RAM*, Processadores e Meios de Armazenamento. À vista disso, 1999 AILAMAKI et al.; relaciona o custo de execução das consultas ao uso da memória principal e aos processadores. Ressaltam-se ainda situações já não tão incomuns, em que a quantidade de memória principal é suficiente para armazenar todos os dados do banco de dados (ou pelo menos as informações mais frequentemente utilizadas) e atender as requisições relacionadas. Por consequência, após um determinado período, espera-se que o acesso às informações das relações concentre-se na memória principal, o que permite focar as operações de *CPU*. Com base no que foi dito, algumas fórmulas de custo das operações focarão além das operações de *E/S*, as operações de *CPU* e o uso de memória principal.

[2] O SGBDR H2 define a quantidade de memória disponível para execução de uma consulta por meio dos parâmetros *MAX_MEMORY_ROWS* e *MAX_OPERATION_MEMORY*. Já o SGBDR PostgreSQL, utiliza a configuração *WORK_MEM* para este propósito.

Para entendimento dos cálculos de custo, as seguintes notações serão utilizadas:

- $\mathbf{T(REL)}$ - Representa o número de tuplas de uma relação ou cardinalidade da relação;
- $\mathbf{P(REL)}$ - Define o número de páginas ou blocos de uma relação com base no tamanho médio das tuplas;
- **PT(REL)** - Determina o número médio de tuplas por página;
- **LI(IDX)** - Caracteriza o número de níveis em um índice multinível;
- ***CPU*** - Coeficiente associado ao custo das operações de *CPU*;
- ***E/S*** - Coeficiente associado ao custo das operações na memória secundária. Para maior facilidade do entendimento das fórmulas de custo, este coeficiente foi simplificado. Entretanto, alguns SGBDRs tal como o PostgreSQL, dividem este coeficiente em pelo menos outros dois (relacionados ao custo de acesso sequencial e randômico de informações), para tentar refletir melhor o comportamento dos discos magnéticos;
- **MC** - Quantidade de memória principal disponível para execução de uma consulta;
- **CO** - Custo para ordenar um conjunto de tuplas de uma relação com base em algum atributo. Este custo é totalmente influenciado por **MC**, pois se o grupo de tuplas puder ser ordenado na memória principal com a quantidade de memória **MC** disponibilizada, um método mais eficiente pode ser aplicado. Caso contrário, o método deverá utilizar a memória secundária para realizar a ordenação de todas as tuplas (mais custoso). O SGBDR H2 utiliza um algoritmo baseado no *MergeSort* para ordenação em memória primária. Já para a ordenação em memória secundária, ele aplica um *MergeSort* apoiado em discos virtuais (*Virtual Disk Tape*), semelhante ao algoritmo apresentado por 1973 KNUTH; para o mesmo propósito. Finalmente, é relevante informar que a ordenação não terá custo algum, caso a relação avaliada possua algum índice ordenado ou seja ordenada fisicamente quanto ao atributo em questão;
- **HS** - Custo relacionado à leitura de uma relação para a construção de uma tabela *hash*, podendo ser resumido por: $T(REL) \times CPU$;
- **V(REL,ATR)** - Descreve o número de valores distintos do atributo ATR na relação REL. Vale ressaltar que manter esta informação sempre atualizada pode ser custoso ao sistema de banco de dados. Para contornar isto, alguns SGBDRs

adotam políticas de atualização desta informação na tentativa de evitar a sobrecarga do sistema[3]. Mais ainda, apoiado em conceitos do campo estatístico, é possível realizar inferências acerca de uma população com base em amostragens da mesma, isto é, o número de valores distintos de um atributo pode ser aproximado com base em uma amostra das tuplas (POOSALA et al., 1996).

A pesquisa por tuplas pode ser complexa e envolver uma série de condições de seleção relacionadas entre si. Uma expressão composta por várias condições de seleção pode ser classificada em três tipos:

- **Conjuntiva** - Este tipo de expressão relaciona todas as condições de seleção apenas por meio do conector lógico AND;
- **Disjuntiva** - Uma expressão deste tipo, utiliza o conector lógico OR para ligar as condições de seleção;
- **Híbrida** - Expressões compostas por predicados conectados pela combinação de conectores lógicos AND e OR.

A otimização da operação de seleção busca escolher uma forma de acesso em uma dada relação que minimize o número de tuplas avaliadas/retornadas e seja a mais eficiente dentre as possibilidades de acesso disponíveis. Com respeito à otimização da seleção para expressões no padrão disjuntivo, ressalta-se uma maior dificuldade de otimização no que diz respeito à minimização do número de tuplas avaliadas/retornadas. Isto se deve ao fato de que as tuplas que satisfazem a apenas uma das condições na expressão podem ser selecionadas, ou seja, o resultado final é composto pela união das tuplas que atendam às condições individuais da expressão. Ademais, pode ser destacado na análise do custo de uma seleção um parâmetro denominado seletividade do predicado de seleção. Tal parâmetro permite estimar o número esperado de tuplas de uma dada relação que atenderão a uma condição em questão. O valor do parâmetro é um número pertencente ao intervalo [0,1], onde 0 indica o retorno esperado de nenhuma tupla e 1 o retorno de todas as tuplas.

Com base em 2008c GARCIA-MOLINA; ULLMAN; WIDOM;, a seletividade de uma condição de seleção ($\sigma_{<ATR=a>}(R)$) envolvendo uma expressão de igualdade entre um atributo ATR e um valor a constante pertencente a algum domínio (os valores de ATR e a são tratados dentro do mesmo domínio) pode ser estimada por:

[3] Por padrão, a cada 2000 alterações em uma dada relação, o SGBDR H2 atualiza as informações sobre o número de valores distintos dos atributos com base em uma amostra de 10000 tuplas. Por outro lado, os SGBDR PostgreSQL possui um mecanismo denominado *Automatic Vacuuming*, que atua automaticamente na limpeza e atualização das informações estatísticas de modo a não sobrecarregar o sistema.

$$sl = \frac{(\frac{T(REL)}{V(REL,ATR)})}{T(REL)} \quad ou \quad \frac{1}{V(REL,ATR)} \tag{2.2}$$

O número de tuplas esperadas no resultado da seleção é dada por:

$$TE(REL) = T(REL) \times sl \tag{2.3}$$

A seletividade de predicados de igualdade envolvendo atributos chave (atributos que compõem a chave-primária da relação) pode ser definida como $\frac{1}{T(REL)}$, visto que a quantidade de valores distintos é igual ao número de tuplas, logo: $T(REL) \div V(REL, ATR) = 1$. De acordo com 2010a ELMASRI; NAVATHE;, a estimativa da seletividade de condições de igualdade com atributos não-chave pode assumir que as tuplas são distribuídas uniformemente entre os valores distintos dos atributos relacionados. Vale ressaltar que a estimativa do número de tuplas não é a prova de erros, e tem como principal objetivo, manter o cálculo de custo das operações relacionadas coerente com a realidade durante a execução da consulta.

Alguns SGBDRs adotam ainda outros tipos de recursos para melhorar a precisão de estimativas. Um exemplo notável é o uso de histogramas para armazenar informações acerca da distribuição dos dados. Nos casos em que os pressupostos de linearidade e uniformidade dos dados não são atendidos, o que reflete melhor as situações reais, o uso de histogramas é uma ferramenta útil para computar inferências mais precisas (PIATETSKY-SHAPIRO; CONNELL, 1984; POOSALA et al., 1996). Histogramas podem ser aplicados na realização de predições sobre o número de tuplas esperadas no resultado de uma dada operação para alguns tipos de condição, especialmente em casos que envolvem predicados de desigualdade entre atributos de uma relação e valores constantes de algum domínio (exemplos de cálculo de seletividade de predicados utilizando histogramas em SGBDRs estão disponíveis em 2010 LIMITED;). Um histograma pode conter a distribuição de frequência dos valores de um dado atributo, valores mais frequentes do atributo, etc. Além disso, como último recurso, alguns SGBDRs utilizam valores constantes para definir a seletividade de predicados nos diferentes tipos de operação (SELINGER et al., 1979).

A seguir, baseado em 2010a ELMASRI; NAVATHE;, são apresentadas algumas fórmulas de custo do operador de seleção ($CS(REL)$) para diferentes algoritmos. Vale frisar que o custo relacionado à leitura/escrita na memória secundária será medida em número de páginas, e que o custo de operações de *CPU* será medido pelo número de tuplas processadas/avaliadas em memória. Por fim, é importante observar que dependendo do hardware disponível, os coeficientes *CPU* e *E/S* podem ser ajustados para refletir melhor o ambiente de execução.

- **Pesquisa Linear** - Nesta pesquisa todas as páginas são acessadas sequencialmente; portanto, o custo é $CS = P(REL) \times E/S + TE(REL) \times CPU$. Para condições de igualdade com atributos chave, pode-se estimar em média a avaliação de metade das tuplas da relação $CS = P(REL)/2 \times E/S + TE(REL) \times CPU$;

- **Pesquisa Binária** - Este tipo de pesquisa permite estimar o custo como $CS = (\log_2 P(REL) + [TE(REL)/PT(REL)] - 1) \times E/S + TE(REL) \times CPU$. O custo pode ser reduzido a $CS = \log_2 P(REL) \times E/S + TE(REL) \times CPU$ nas condições de igualdade em atributos chave, pois o número de tuplas esperadas é igual a 1;

- **Pesquisa por atributo chave e uma tupla de retorno** - Em índices primários o custo é $CS = (LI(IDX) + 1) \times E/S + CPU$, sendo uma página para cada nível do índice e mais uma página para recuperação do dado. Em índices com tabelas *hash*, o custo cai para aproximadamente $CS = E/S + CPU$;

- **Pesquisa em índice ordenado e múltiplas tuplas de retorno** - O custo para pesquisa envolvendo condições com operações de desigualdade ($<$,$\leq$,$>$,$\geq$), pode ser estimado pela soma do número de níveis do índice mais uma previsão de avaliação de metade das tuplas da relação $CS = (LI(IDX) + P(REL)/2) \times E/S + TE(REL) \times CPU$;

- **Pesquisa em índice secundário (árvore-B)** - Em índices do tipo árvore-B, a pesquisa com atributos únicos implica em $CS = (LI(IDX)+1) \times E/S + CPU$. Já a pesquisa com atributos não necessariamente únicos, o resultado no pior caso é dado por $CS = (LI(IDX) + TE(REL) + 1) \times E/S + TE(REL) \times CPU$, onde são lidas as páginas do níveis do índice, uma página diferente para cada tupla esperada e uma página com os ponteiros dos registros. Por fim, em condições de desigualdade, 2010a ELMASRI; NAVATHE; estima o custo grosseiramente como $CS = (LI(IDX) + T_{P_1}(REL)/2 + T(REL)/2) \times E/S + (T_{P_1}(REL)/2 + T(REL)/2) \times CPU$, o qual é formado pelas somas dos níveis do índice, avaliação de metade das tuplas da página no primeiro nível do índice e a recuperação de metade das tuplas da relação.

A operação de seleção pode ser composta por uma expressão do tipo conjuntiva. Assim, uma forma de lidar com as várias condições de seleção, é utilizar primeiro a condição mais restritiva e eficiente, ou seja, aquela que resulte em um menor número de tuplas e o método de acesso mais eficiente. Após isso, as demais condições podem ser validadas com base nas tuplas retornadas com a primeira condição.

Tal como na seleção, a operação de junção será estudada apenas para expressões de igualdade e o seu custo será medido pelo número de operações de *CPU* e

E/S relacionadas e a quantidade de memória principal disponível para o processamento. Desta maneira, cabe observar que a seletividade de um predicado de junção (*join selectivity*) também é uma propriedade importante na avaliação do custo da operação, pois com este parâmetro é possível estimar o número tuplas esperadas que irão compor o resultado inerente a uma junção. Antes de apresentar as fórmulas dos custos, é necessário ressaltar uma dificuldade natural em se determinar a quantidade de pares de tuplas que satisfazem a uma determinada condição de junção, isto é, o quão difícil é se estimar a seletividade de um predicado de junção. O fato de não se saber como os conjuntos de valores dos atributos das relações se conectam, implica na possibilidade de diferentes resultados para a junção de tais relações. Em 2008c GARCIA-MOLINA; ULLMAN; WIDOM; são destacadas as seguintes situações:

1. Duas relações R e S possuem conjuntos de valores disjuntos entre os atributos relacionados, o que implica em $TE(R \bowtie S) = 0$;

2. Um atributo ATR é chave-primária da relação S e corresponde a uma chave-estrangeira em R, logo, cada tupla de R combina com exatamente uma tupla de S, resultando em $TE(R \bowtie S) = T(R)$;

3. O conjunto de valores dos atributos envolvidos na junção das R e S é quase o mesmo. Ademais, a junção entre essas relações pode ser aproximada por $TE(R \bowtie S) = T(R) \times T(S)$.

Como dito anteriormente, variadas situações podem emergir de uma junção. Contudo, optou-se pela mesma abordagem apresentada em 2008c GARCIA-MOLINA; ULLMAN; WIDOM;, onde se foca as situações mais comuns, e para tal, são assumidas como verdadeiras as seguintes suposições:

1. **Conjunto de valores auto-contidos** - Se duas relações R e S possuem um atributo em comum ATR e $V(R, ATR) \leq V(S, ATR)$, então cada valor do atributo de R estará presente em S. Mais ainda, é observado em 2008c GARCIA-MOLINA; ULLMAN; WIDOM; que tal pressuposto pode ser violado, mas que ele se mantém em situações como a do item 2 na listagem anterior e que também pode ser verdadeiro em outras circunstâncias, onde a relação S possui muitos valores para ATR, o que acarreta a possibilidade de que os valores de R estejam presentes em S;

2. **Preservação do conjunto de valores** - Se um dado atributo ATR está presente em uma relação R mas não em S, o resultado da junção entre as relações considerando algum predicado de junção com outro atributo para S não implicará na perda do conjunto de valores possíveis para a junção, ou seja,

$V(R \bowtie S, ATR) = V(R, ATR)$. De acordo com 2010a ELMASRI; NAVATHE;, este pressuposto também pode ser violado (item 1 da listagem anterior), mas é satisfeito em situações como descrito no item 2 da listagem anterior.

Com base nas propriedades anteriores, a seletividade em condições de junção envolvendo operações de igualdade pode ser estimada como segue:

$$jsl = \frac{1}{max(V(REL1, ATR), V(REL2, ATR))} \tag{2.4}$$

O número de tuplas esperadas pela condição de junção é dado por:

$$JTE(REL1, REL2) = T(REL1) \quad \times \quad T(REL2) \quad \times \quad jsl \tag{2.5}$$

A seguir, são apresentadas as fórmulas de custo (CJ) inerentes aos métodos de junção comumente encontrados na literatura. Além disso, é importante salientar que os cálculos basearam-se nos trabalhos de 1983 KITSUREGAWA; TANAKA; MOTO-OKA;2008c GARCIA-MOLINA; ULLMAN; WIDOM;2010a ELMASRI; NAVATHE; e no modelo de custo do SGBDR PostgreSQL[4]:

- ***Nested-Loop Join*** - Neste método não há a preocupação quanto à memória principal disponível, pois as páginas relacionadas podem ser acessadas na medida em que forem necessárias e as tuplas processadas uma a uma. Assim, definiu-se o custo da seguinte forma: $CJ = CS(REL1) + T(REL1) \times CS(REL2) + JTE(REL1, REL2) \times CPU$;

- ***Merge-Sort Join*** - Diferente do método anterior, o *Merge-Sort Join* é influenciado pela quantidade de memória disponível para uma possível ordenação das tuplas das relações envolvidas na junção. Dito isso, o custo pode ser descrito como segue: $CJ = CS(CO(REL1, ATR1)) + CS(CO(REL2, ATR2)) + JTE(REL1, REL2) \times CPU$. Vale dizer que o custo de ordenação das relações pode ser nulo, o que torna este método atraente em tais situações;

- ***Hash Join*** - Caso a quantidade de memória **MC** disponível seja suficiente para a construção da tabela *hash* da relação $REL2$, tal construção é realizada e as tuplas da relação $REL1$ podem ter os valores do atributo relacionado convertidos iterativamente como mesma função de *hash* durante a fase de junção ou *probing*. Neste cenário, o custo da junção pode ser estimado pela seguinte função: $CJ = CS(REL1) + HS(REL1) + CS(REL2) + HS(REL2) + JTE(REL1, REL2) \times$

[4] Cálculos de Custo disponíveis em src/backend/optimizer/path/costsize.c

CPU. Caso não haja memória suficiente para comportar a tabela *hash* da relação $REL2$, a estratégia apresentada na Subseção 2.1.1 pode ser aplicada. Assim, o custo neste caso pode ser simplificado por: $CJ = 3 \times (CS(REL1) + CS(REL2)) + HS(REL1) + HS(REL2) + JTE(REL1, REL2) \times CPU$, onde considerou-se 3 conjuntos de operações de entrada e saída, 1 leitura de páginas durante a partição, 1 escrita das partições no disco e 1 leitura de páginas para a fase de junção (*probing*).

2.1.3 Novas Tecnologias de Armazenamento

Esta subseção tem como objetivo apenas introduzir alguns conceitos relacionados às operações de seleção e junção em novas tecnologias de armazenamento, tais como: *Flash Drives* e *Solid State Drives* (SSD). É fato que várias pesquisas têm seu foco em modelos de custo compatíveis com discos magnéticos. Contudo, meios de armazenamento como SSD têm se tornado populares, resultando em contribuições relacionadas a novos métodos de seleção e junção em tais meios de armazenamento (SHAH et al., 2008; TSIROGIANNIS et al., 2009).

Os métodos de seleção e junção discutidos em 2008 SHAH et al.;2009 TSIROGIANNIS et al.; fazem uso de um modelo de organização de arquivos baseado em uma metodologia denominada DSM (*Decomposition Storage Model* (COPELAND; KHOSHAFIAN, 1985)). Tal metodologia tende a minimizar as operações de *E/S* e otimizar o uso da memória principal e da CPU em alguns tipos de consulta (AILAMAKI; DEWITT; HILL, 2002; GRAEFE, 2007; ZUKOWSKI; NES; BONCZ, 2008). Um modelo de organização de tuplas que merece destaque é o PAX (*Partition Attributes Across* (AILAMAKI; DEWITT; HILL, 2002)), exemplificado na Figura 8 (Figura retirada de 2002 AILAMAKI; DEWITT; HILL;). Neste modelo, as colunas das tuplas são distribuídas verticalmente entre as chamadas mini-páginas. Cada mini-página possui os valores de uma dada coluna armazenados contiguamente. Na proposta original do esquema PAX, os autores focaram a pesquisa na otimização do uso do *cache* da CPU durante leitura de um subconjunto das colunas.

Com o advento de novos meios de armazenamento, observou-se grandes avanços no tempo de *seek* do disco. Tal característica levou a exploração de novas estratégias, onde se adia a projeção de colunas necessárias no resultado final até o último momento possível, mantendo a memória principal menos carregada e otimizando o uso do *cache* da *CPU*. Além disso, tal metodologia também permite projetar para as operações intermediárias de junção, apenas as colunas relacionadas ao predicado de junção. A estratégia descrita anteriormente foi igualmente aplicada por 2008 SHAH et al.;2009 TSIROGIANNIS et al.;. Diante disto, decidiu-se por apresentar, a seguir, apenas o trabalho de 2009 TSIROGIANNIS et al.;, uma vez que os autores deram maiores

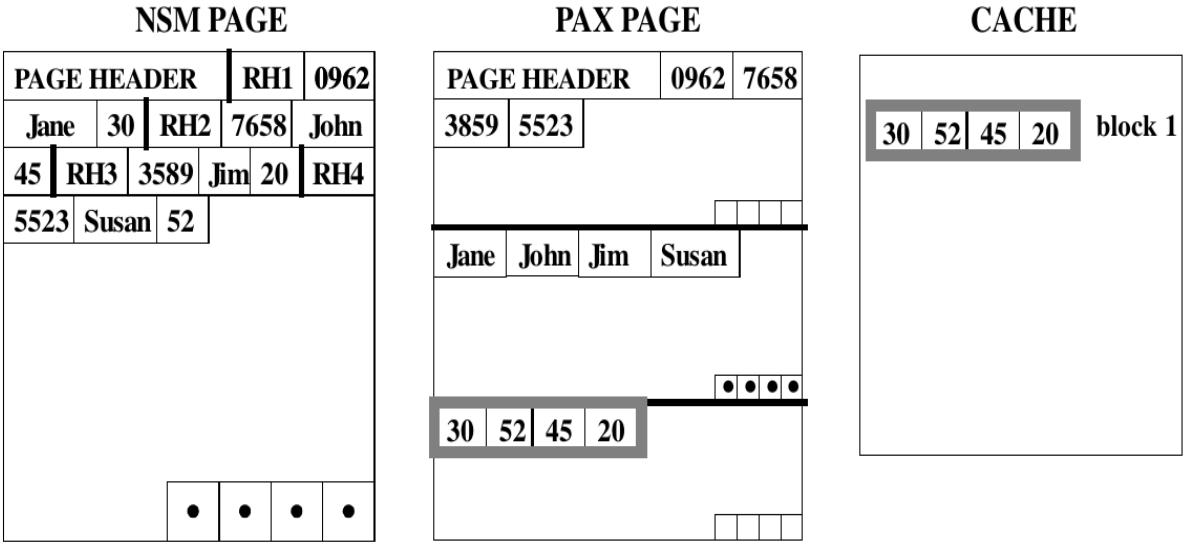


Figura 8 – *Partition Attributes Across* – PAX.

detalhes acerca dos métodos de seleção e junção, além de tê-los implementados no SGBDR PostgreSQL.

- **Seleção** - Foi implementado um método denominado *FlashScan*, cuja estratégia é ler nas páginas das tuplas (organizadas no modelo PAX) apenas as informações de determinadas colunas. Diferentemente do modelo NSM (*N-ary Storage Model*, mais informações em 2002 AILAMAKI; DEWITT; HILL;), que irá ler todo o conteúdo da página, independentemente das colunas projetadas, o operador *FlashScan* foi capaz de reduzir sensivelmente o tempo de seleção em experimentos com diferentes valores de seletividade e número de colunas projetadas;

- **Junção** - Para usufruir de vantagens relacionadas à rapidez nas leituras randômicas dos dispositivos SSDs, foi proposto um operador de junção nomeado como *FlashJoin*. Este operador faz uso da mesma metodologia empregada pelo *FlashScan*, onde se evita ler atributos desnecessários para a junção. Ressalta-se ainda o uso de uma estratégia chamada de *late materialization*, na qual são projetadas para as outras operações de junção, apenas as colunas necessárias para tal operação, adiando para a última operação, projetar os atributos necessários para o resultado. Os resultados obtidos mostraram um grande ganho no que diz respeito ao tempo de execução e ao uso de memória principal em várias situações de teste.

Para concluir, os autores argumentam em 2009 TSIROGIANNIS et al.;, que ainda há uma limitação quanto ao uso dos operadores propostos em situação gerais. Isto se deve à incompatibilidade dos operadores com outros componentes importan-

tes do SGBDR, como por exemplo, o mecanismo de MVCC. Um outro ponto extremamente importante, é o fato de que nenhuma alteração foi necessária no otimizador de consultas (componente responsável pela ordenação de junções) para utilizar os novos operadores. Essa característica é devida à abstração utilizada pelos otimizadores no *PostgreSQL*, que delegam para o componente de custo a tarefa de determinar a melhor forma de acesso e o método de junção associados à ordem definida para as relações. Tal aspecto corrobora a escolha do SGBDR H2, pois não altera o foco principal do trabalho (avaliação da qualidade do otimizador de junções), uma vez que seus otimizadores também se apoiam em um componente de custo para a definição dos métodos de acesso e junção.

2.2 Problema de Ordenação de Junções

O otimizador de consultas desempenha várias tarefas até a entrega do plano de consulta físico final. Foram discutidas até agora as escolhas individuais de métodos para as operações que compõem um plano de execução. Ressalta-se que um plano pode ser composto por uma série de operações relacionais binárias: interseção, união, junção, etc. Diante de tal situação, é necessário definir uma ordem para execução das operações, de forma que o resultado de uma operação seja composta pelo menor número possível de tuplas e executada da maneira mais eficiente. O resultado de uma operação será utilizado como entrada para a próxima operação, e assim por diante até o resultado final.

Como o título da subseção sugere, será tratado aqui um problema que envolve a definição de uma sequência ordenada de operações de junção. Sabe-se que os operadores de álgebra relacional possuem propriedades comutativas ou também associativas. De posse desta informação, é possível gerar várias expressões relacionais diferentes e equivalentes. Em relação à operação de junção, a Tabela 2 apresenta algumas propriedades:

Propriedade	**Definição**
Comutatividade de $\bowtie$ e $\times$	$R1 \bowtie_c R2 \equiv R2 \bowtie_c R1$ $R1 \times R2 \equiv R2 \times R1$
Comutatividade de σ com $\bowtie$ ou $\times$	$\sigma_{c_{R1}}(R1 \bowtie R2) \equiv (\sigma_{c_{R1}}(R1)) \bowtie R2$ $\sigma_{c_{R1}ANDc_{R2}}(R1 \bowtie R2) \equiv (\sigma_{c_{R1}}(R1)) \bowtie (\sigma_{c_{R2}}(R2))$
Comutatividade de π com $\bowtie$ ou $\times$	$\pi_{A_{R1},A_{R2}}(R1 \bowtie_c R2) \equiv (\pi_{A_{R1}}(R1)) \bowtie_c (\pi_{A_{R2}}(R2))$
Associatividade de $\bowtie$ ou $\times$	$(R1 \bowtie_c R2) \bowtie_c R3 \equiv R1 \bowtie_c (R2 \bowtie_c R3)$
Conversão de uma expressão $(\sigma, \times)$ em $\bowtie$	$\sigma_c(R1 \times R2) \equiv R1 \bowtie_c R2$

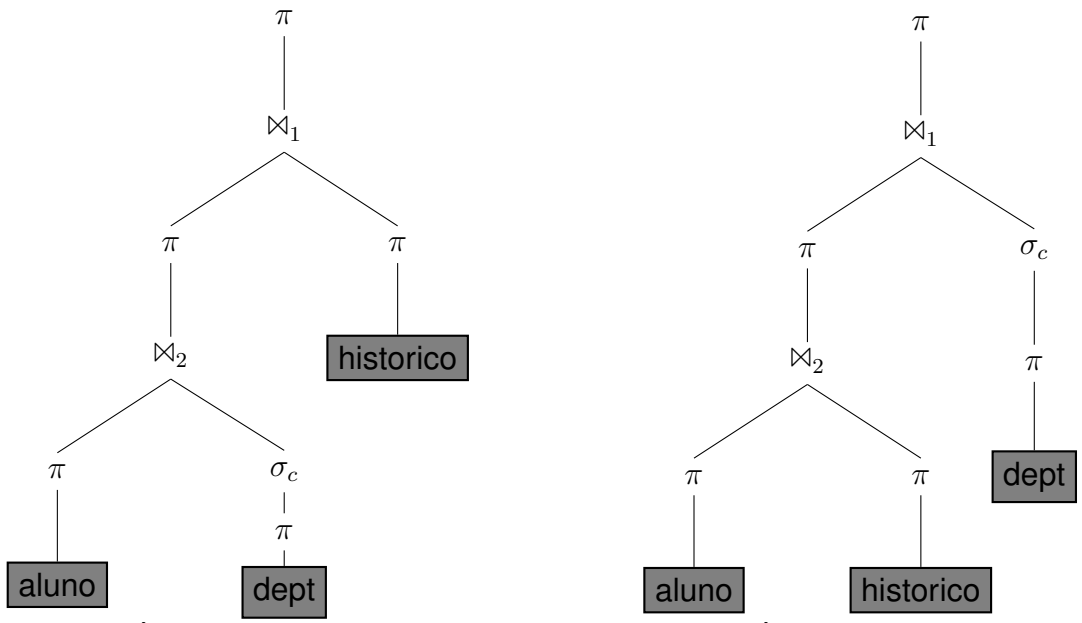
Tabela 2 – Propriedades do operador $\bowtie$

Para as árvores de consulta apresentadas a seguir, considere a seguinte consulta SQL para seleção do histórico de disciplinas de um aluno em um dado departamento.

Listing 2.2 – Exemplo - Consulta SQL.

```
SELECT nome, disciplina, nota FROM aluno, historico, dept WHERE
    mtr_aluno=mtr_historico AND dpt_aluno=dpt_dept AND
  dpt_historico=dpt_dept AND dpt_dept='DECOM'
```

A Figura 9 exibe duas árvores de consulta equivalentes. Observa-se, na árvore 9a, que os alunos são mesclados primeiro com departamento, sendo retornado os alunos de um determinado departamento. Somente após isso o histórico de disciplinas dos alunos é recuperado. Já na árvore 9b primeiro recupera-se o histórico de todos os alunos, para na segunda junção selecionar-se apenas o histórico de um departamento específico. Este exemplo mostra o uso das propriedades do operador de junção na definição de diferentes planos equivalentes.



(a) Árvore de Consulta 1 (b) Árvore de Consulta 2

Figura 9 – Exemplo - Árvores de Consulta Equivalentes.

A ordenação de junções trata-se de um problema de otimização combinatória complexo. O espaço de soluções deste problema pode ser imenso, chegando a ser inviável, em determinadas situações, explorá-lo exaustivamente com vistas à obtenção de uma solução ótima. De acordo com 1996 IOANNIDIS;, este espaço pode ser dividido em dois módulos:

- Espaço Algébrico;
- Espaço de Estrutura e Métodos.

O espaço algébrico se refere ao assunto discutido nesta subseção, pois trata a ordem de execução das operações de álgebra relacional presentes na consulta. Já o espaço de estrutura e métodos, determina as escolhas de implementação disponíveis no SGBDR em questão. Esta escolha é relativa ao assunto abordado na Subseção 2.1.1. As maneiras de se acessar uma relação combinada com os algoritmos de junção definem uma série de possibilidades para a ordenação das junções, o que define um problema NP-Completo no que diz respeito a procura pela solução ótima (IBARAKI; KAMEDA, 1984). O espaço de soluções para uma consulta envolvendo N junções, ou seja, uma consulta envolvendo $N+1$ relações, é composto por $\binom{2N}{N} N!$ soluções possíveis (SWAMI; GUPTA, 1988). A Tabela 3 exibe a quantidade de soluções em consultas com 1 a 10 relações.

Número Relações	Número Soluções
1	1
2	2
3	12
4	120
5	1.680
6	30.240
7	665.280
8	17.297.280
9	518.918.400
10	17.643.225.600

Tabela 3 – Espaço de Soluções.

Em consequência do alto grau combinatório de soluções no espaço algébrico, algumas restrições e heurísticas podem ser adotadas para restringir o espaço de busca, sendo algumas delas (IOANNIDIS, 1996; GARCIA-MOLINA; ULLMAN; WIDOM, 2008c):

1. Não considerar seleções e projeções separadamente, seleções devem ser aplicadas durante a leitura da relação e as projeções sobre o resultado de outras operações ;
2. Posicionar as operações de seleção o quanto antes na árvore de consulta . Segundo 1996 IOANNIDIS;2008c GARCIA-MOLINA; ULLMAN; WIDOM;, tal heurística se destaca como uma das mais importantes descritas aqui;
3. Posicionar as operações de projeção o quanto antes na árvore de consulta de forma a se reduzir o tamanho das tuplas dos resultados intermediários;
4. Não permitir a formação de planos cartesianos, isto é, soluções compostas por operações junção entre relações que não possuam um predicado de junção explícito/implícito. As relações devem sempre ser combinadas através de predicados de junção.
5. Os nós interiores da árvore de consulta devem sempre ser relações básicas.

Uma forma de representação do espaço de soluções que pode atender a algumas das restrições e heurísticas citadas é a árvore de junções binária. Nesta representação, os nós folhas são relações básicas e os nós internos, operações de álgebra relacional. Este tipo de representação utiliza operadores de junção apenas na forma binária, o que corrobora todas as citações anteriores onde o operador de

junção foi tratado como uma operação binária. Na Figura 10 são exibidas 4 representações comumente encontradas na literatura (SWAMI; GUPTA, 1988; IOANNIDIS, 1996; GARCIA-MOLINA; ULLMAN; WIDOM, 2008c; ÖZSU; VALDURIEZ, 2011):

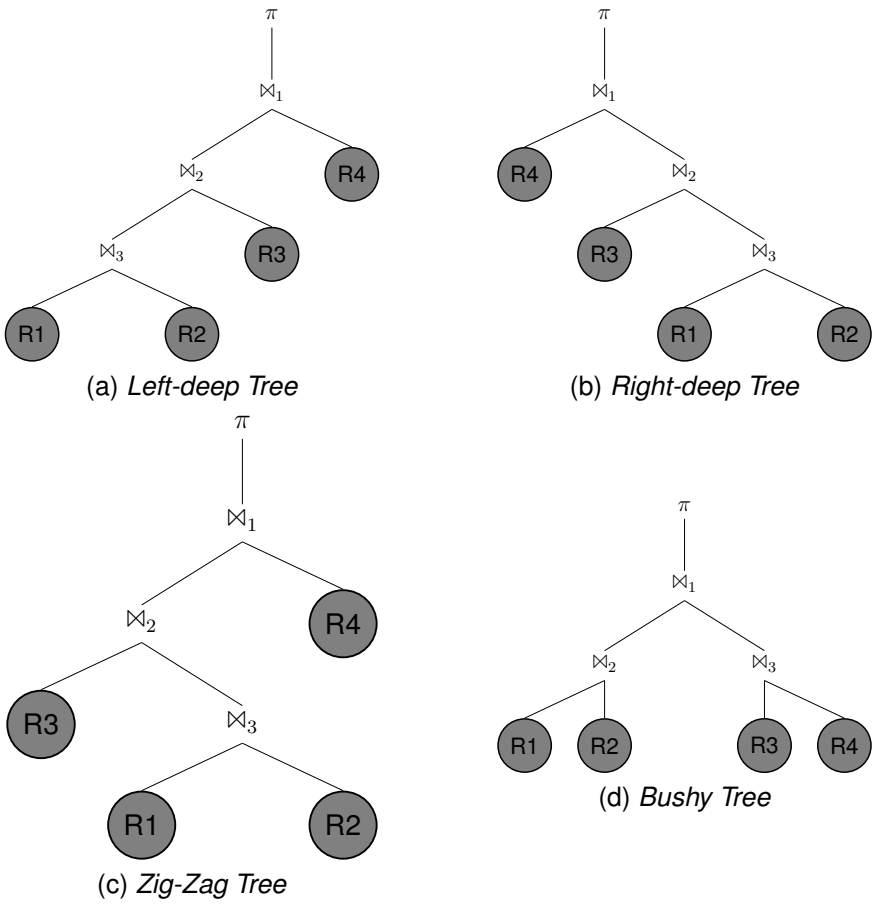


Figura 10 – Árvores Binárias de Junção.

Ressalta-se que o uso da representação árvore em profundidade à esquerda (*left-deep tree*) é a mais comum em sistemas de banco de dados relacionais. Inclusive, o SGBDR adotado neste trabalho utiliza esta representação. Uma vantagem da representação árvore em profundidade à esquerda, é a possibilidade de se utilizar a técnica de *pipeline* durante o processamento do plano de consulta, isto é, eliminar a necessidade de se executar por completo a operação de junção e armazenar o resultado possivelmente em memória secundária. Nestas representações, as tuplas resultantes de operações mais internas podem ser processadas uma por vez pelos nós mais externos sem a necessidade de armazenamento na memória secundária.

Observa-se, ainda, que a árvore fechada (*bushy tree*) possui uma característica natural que permite a paralelização da execução das operações de junções. Contudo, o espaço de soluções desta representação é maior do que em *left-deep tree* e *right-deep tree*. Embora as representações apresentadas reduzam o espaço de soluções, o problema ainda continua complexo, sendo inviável a localização de um ótimo global

dependendo do número de junções envolvidas. Na tabela a seguir, são apresentadas algumas fórmulas com a quantidade mínima e máxima de soluções no espaço de busca das representações discutidas (LANZELOTTE; VALDURIEZ; ZAÏT, 1993).

Tipo Árvore	Número Mínimo Soluções	Número Máximo Soluções
Left-deep e *Right-deep*	2^{N-1}	$N!$
Zig-Zag	$2^{N-2} \times 2^{N-1}$	$2^{N-2} \times N!$
Bushy	$\frac{2^{N-1}}{N} \times \binom{2N-2}{N-1}$	$\frac{(2N-2)!}{(N-1)!}$

Tabela 4 – Quantidade de soluções para N relações.

Embora o problema em questão seja abordado apenas no âmbito de bancos de dados relacionais, a ordenação de junções não é restrita apenas ao modelo relacional. Um exemplo que traduz tal observação é o campo de otimização com modelos de dados do tipo RDF (*Resource Description Framework* (KLYNE; CARROLL, 2004)), onde um típico cenário é composto por várias fontes de dados heterogêneas interconectadas e distribuídas pela rede. O problema de ordenação de junções aparece quando uma consulta RDF (PRUD'HOMMEAUX; SEABORNE, 2008) necessita combinar informações de várias fontes. A operação de junção neste contexto é similar a de bancos de dados relacionais, e consequentemente, impacta o tempo de resposta das consultas (mais informações estão disponíveis em 2004 KLYNE; CARROLL;2008 PRUD'HOMMEAUX; SEABORNE;).

2.3 Técnicas de otimização

As técnicas apresentadas a seguir visam a escolher um plano que minimize o número de operações referentes às métricas aplicadas na definição do custo de execução de um plano. A todo plano de consulta físico pode ser atribuído um custo estimado, o qual é calculado por meio das fórmulas apresentadas na Subseção 2.1.2. Por exemplo, levando-se em conta apenas o método de junção *nested-loop* e um plano de execução representado pela árvore *left-deep tree* (Figura 10a), o custo pode ser calculado de forma simplificada como segue:

$$TOTAL = CJ(\bowtie_1) + CJ(\bowtie_2) + CJ(\bowtie_3) \tag{2.6}$$

em que $CJ(\bowtie)$ estima o custo da operação de junção de acordo com a Subseção 2.1.2. A operação $\bowtie_3$ utiliza informações das relações $R1$ e $R2$, logo seu custo é dado por $CJ(\bowtie_3) = CS(R1) + T(R1) \times CS(R2) + JTE(R1, R2) \times CPU$, e o número estimado de tuplas resultante da junção é $JTE(R1, R2)$. O custo da operação $\bowtie_2$ é dado por $CJ(\bowtie_2) = CS(JTE(R1, R2)) + T(JTE(R1, R2)) \times CS(R3) + JTE(JTE(R1, R2), R3) \times CPU$ e o resultado estimado por $JTE(JTE(R1, R2), R3)$.

De forma análoga, o custo de $\bowtie_1$ pode ser calculado, o que resulta no custo final do plano.

A otimização do problema de ordenação de junções pode utilizar técnicas categorizadas em dois tipos: exaustivas (algoritmos exatos) e não-exaustivas (algoritmos não exatos). Algoritmos exaustivos garantem o ótimo global, ao passo que as técnicas não-exaustivas não dão esta garantia. O problema de ordenação de junções apresenta uma dificuldade natural em se expressar analiticamente a função objetivo e as restrições relacionadas. Consequentemente, o uso de heurísticas é indicado em tais situações. Mais ainda, é importante observar que um problema envolvendo uma consulta com 10 ou mais relações pode ser classificado como um problema de ordenação com muitas junções (SWAMI; GUPTA, 1988), o que possivelmente inviabiliza o uso de algoritmos exaustivos devido à natureza combinatória complexa do problema. Finalmente, vale lembrar que os experimentos computacionais deste trabalho focam os problemas com muitas junções.

A Subseção 2.3.1 descreve alguns algoritmos exatos, enquanto que a Subseção 2.3.2 apresenta algoritmos não-exatos aplicados ao problema em foco. Ressalta-se que várias outras metodologias podem ser aplicadas para se resolver este problema. Contudo, o intuito desta seção é apresentar apenas alguns algoritmos comumente citados na literatura.

2.3.1 Algoritmos Exaustivos

Algoritmos exaustivos podem considerar todas as possibilidades de soluções, ou seja, todas as permutações possíveis de junção entre as relações. Existem algumas técnicas que são mais inteligentes neste sentido, pois permitem uma exploração mais otimizada do espaço de busca, evitando uma avaliação completa desnecessária (ex.: o custo da melhor solução corrente define um limite de exploração, se o custo de uma solução parcial extrapola este limite, logo a exploração da mesma pode ser descartada). Os seguintes algoritmos podem ser citados: força bruta, busca em profundidade e programação dinâmica.

O método força bruta é o mais simples entre todos os citados, ele consiste em avaliar todas as permutações possíveis entre as junções das relações presentes na consulta. O SGBDR *H2* na versão $1.3.170$, utiliza um método como este para resolver problemas envolvendo de 1 a 7 relações.

Algoritmos do tipo busca em profundidade realizam uma enumeração sistemática das soluções partindo do nó raiz. Quando uma solução é encontrada ou é detectada a impossibilidade de aprofundar a pesquisa, o método retrocede e continua a busca do nó anterior. Os SGBDRs *MySQL* $5.1.72$ e *Derby* $10.9.1.0$ aplicam métodos

baseados em busca em profundidade para a otimização de consultas.

Entre os mais conhecidos na literatura, está o algoritmo de programação dinâmica. Ele utiliza uma estratégia *bottom-up* e constrói o plano de execução em direção ao nó raiz. A ideia principal é utilizar uma tabela de custos das junções, onde apenas os custos menores são mantidos. De forma resumida, ele inicia avaliando o custo de pesquisa individual de cada relação. A seguir, para cada par de relações, avalia-se o custo de junção entre elas, sendo mantida a forma mais barata de junção. O procedimento prossegue de forma incremental, testando a junção de novas relações com as previamente avaliadas. A tabela de custos sobre as junções anteriores é utilizada para se decidir qual a nova melhor forma de junção, e por consequência, esta tabela é atualizada a cada nova inclusão. Maiores detalhes estão disponíveis em 2008c GARCIA-MOLINA; ULLMAN; WIDOM;. Ainda quanto ao algoritmo, existe uma variação (SELINGER et al., 1979), a qual mantém, além do custo das junções, informações sobre a ordem resultante das operações, o que pode diminuir posteriormente o custo de junção.

2.3.2 Algoritmos Não-exaustivos

Os algoritmos não-exaustivos caracterizam-se por não darem garantias acerca do melhor plano de execução. Contudo, na prática, têm a capacidade de retornar boas soluções. Como citado anteriormente, o problema de ordenação de junções é complexo. Diante disto, o uso de algoritmos exaustivos para enumeração dos planos pode ser proibitiva, visto que o tempo gasto e a quantidade de memória principal utilizados para a definição do plano pode ser inviável. O uso de técnicas não-exaustivas é necessário nestas situações. Todavia, a avaliação eficiente do espaço de busca permanece extremamente importante, uma vez que um plano retornado pode nunca terminar sua execução em tempo viável devido ao seu alto custo. Dito isso, é necessária uma combinação de esforços entre o tempo gasto para a enumeração dos planos e custo final deles. O algoritmo ideal deve retornar um bom plano em pouco tempo.

Existem vários algoritmos que podem ser classificados como não-exaustivos. Apenas alguns serão tratados nesta subseção, os quais podem ser categorizados como segue:

- **Heurísticos e Meta-heurísticos** - método de descida completa, busca tabu e têmpera simulada;
- **Evolucionários** - algoritmo genético.

Os métodos heurísticos e meta-heurísticos utilizam o conceito de vizinhança para explorar o espaço de busca. A vizinhança de uma solução representa todas as

soluções alcançáveis a partir dela. Um vizinho é gerado por meio de movimentos aplicados na solução em questão. Uma possível vizinhança é a de troca V^T, onde um vizinho é gerado pela troca de posição entre duas relações na solução. Ressalta-se que o movimento pode levar a um produto cartesiano dependendo da posição das trocas. A solução corrente do problema de junções pode ser representada por um vetor de inteiros contendo as identificações das relações na ordem em que as junções serão aplicadas. A Figura 11 exemplifica este processo.

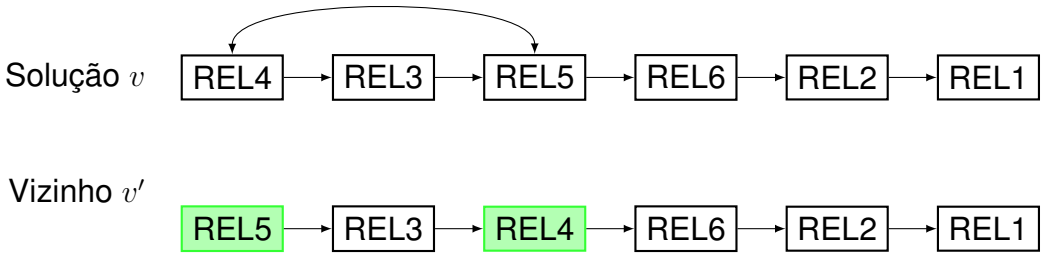


Figura 11 – Movimento de troca de duas relações - V^T

A heurística de refinamento Descida Completa (*Best Improvement*), executa uma avaliação completa da vizinhança de uma dada solução de partida; desta maneira, é necessário um esforço computacional maior para a execução do método. Assim, após a avaliação completa da vizinhança e verificação de pelo menos um vizinho com custo menor que a solução corrente, o melhor dentre os vizinhos é selecionado e aceito como a solução corrente. A avaliação da vizinhança recomeça e um novo vizinho é possivelmente selecionado. O método termina a execução quando a avaliação da vizinhança não retornar nenhum vizinho melhor, indicando que a solução corrente é um ótimo local, ou seja, de acordo com a vizinhança adotada, não haverá uma solução vizinha melhor que a corrente.

De acordo com 1997 GLOVER; LAGUNA;, a meta-heurística Busca Tabu é um procedimento de busca local que explora o espaço de soluções, além da otimalidade local, utilizando uma estrutura de memória. Ela consiste em avaliar a vizinhança de uma dada solução inicial e mover para o seu melhor vizinho. É importante notar que a escolha do vizinho pode resultar em situações de piora. Após o movimento, o vizinho ou atributos que melhor representem o vizinho são armazenados em uma estrutura de memória. Esta estrutura é utilizada para impedir por um tempo determinado o retorno a soluções previamente visitadas. A restrição é adotada para que a busca não fique presa em ótimos locais e consiga explorar o espaço de soluções mais efetivamente.

A meta-heurística Recozimento Simulado (*Simulated Anneling*) (KIRKPATRICK; GELATT; VECCHI, 1983) baseia-se na simulação do processo de recozimento de metais. Neste processo, o resfriamento rápido conduz a produtos meta-estáveis (maior energia interna), ao passo que o resfriamento mais lento conduz a produtos mais está-

veis, ditos estruturalmente mais fortes, de menor energia. No problema de ordenação de junções, pode-se associar os estados possíveis do metal a soluções do espaço de busca, onde a energia de cada estado pode ser representada pelo custo estimado do plano. O procedimento começa sua a execução com uma temperatura de recozimento inicial T, novos estados são gerados por meio mudanças aleatórias em uma dada solução, sendo o novo estado aceito se a sua energia for menor que a do estado corrente (problema de minimização). Caso a energia seja maior que a do estado corrente, a solução somente é aceita mediante uma probabilidade relacionada a T, onde quanto maior a temperatura, maior a probabilidade de aceitação do novo estado. A cada iteração, T é decrementada, o método termina sua execução quando T estiver próximo de 0, pois nenhuma solução pior será aceita.

Algoritmos evolucionários diferem-se dos demais pelo fato de trabalharem com uma população de soluções. Um algoritmo evolucionário clássico é o Algoritmo Genético (GOLDBERG, 1989). Este algoritmo é inspirado em ideias do modelo evolucionário proposto por *Darwin* e conceitos de genética. De forma sucinta, o algoritmo gerencia uma população de indivíduos ou soluções (gerados inicialmente de acordo com algum critério: aleatório ou determinístico), dos quais alguns são selecionados para realização de operações de cruzamento (combinação de informações dos indivíduos envolvidos) ou para sofrerem mutações (modificações em suas soluções). Após isto, uma nova população de descendentes é gerada, dentre os quais alguns deles serão selecionados para substituir parte da população corrente. A definição de métodos de seleção da população, operadores de cruzamento e mutação definem a forma de avaliação do espaço de soluções.

2.4 Resumo

Este capítulo teve como objetivo uma revisão sobre o processamento de consultas. Foram apresentados conceitos sobre alguns operadores de álgebra relacional e sua relação com a fase de planejamento de consultas. Também foi discutida a importância do papel do otimizador de consultas no que concerne à geração de planos de execução para consultas em banco de dados. Apresentou-se neste capítulo implementações comumente encontradas na literatura para os operadores de álgebra relacional em foco neste trabalho. Além disso, foram discutidas formas de se estimar o custo de tais operadores. O problema de ordenação de junções, tema deste trabalho, também foi comentado com mais detalhes. Por fim, algumas metodologias para enumeração de planos para as consultas foram apresentadas.

3 Planejamento e Execução de Consultas no SGBDR H2

O SGBDR utilizado para desenvolvimento do algoritmo proposto neste trabalho é o *H2 Database*[1]. Este sistema é totalmente desenvolvido em *Java* e tem as seguintes qualidades, entre outras:

- Eficiência;
- Utilização de pouca memória RAM para sua execução, ou pequeno *footprint*;
- Código-fonte aberto;
- Boa documentação;
- Estrutura de código bem organizada, ou seja, bem modularizado e classes com papéis bem definidos. Este item é essencial para iniciantes no projeto realizarem contribuições/estudos (novas implementações para alguma funcionalidade existente e avaliações práticas de metodologias);
- Seguro para uso de *threads*, isto é, possibilidade de criação e execução de processos que possam utilizar seguramente alguns recursos compartilhados necessários e sem prejuízo ao sistema[2]. Este é um pré-requisito essencial para o desenvolvimento do algoritmo proposto;
- Suporte a uma grande quantidade de funcionalidades geralmente encontradas em SGBDRs mais conhecidos.

Este capítulo é dividido como segue. Na Seção 3.1 são abordados aspectos inerentes ao planejamento e a execução de consultas no SGBDR em questão. Os otimizadores oficiais do H2 são descritos na Seção 3.2. Informações acerca do modelo de custo utilizado pelos otimizadores são apresentadas na Seção 3.3. Por fim, o capítulo é resumido na Seção 3.4.

[1] H2 Database: www.h2database.com

[2] O SGBDR PostgreSQL é um modelo de sistema multi-processo (utiliza vários processos para atender as requisições em detrimento do uso de *threads*), e sua arquitetura impede o uso de threads, pois ele não é seguro (*thread-safe*). Um exemplo de tal limitação é a chamada mais básica de alocação de memória, a qual não é *thread-safe*. Além disso, não possui um recurso que possa suprir a necessidade de se distribuir uma dada tarefa para vários processos (característica necessária ao otimizador proposto).

3.1 Planejamento e Execução das Consultas

Esta Seção tem como objetivo principal apresentar uma visão geral dos principais componentes envolvidos no planejamento e execução de consultas do H2. A Figura 12 destaca as principais classes envolvidas no processo. O servidor H2 atende a requisições do tipo TCP por meio da classe *TcpServer*, que é responsável por criar uma linha de execução (*TcpServerThread*) separada para cada nova requisição. A classe *TcpServerThread* é responsável por todo o processamento relacionado a solicitação e comunicação com o solicitante. Vale ressaltar, que como em qualquer sistema com um mínimo de segurança, há uma fase de autenticação. No H2, após o processo de autenticação, a sessão do usuário com o SGBDR é estabelecida por meio da classe *Session*, sendo esta classe utilizada por vários outros componentes. Ela reúne várias informações a respeito do usuário logado. A partir deste ponto, a solicitação relacionada já pode ser avaliada, planejada, executada e o resultado retornado. Portanto, a discussão desta Seção será distribuída entre os componentes de planejamento e execução do H2.

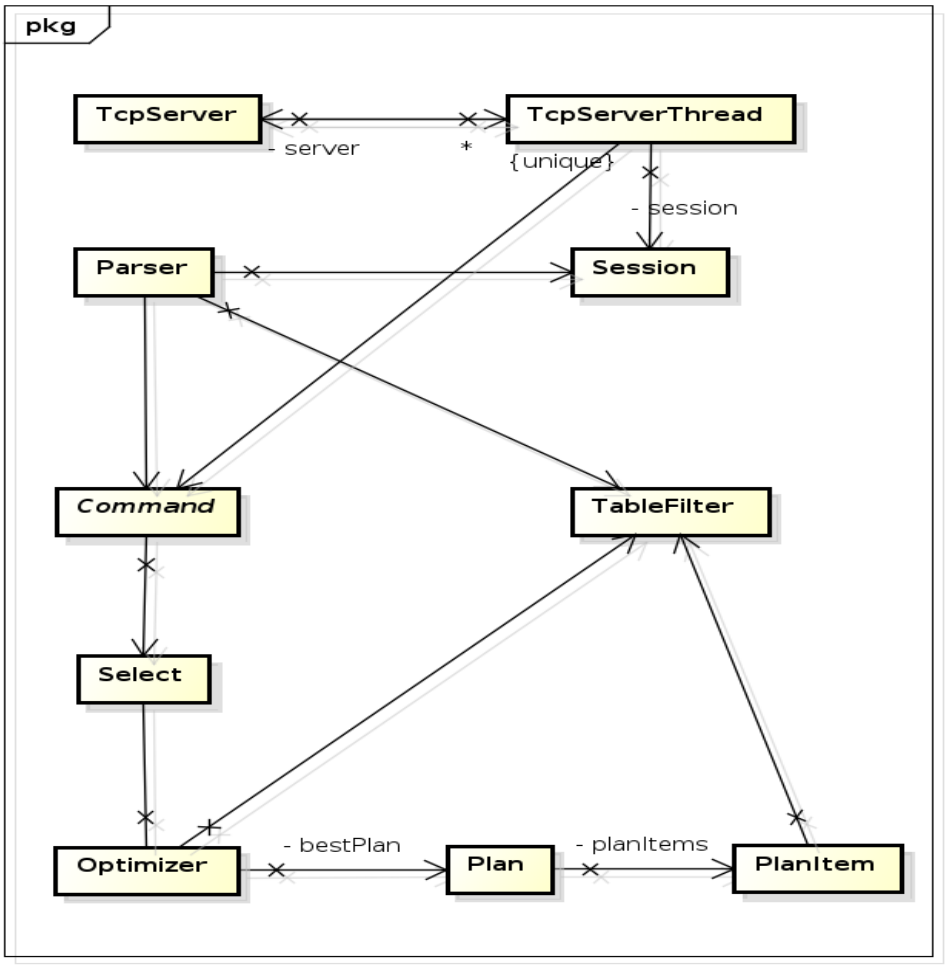


Figura 12 – Visão Geral - Planejamento e Execução.

A fase de planejamento inicia-se com o comando SQL sendo analisado pelo *Parser*. Pode-se dizer que esta avaliação executa a primeira tarefa apresentada na sequência de passos para a execução de uma consulta (Figura 3), que diz respeito aos passos: *Scanning*, *parsing* e validação. A seguir, o *Parser* produz um comando de acordo com a operação SQL, isto é, caso se trate de uma operação de seleção, o *Parser* irá instanciar uma classe do tipo *Select*. A classe *Select* é uma representação intermediária gerada pelo *Parser* que contém todas as informações necessárias para otimização e execução da consulta, sendo algumas dessas informações distribuídas a seguir: condições de junção, condições de seleção, tabelas envolvidas na consulta, informações sobre ordenação, informação sobre agrupamento de dados e plano de consulta físico. A Figura 13 detalha os atributos da classe *Select*.

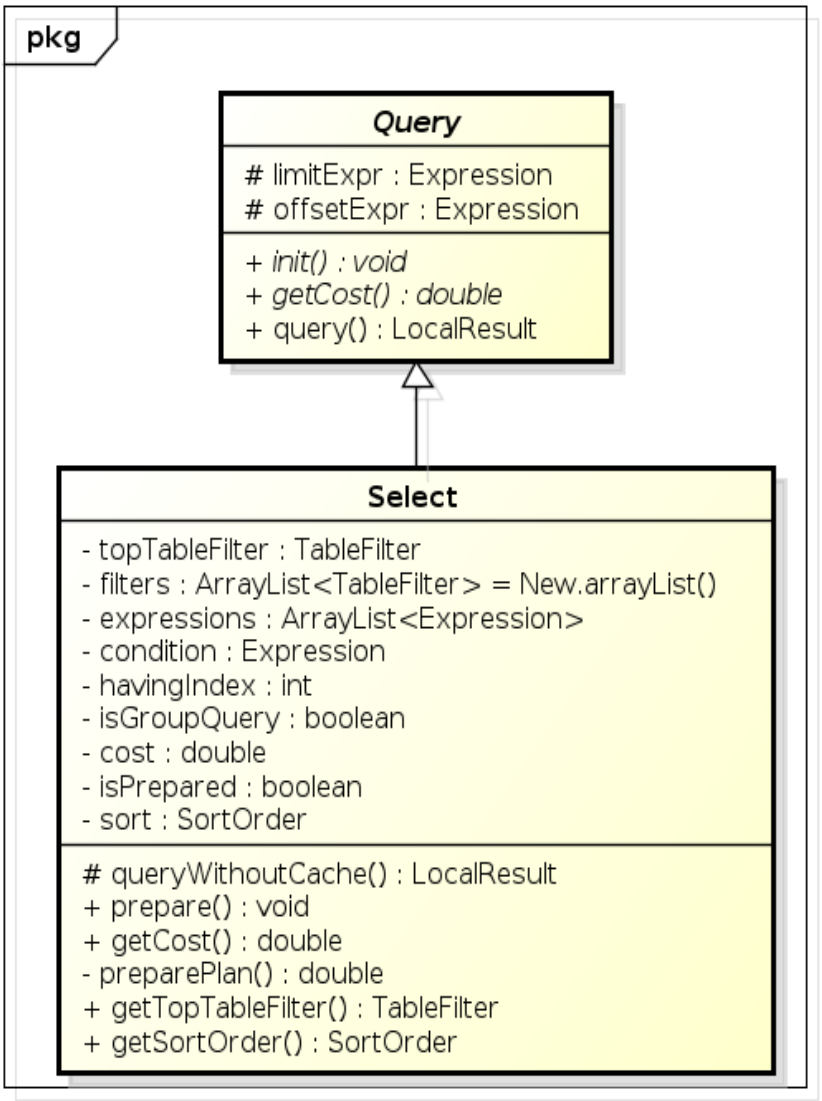


Figura 13 – Classe - Comando *Select*.

Toda a lógica relacionada às fases de planejamento e execução são acessadas

por meio da classe *Select*. Ainda em relação à classe, destaca-se a existência de três atributos: *filters*, que define as tabelas presentes na consulta, *condition*, que é a condição de junção entre as relações e *topTableFilter*, que determina a primeira relação da solução, isto é, o ponto de partida do método de junção no comando de seleção. Cada tabela presente na consulta é representada pela classe *TableFilter*, cuja a função é reunir todas as informações sobre uma relação específica validada pelo *Parser*. Durante a fase de *parsing*, cada relação lida e validada na consulta será associada a uma instância de *TableFilter*. A Figura 14 exemplifica este processo.

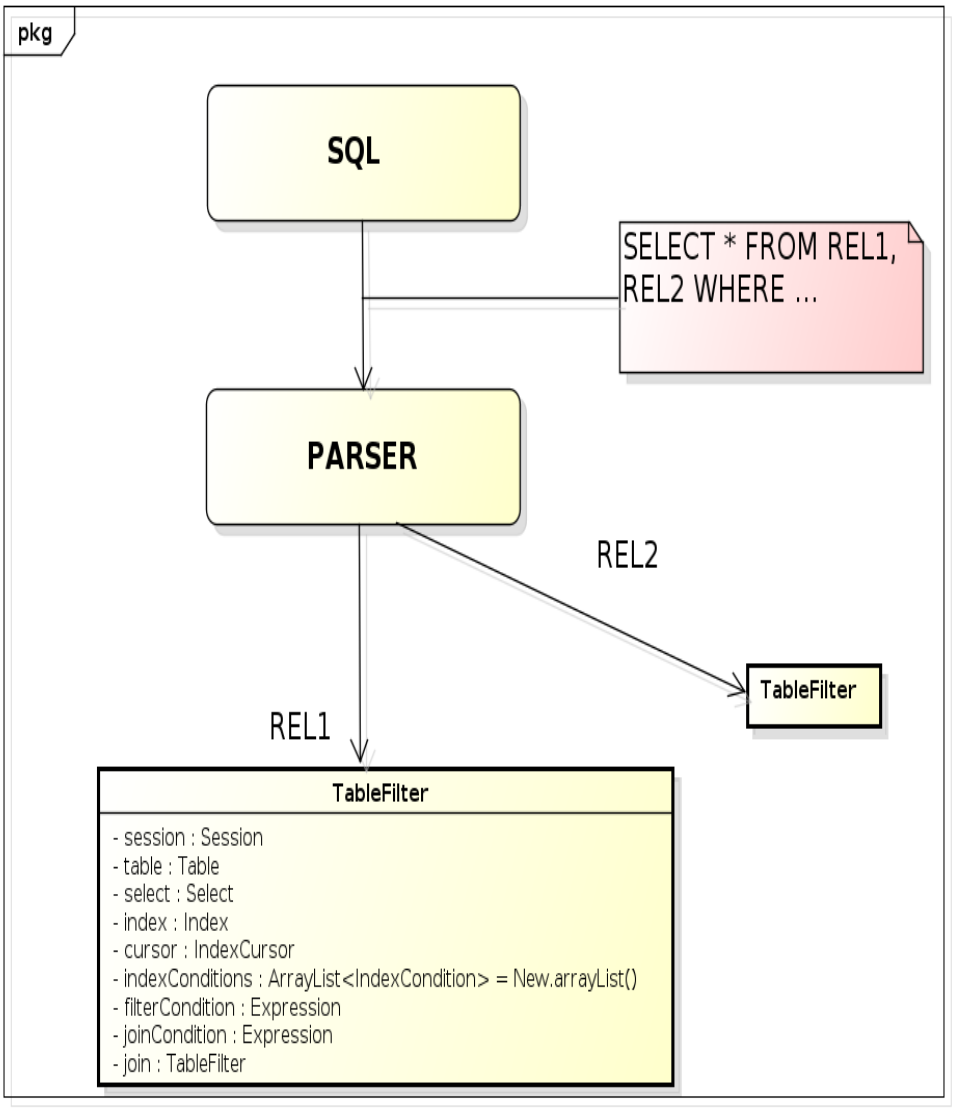


Figura 14 – Leitura das Relações da Consulta.

Em relação à classe *TableFilter*, podem ser destacados os seguintes atributos:

- ***table*** - Representa a tabela relacionada. Todas as tabelas do banco de dados são associadas internamente a instâncias da classe *Table*;

- ***index*** - Este atributo define o método de acesso que será utilizado na leitura da tabela relacionada. A classe base para os métodos de acesso no H2 é dada por *Index*;
- ***cursor*** - Toda a leitura de tuplas é abstraída pela Classe base *Cursor*, isto é, dado um método de acesso e uma tabela, o cursor irá varrer o arquivo de dados e retornará as tuplas iterativamente;
- ***filterCondition*** - Representa uma condição de seleção para leitura das tuplas;
- ***joinCondition*** - Representa a condição de junção entre tabela atual e o seu par, sendo tal relação definida pelo atributo ***join***;

O planejamento em si se concentra nas classes *Select* e *Optimizer*. Contudo, esta seção abordará aspectos do planejamento sem entrar em detalhes dos métodos de ordenação de junção disponíveis. Vale observar que tal assunto será discutido na Seção 3.2. Assim, com vistas a permitir um melhor entendimento acerca dos passos percorridos até o planejamento, é apresentado na Figura 15, um diagrama de sequência que ilustra a cadeia chamadas realizada na fase de planejamento.

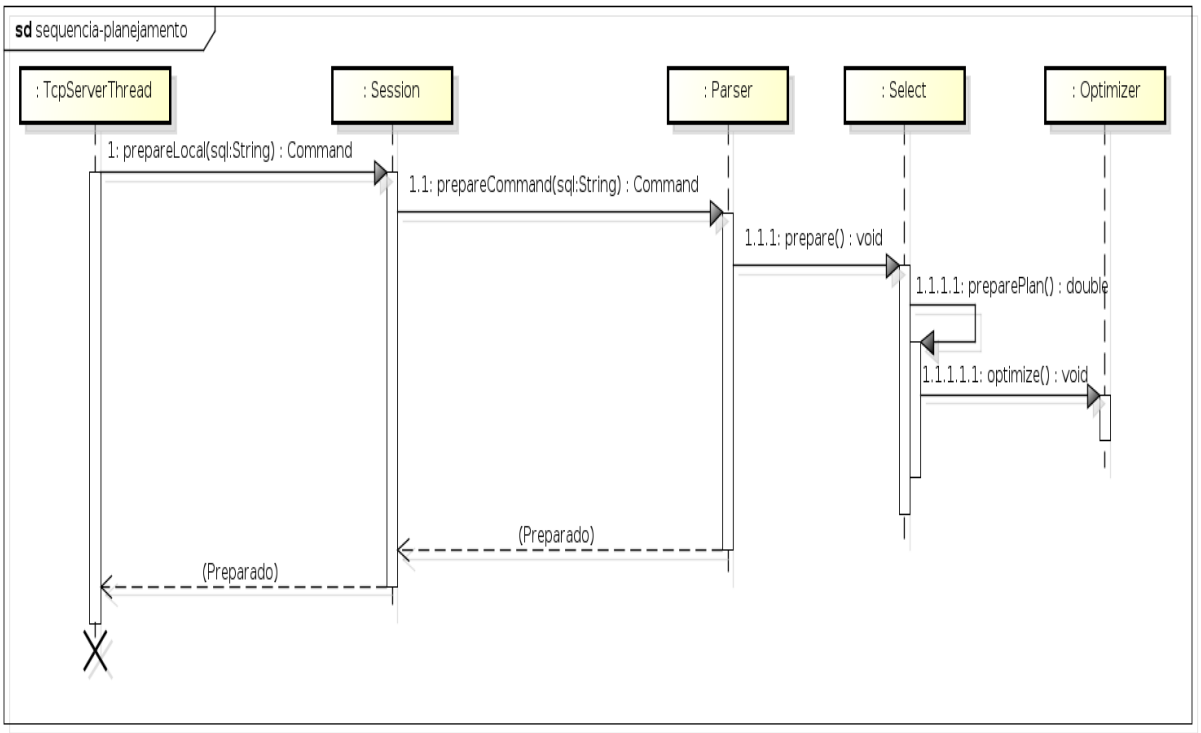


Figura 15 – Sequência de Chamadas no Planejamento de Consultas.

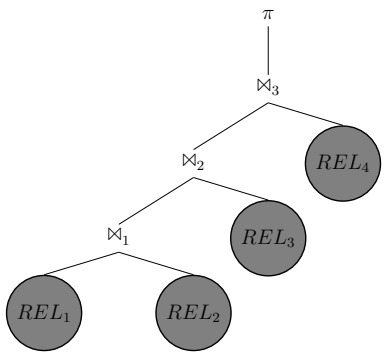
De acordo com a sequência apresentada, é possível notar que a classe *Select* inicia a preparação do plano por meio do método *prepare()*. Neste momento, são

realizadas, se necessário, algumas otimizações/transformações na consulta com intuito de otimizá-la. Uma otimização que merece destaque, é a de geração de predicados de seleção ou junção implícitos na consulta, tal otimização se apoia na propriedade de transitividade dos predicados (*transitive clousure*), a qual define que: $A = B \ e \ B = C \Rightarrow A = C$. O planejamento segue, e portanto, executa-se *prepare-Plan()*, que por sua vez realiza a ordenação de junções por meio do método *optimize()* em *Optimizer*. Após a otimização efetuada por *Optimizer*, o método *preparePlan()* recupera a primeira relação da melhor solução gerada e carrega com esta informação o atributo *topTableFilter* em *Select*. Um comando *Select* é dito planejado ou preparado, quando o atributo *topTableFilter* estiver carregado com a primeira tabela da solução do problema.

O SGBDR H2 adota como padrão de representação para as soluções, o modelo *left-deep tree* apresentado na Subseção 2.2. Deste modo, as operações de junção são compostas inicialmente por duas relações básicas, e as operações subsequentes são formadas por uma operação junção mais interna e uma relação básica. Com base nisso, casos em que houver alguma junção entre relações, a classe *Select* irá compor a primeira junção J_1, com *topTableFilter* sendo o nó externo REL_1 e a relação definida no atributo *join* de *topTableFilter* como o nó interno REL_2. Quando necessário, novas operações de junções são formadas pela operação de junção anterior e a próxima relação da solução, ou seja, para construir a segunda junção J_2, utiliza-se J_1 como nó externo e a relação definida pelo atributo *join* de REL_2 como nó interno REL_3. Para um melhor entendimento, considere a solução exemplo $[REL_1 , REL_2 , REL_3 , REL_4]$, em que REL_1 é definido como o *topTableFilter* da classe *Select*. Por fim, a Figura 16 apresenta a árvore resultante.

TableFilter	**Atributo *join***
REL_1	REL_2
REL_2	REL_3
REL_3	REL_4
REL_4	Nulo

(a) *Relações - Solução*



(b) *Árvore de Junções Resultante*

Figura 16 – Exemplo - Composição da Árvore de Junções.

O exemplo anterior mostra que a solução do problema pode ser reconstruída pelo executor partindo-se de uma instância *TableFilter* raiz e gerando as operações de

junção subsequentes com base na cadeia de relacionamentos entre as demais instâncias de *TableFilter*. O diagrama de sequência da Figura 17 exibe o comportamento das classes após o planejamento, ou seja, os passos seguidos para a execução do *Select* preparado.

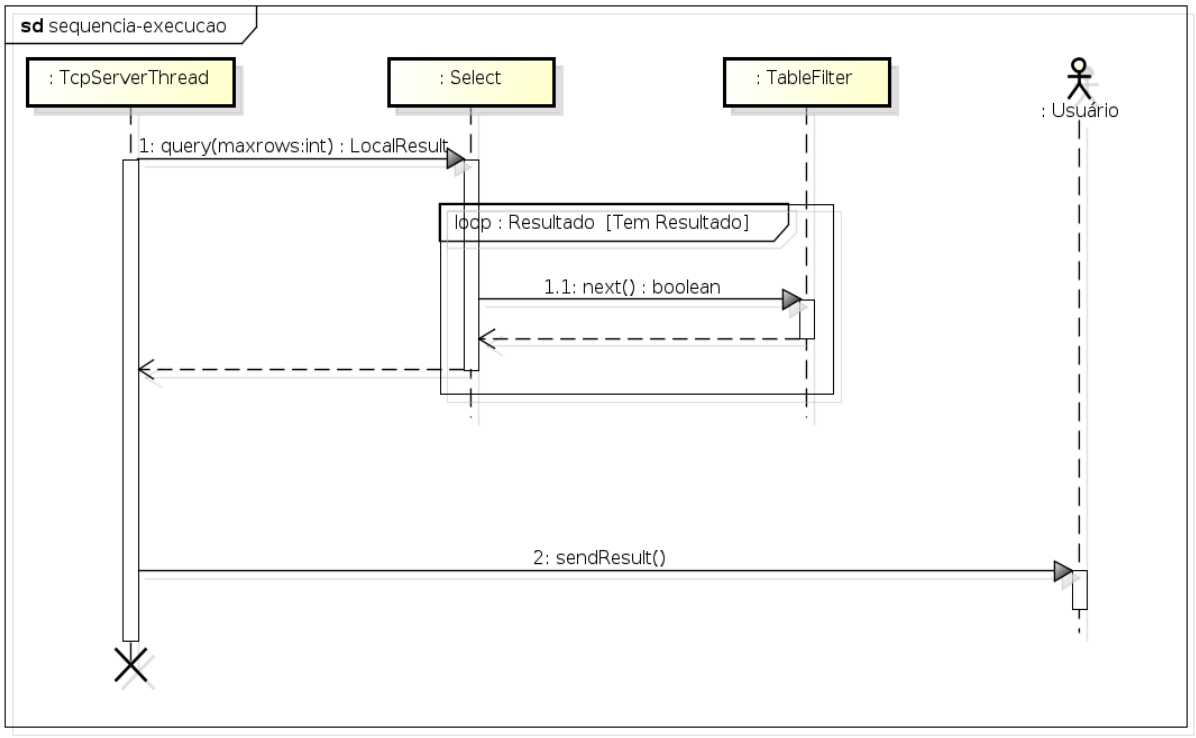


Figura 17 – Sequência de Chamadas na Execução de Consultas.

Pelo diagrama apresentado pode ser verificado que a lógica de execução do plano definido está presente no método *query()*. O SGBDR H2 utiliza como abordagem para a junção das relações da consulta, apenas o método *Nested-Loop Join*, tal como apresentado na Subseção 2.1.1. Cabe ressaltar que o executor de consultas é a própria classe *Select*, ela utiliza um laço de repetição para construir o resultado. Além disso, observa-se no diagrama da Figura 17, que no laço citado, é realizada a chamada do método *next()* em *TableFilter*, que por sua vez irá chamar recursivamente o mesmo método *next()* das instâncias aninhadas. Toda lógica de execução relacionada ao método *Nested-Loop Join* está codificada em *next()*. A metodologia de laços aninhados do método de junção em questão é realizada com base nos seguintes passos:

1. **Próxima Tupla** - Aqui é realizada a chamada que inicia ou continua a execução do método de junção;
2. **Carregar Tupla** - Carrega uma tupla com base no último ponto de parada e salva o estado atual, isto é, no início da leitura, carrega-se a primeira tupla da relação, caso contrário, continua-se com a próxima tupla que segue a última

carregada. É importante observar que durante o carregamento, também valida-se, se necessário, a condição de seleção relacionada (atributo *filterCondition* de *TableFilter*);

3. **Validar Junção** - Verifica se as tuplas carregadas nas relações em questão satisfazem à condição de junção das mesmas (condição de junção nula implica em verdadeiro na validação). Vale ressaltar que a condição de junção é definida pelo atributo *joinCondition* de *TableFilter*, e que a instância *topTableFilter* do comando *Select* não possui uma condição de junção, pois tal validação é delegada ao seu par na junção;

4. **Limpar Estado** - Força a leitura das tuplas a reiniciar da primeira tupla novamente;

O algoritmo 1 apresenta um esquema simplificado do funcionamento da metodologia de junção utilizada pelo H2. O método inicia com uma tentativa de continuação de algum processo anterior (linha 4). Caso seja a primeira execução ou a continuação do processo anterior falhe, ele prosseguirá para o laço da linha 11. Assim, o procedimento carrega a tupla correspondente da relação em avaliação na linha 12. A condição de junção relacionada é testada na linha 17. Vale ressaltar que condições nulas são automaticamente validadas. Por conseguinte, nos casos em que houver alguma relação aninhada à atual, isto é, alguma relação que forme um par de junção com a relação avaliada no momento, o estado de leitura da relação aninhada é zerado (linha 21) e o procedimento atual (*next*) será chamado recursivamente para a relação aninhada, o que encadeará o carregamento de suas tuplas, validação de sua condição de junção e posterior avaliação das demais relações aninhadas. Por fim, uma junção entre todas as relações será totalmente validada se todos os métodos *next* retornarem verdadeiro, o que significa que existe uma tupla carregada para cada relação envolvida que satisfaz as condições de seleção e junção relacionadas.

Algoritmo 1: next

```
   Data: relation
   Result: True, if some tuple was loaded, otherwise, returns False
 1 begin
 2     //Inicialmente é necessário testar se é possível continuar uma execução anterior;
 3     joinRelation ← relation→join;
 4     if joinRelation <> ∅ then
 5         state ← next(joinRelation);
 6         if state = TRUE then
 7             return TRUE;
 8         end if
 9     end if
10     //Neste ponto, itera-se indefinidamente sobre o conjunto de tuplas da relação
       passada como parâmetro e nas demais relações aninhadas;
11     while TRUE do
12         state ← loadTuple(relation);
13         if state = NOT-FOUND then
14             return FALSE;
15         end if
16         if state = FOUND then
17             joinState ← validateJoin(relation→joinCondition);
18             if joinState = TRUE then
19                 joinRelation ← relation→join;
20                 if joinRelation <> ∅ then
21                     clearState(joinRelation);
22                     state ← next(joinRelation);
23                     if state = FALSE then
24                         retorne para a linha 12;
25                     end if
26                 end if
27                 return TRUE;
28             end if
29         end if
30     end while
31 end
```

Durante o carregamento das tuplas na execução das consultas, o SGBDR H2 faz uso de um intermediador conhecido como *Cursor*. Esse componente permite à *TableFilter* iterar de forma simplificada sobre as tuplas correspondentes e provê uma interface de comunicação simples entre o executor de consultas e os componentes responsáveis pelo carregamentos das páginas. Como dito anteriormente, cada instância *TableFilter* conhece a sua tabela relacionada (atributo *table*) e o seu método de acesso (atributo *index*), desta forma, um *Cursor* pode ser formado com base em tais informações. A Figura 18 apresenta o esquema de interação entre *TableFilter* e *Cursor*. Verifica-se que *TableFilter* acessa as tuplas por meio da classe *IndexCursor*, que por sua vez utiliza a interface *Index*, a qual é base para qualquer método de acesso à informações no H2. Há dois métodos explicitados no relacionamento apresentado,

são eles: acesso sequencial ao arquivo de dados (PageDataIndex) e acesso por meio de um índice do tipo Árvore-B (PageBtreeIndex). Todos os métodos de acesso utilizam um componente chamado *PageStore*, que dentre outras tarefas, é responsável por abstrair certos detalhes de organização física das páginas no disco, permitindo que as mesmas sejam recuperadas de forma simplificada. Mais ainda, o componente *PageStore* também é responsável por alimentar e acessar a área destinada ao cache de páginas. O cache de páginas permite que a leitura das tuplas em alguns momentos seja realizada em memória primária, eliminando todo o custo relacionado ao acesso a disco[3].

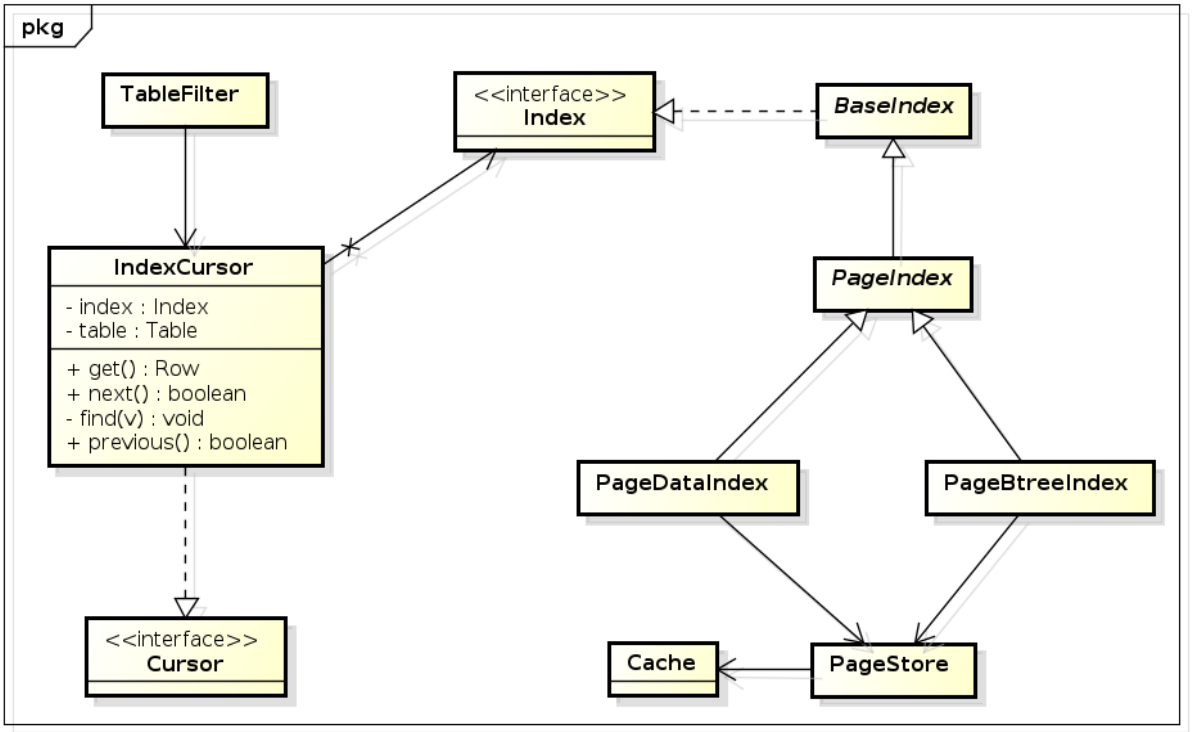


Figura 18 – Interação entre *TableFilter* e *Cursor*.

Além da simplificação do processo de iteração sobre as tuplas, uma outra característica importante que merece destaque, é a capacidade da classe *IndexCursor* em identificar propriedades nos métodos de acesso que o permite interromper a leitura das tuplas durante o processo de junção, o que evita a leitura e o processamento desnecessário de tuplas (o método ***Merge-Sort Join*** da Subseção 2.1.1 também utiliza tal heurística). Essa funcionalidade pode ser exemplificada com a junção entre duas relações $REL_1 \bowtie_{ATR_1=ATR_2} REL_2$, onde o método de acesso em REL_2 é dado por um índice ordenado. Ressalta-se que índices ordenados permitem concluir naturalmente se um conjunto de valores para uma dada coluna serão menores/maiores que um

[3] O espaço total destinado ao cache de páginas é definido pelo atributo de configuração *CACHE_SIZE*.

dado valor em algum momento da leitura. Para um melhor entendimento, considere as tabelas da Figura 19, que distribui os valores das tuplas das colunas ATR_1 e ATR_2, de REL_1 e REL_2, respectivamente.

Valor - ATR_1
1
$\longrightarrow \mathbf{3}$
4
5

(a) REL_1

Valor - ATR_2
$\longrightarrow \mathbf{4}$
5
5
6

(b) REL_2

Figura 19 – Exemplo - Distribuição valores de REL_1 e REL_2.

A metodologia proposta pela implementação do tipo ***Nested-Loop Join*** (Algoritmo 1), define que para cada tupla de REL_1, deverão ser lidas e validadas todas as tuplas de REL_2, sendo selecionadas aquelas que satisfazerem a condição de junção ($\bowtie_{ATR_1=ATR_2}$). Portanto, em uma situação exemplo onde a tupla carregada pelo *IndexCursor* de REL_1 possua o valor 3, o *IndexCursor* de REL_2 irá interromper a leitura logo na primeira iteração, pois o primeiro valor lido é 4 e o índice ordenado permite concluir que os valores subsequentes serão maiores ou iguais a 4, o que implica na violação do predicado de junção para todos os valores de ATR_2. Tal circunstância terá como consequência o não-carregamento das tuplas pelo método *CarregarTupla* para REL_2.

3.2 Otimizadores H2

A discussão desta seção trata especificamente das implementações oferecidas pelo SGBDR H2 para resolver o problema de ordenação de junções. A classe *Optimizer* é responsável por realizar tal otimização e o seu relacionamento com o comando *Select* é exibido na Figura 20. Observa-se que o método *calculateBestPlan()* é quem define qual metodologia aplicar: exaustiva ou não-exaustiva. Assim, a classe *Select* sempre invocará este método para realizar a ordenação de junções.

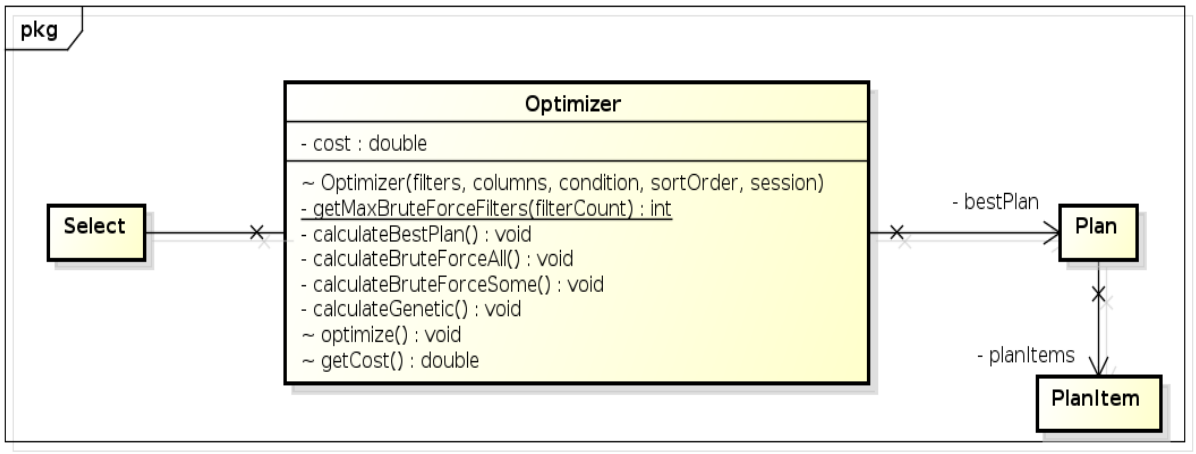


Figura 20 – Otimização do Problema de Ordenação de Junções.

O método *calculateBruteForceAll()* (Algoritmo 2) implementa lógica exata ou exaustiva do H2. Ele garante o ótimo global e é aplicado em consultas envolvendo até 7 relações. Sua implementação é simples, são testadas todas as permutações possíveis entre as relações da consulta e retornado o plano com a permutação de menor custo. Um ponto negativo que pode ser destacado é a possível avaliação de soluções com planos cartesianos.

Algoritmo 2: calculateBruteForceAll

```
   Data: relations
   Result: Best Plan
 1 begin
 2     allPermutations ← generatePermutations(relations);
 3     bestCost ← ∞;
 4     bestPlan ← ∅;
 5     for solution in allPermutations do
 6         plan ← testSolution(solution);
 7         if plan→cost < bestCost then
 8             bestCost ← plan→cost;
 9             bestPlan ← plan;
10         end if
11     end for
12     return bestPlan;
13 end
```

Para consultas envolvendo 8 ou mais relações, uma versão não-exaustiva de otimização é utilizada. Os métodos *calculateBruteForceSome()* (Algoritmo 3) e *calculateGenetic()* (Algoritmo 4) são responsáveis por tal tarefa. A otimização mescla entre metodologia exata, gulosa e avaliação aleatória. Inicialmente é determinado o número máximo de relações que serão otimizadas de forma exata. Este limite é calculado com base numa heurística definida no método *getMaxBruteForceFilters()*. Definido o nú-

mero de relações, o método *calculateBruteForceSome()* avalia todas as permutações possíveis dentro do limite máximo de relações estabelecido. A seguir, ele seleciona a solução parcial cuja permutação seja a menos custosa, sendo as demais relações inseridas na solução de forma gulosa. Esta inserção utiliza o custo de junção entre todas as relações ainda não inseridas, sendo selecionada em cada rodada a relação com o menor custo de junção. Após a solução completa ser formada, a mesma sofre um número de pertubações com vistas a se explorar o espaço de soluções aleatoriamente (método *calculateGenetic()*). O número máximo pertubações é definido por padrão em 500 e sua a escolha não está documentada no código, o que impede uma explicação acerca de tal escolha. A avaliação do espaço de soluções aplica movimentos de troca entre relações. O algoritmo varia entre a troca de duas relações selecionadas aleatoriamente e a realização de N trocas aleatórias, sendo N o número de relações. Caso alguma troca implique em melhora no custo, o otimizador move para esta solução.

Algoritmo 3: calculateBruteForceSome

```
   Data: relations
   Result: Best Plan
 1 begin
 2     max ← getMaxBruteForceFilters(relations);
 3     allPermutations ← generatePermutations(relations, max);
 4     partialCost ← ∞;
 5     bestPlan ← ∅;
 6     for solution in allPermutations do
 7         plan ← testSolution(solution);
 8         if plan→cost < partialCost then
 9             partialCost ← plan→cost;
10             bestPlan ← plan;
11         end if
12     end for
13     remainingRelations ← remaining(relations,bestPlan);
14     addGreedyWay(remainingRelations, bestPlan);
15     return bestPlan;
16 end
```

Algoritmo 4: calculateGenetic

```
   Data: solution
   Result: Best Plan
 1 begin
 2     bestPlan ← testSolution(solution);
 3     for i=0 to 500 do
 4         if random() < 0.5 then
 5             solutionAux ← shuffleTwo(solution);
 6             plan ← testSolution(solutionAux);
 7             if plan→cost < bestPlan→cost then
 8                 bestPlan ← plan;
 9                 solution ← bestPlan→solution;
10             end if
11         else
12             size ← sizeOf(solution);
13             N ← random(size);
14             solutionAux ← shuffleN(solution, N);
15             plan ← testSolution(solutionAux);
16             if plan→cost < bestPlan→cost then
17                 bestPlan ← plan;
18                 solution ← bestPlan→solution;
19             end if
20         end if
21     end for
22     return bestPlan;
23 end
```

Todo processo de otimização se apoia no componente de modelo de custo (classe *Plan* da Figura 21), isto é, os otimizadores se preocupam apenas em gerar soluções com diferentes formas de ordenação das relações, deixando a cargo do componente de custo, a tarefa de inferir o custo da permutação gerada. Este tipo abstração provê uma camada de modularização bastante interessante, uma vez que o otimizador se concentra totalmente na exploração do espaço de soluções e o componente de custo fica encarregado de ler a solução, determinar os predicados de junção relacionados, definir o melhor método acesso com base nos pares de junção e por fim, retornar o custo da solução avaliada. Mais ainda, esta separação de responsabilidades permite ao otimizador trabalhar indiferentemente com problemas no domínio dos inteiros, reais, tipos textuais, etc. O componente de custo abstrai todos os detalhes inerentes aos cálculos relacionados aos diferentes tipos de dados e às operações de comparação. De acordo com a Figura 21, a interação do componente de custo com o otimizador pode ser resumida em dois pontos na classe *Plan*: método construtor e o método *calculateCost()*. O otimizador sempre cria uma instância da classe *Plan* para calcular o custo de uma solução. Para tal, a classe *Optimizer* sempre carrega o método construtor de *Plan* com dois parâmetros: o atributo *filters*, que representa

a solução a ser avaliada (toda solução em *Optimizer* é representada por um vetor do tipo *TableFilter*) e *condition*, que reúne todos os predicados de junção presentes no comando *Select*. Finalmente, de posse da instância de *Plan*, o custo da solução passada é gerado por meio do método *calculateCost()*, que cria um plano específico (classe *PlanItem*) para cada *TableFilter* e retorna o custo total. Mais detalhes acerca do cálculo de custo são apresentados na Seção 3.3.

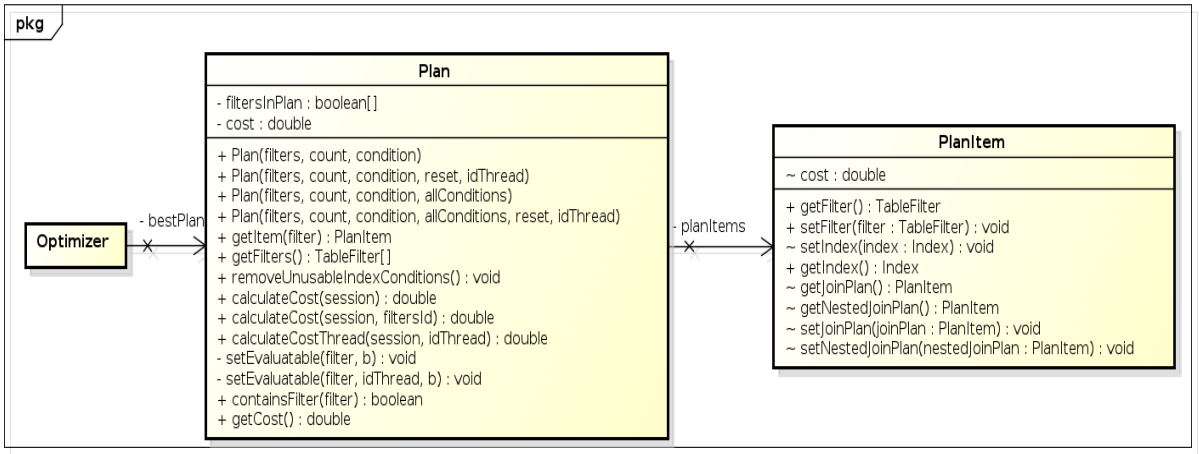


Figura 21 – Interação entre *Optimizer* e o Componente de Custo.

3.3 Modelo de Custo

Todo o processo de otimização depende diretamente do modelo de custo empregado pelo H2 para estimar o custo dos planos. Como descrito anteriormente na Seção 3.2, o otimizador de junções gera uma solução e delega ao componente de custo a tarefa de estimar o esforço necessário para se executar a consulta de acordo com a ordem das relações definida. A Figura 22 apresenta um diagrama do relacionamento entre as principais classes envolvidas no processo de estimativa de custos.

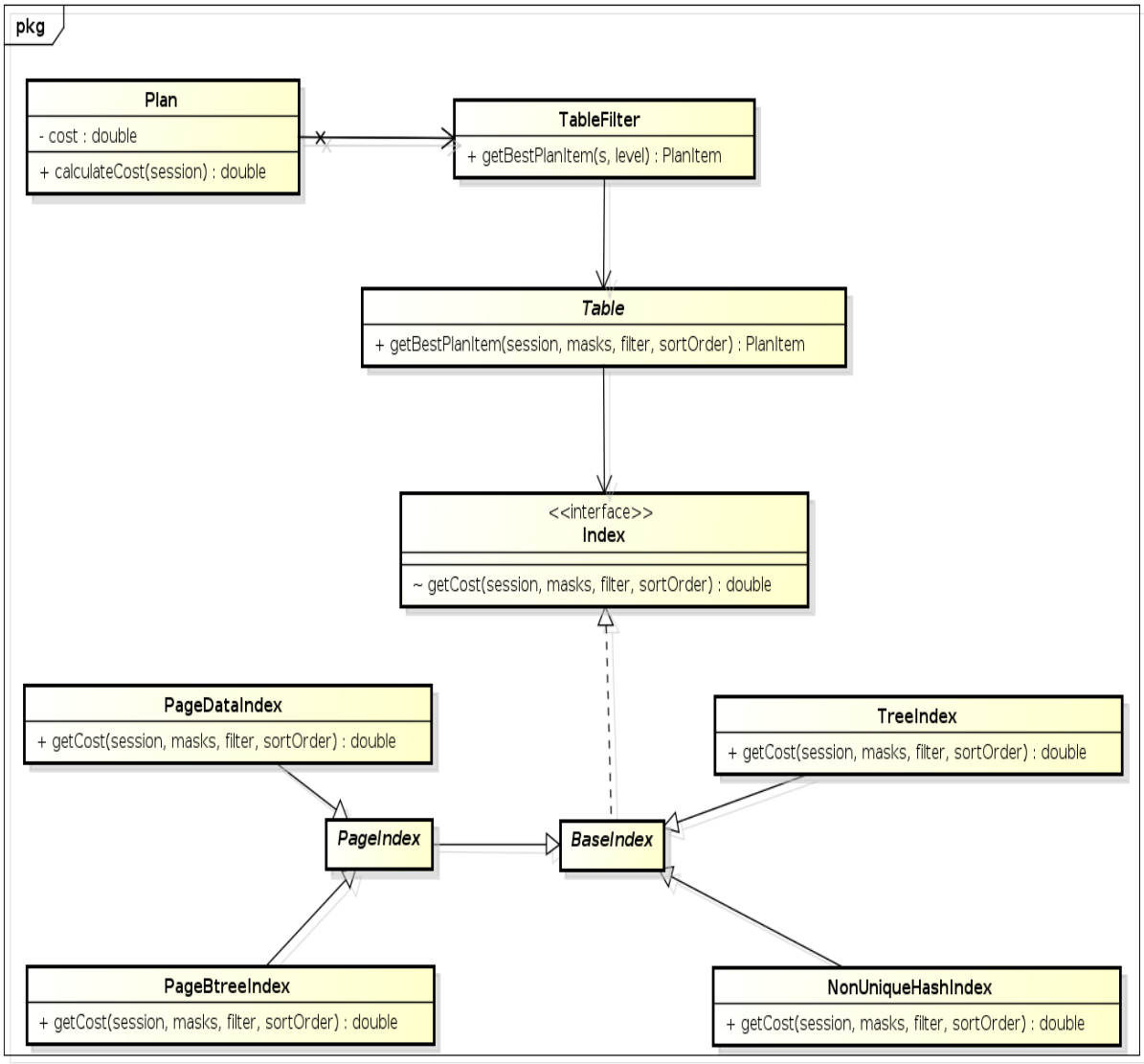


Figura 22 – Principais Classes do Modelo de Custo.

Em resumo, o custo total de uma solução é dado pela multiplicação de todos os custos individuais associados à forma de acesso escolhida para a junção de cada uma das relações envolvidas. O método *calculateCost()* da classe *Plan* é responsável por iterar sobre cada instância *TableFilter* da solução, obter o custo individual associado à junção da relação e multiplicar o custo retornado ao custo atual. O pseudo-código apresentado no bloco 5 exemplifica este processo. O custo individual de junção de cada relação é obtido na linha 6 e posteriormente incluído ao custo total na linha 7. O registro da relação na solução, isto é, a indicação de que a relação foi incluída na solução é realizada na linha 8.

Algoritmo 5: calculateCost

```
   Data: solution
   Result: Total Cost
1  begin
2      totalCost ← ∅;
3      size ← sizeOf(solution);
4      for i=1 to size do
5          tableFilter ← solution[i];
6          planItem ← tableFilter→getBestPlanItem();
7          totalCost ← totalCost + (totalCost × planItem→cost);
8          addToSolution(tableFilter,planItem);
9      end for
10     return totalCost;
11 end
```

O custo associado à junção de uma dada relação à solução atual é medido pelo método *getBestPlanItem()* de *TableFilter*. O trabalho realizado por esse método é descrito sucintamente no Algoritmo 6.

Algoritmo 6: getBestPlanItem

```
   Data: relation
   Result: PlanItem of the related TableFilter
1  begin
2      planItem ← ∅;
3      mask ← ∅;
4      allJoinPredicates ← getAllPredicatesWithTable(relation);
5      for predicate in allJoinPredicates do
6          state ← isInSolution(predicate→outerRelation);
7          if state = TRUE then
8              encode(mask,relation,predicate→innerColumn,predicate→operator);
9          end if
10     end for
11     planItem ←relation→getBestPlanItem(mask);
12     return planItem;
13 end
```

Com base no algoritmo 6, verifica-se que na linha 4 são selecionados inicialmente todos os predicados de junção entre a relação de entrada e as demais relações. Cada predicado é composto pela relação de entrada (*innerRelation*), o seu par na junção (*outerRelation*), as respectivas colunas *innerColumn* e *outerColumn* do predicado de junção e o operador de comparação (*operator*). O algoritmo prossegue iterando sobre cada um dos predicados (linha 5), selecionando nas linhas 6 e 7, aqueles que a relação *outerRelation* estiver presente na solução (uma relação é inserida na solução por meio do método addToSolution() no algoritmo 5). Tal validação atesta se existe uma forma de junção possível entre a relação de entrada e outra já presente na solução, o que possibilita a utilização de algum método de acesso associado à

tabela de entrada que possua a coluna *innerColumn* em sua estrutura. A seguir, os predicados selecionados são codificados (linha 8). O processo de codificação permite que, posteriormente, seja possível saber para cada coluna de uma relação, se existe algum predicado de junção válido e qual o operador de comparação desse predicado ($=, <, \leq, \ldots$). Os predicados codificados na variável *mask* são utilizados como entrada para o método *getBestPlanItem(mask)* da classe Table (chamada da linha 11). Com base nisso, o procedimento *getBestPlanItem(mask)* irá estimar o esforço necessário para a junção da relação de entrada. A metodologia utilizada pela classe *Table* para calcular o custo é exemplificada no Algoritmo 7.

Algoritmo 7: getBestPlanItem

```
   Data: relation, mask
   Result: PlanItem of the related TableFilter
 1 begin
 2     allIndexes ← getAllAcessMethods(relation);
 3     bestCost ← ∞ ;
 4     bestIndex ← ∅;
 5     for index in allIndexes do
 6         state ← validIndex(mask,index);
 7         if state = TRUE AND index→cost < bestCost then
 8             bestCost ← index→cost;
 9             bestIndex ← index;
10         end if
11     end for
12     planItem ← buildPlanItem(relation,bestIndex,bestCost);
13     return planItem;
14 end
```

A lógica aplicada no Algoritmo 7 basicamente varre todos os métodos de acesso disponíveis para a relação em questão (linha 5), valida aqueles que podem ser selecionados (linha 6) e elege o de menor custo (linhas 7, 8 e 9). A verificação da possibilidade de uso do método de acesso efetuada na linha 6, essencialmente avalia se as colunas existentes na estrutura do meio de acesso estão presentes nos predicados codificados (variável *mask*) e se a operação relacionada é compatível com o método de acesso, isto é, tabelas *hash* podem ser aplicadas apenas em operações de igualdade, árvores-b são adequadas tanto para igualdades, quanto para desigualdades, e assim por diante. O método de acesso representado pela variável *index* (linha 5) corresponde à interface *Index* da Figura 23. Vale ressaltar que cada forma de acesso (*PageDataIndex*, *TreeIndex*, etc) implementa indiretamente a interface *Index* e define seu próprio cálculo de custo por meio do método *getCost()*.

A metodologia utilizada pelo H2 mescla entre custos associados às operações de *CPU* e *I/O*. O cálculo associado à *CPU* é quantificado por meio do número de tuplas a serem processadas na memória principal. Já o esforço relacionado ao acesso

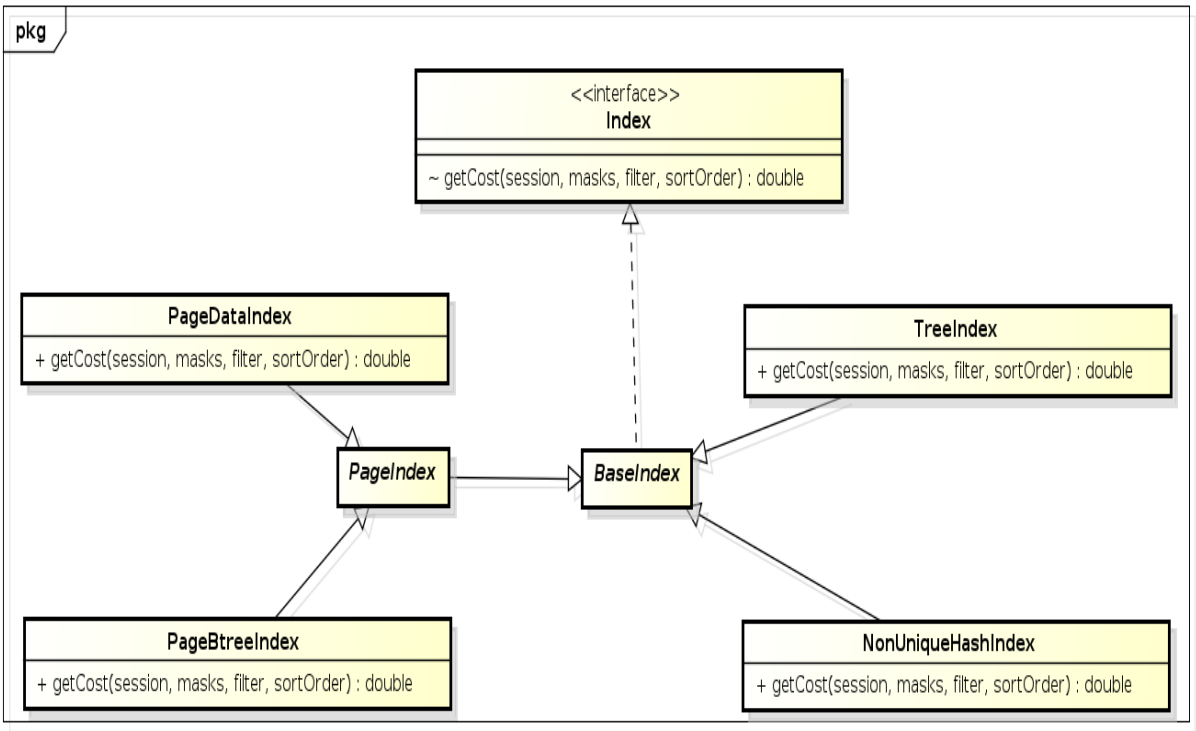


Figura 23 – Métodos de Acesso do Modelo de Custo.

a disco é definido por meio de constantes ligadas a cada tipo de acesso. A listagem seguinte descreve o custo de cada um dos métodos de acesso destacados na Figura 23.

- ***PageDataIndex (PD)*** - Tipo de acesso que varre todas as páginas de tuplas sequencialmente (método *Table-Scan* da Subseção 2.1.1) e possui o custo dado por $CUSTO = 10 \times (T(REL) + PENALIDADE)$, onde o número 10 representa a constante de *I/O* do meio de acesso, $T(REL)$ o número de tuplas processadas e *PENALIDADE* uma constante de penalização configurada por padrão com o valor 1000. Cabe ressaltar que essa forma de acesso é a mais cara, sendo utilizada na falta de outros meios de acesso ou quando a relação analisada é a primeira da solução;
- ***TreeIndex (TI)*** - Implementação de árvore-AVL que executa apenas na memória principal e tem o custo definido como $CUSTO = 1 \times TE(REL)$, em que a constante de *I/O* é 1 e o número de tuplas processadas é dado pelo número de tuplas esperadas $TE(REL)$. Mais ainda, é importante atentar que a metodologia utilizada para determinar o número de tuplas esperadas é a mesma apresentada na Subseção 2.1.2;
- ***PageBtreeIndex (PB)*** - Meio de acesso do tipo árvore-b. Essa implementação não tem sua execução limitada à memória principal e tem o custo estimado por

$CUSTO = 10 \times TE(REL)$, onde os fatores I/O e CPU são estimados pela constante 10 e pelo número de tuplas esperadas $TE(REL)$, respectivamente;

- ***NonUniqueHashIndex (HI)*** - Este método de acesso se apoia em tabelas *hash* e diferente dos outros métodos, só pode ser usado em condições envolvendo igualdades. Além disso, tem a execução delimitada à memória principal. O custo é definido como $CUSTO = 2$, onde a constante de *I/O* é expressa por 2 e observa-se, implicitamente, que apenas uma tupla será processada.

É importante destacar que o modelo de custo adotado pelo H2 não faz uso da seletividade dos predicados de junção para tentar inferir o número de tuplas resultantes de uma operação de junção. O custo estimado limita-se a cada relação inserida, no que diz respeito às tuplas processadas e à metodologia de acesso escolhida. Segundo a comunidade do H2, este modelo se comporta bem na grande maioria das situações levantadas pelos usuários e desenvolvedores nos fóruns relacionados. Além disso, os resultados que serão apresentados na seção experimental corroboram com as afirmações da comunidade. Mais ainda, o popular SGBDR *MySQL* na versão 5.1, também utiliza uma metodologia bem semelhante a do H2. Destaca-se também que a otimização das junções é indiretamente afetada pela ordem das tuplas, visto que o custo de uma solução é estimado considerando-se a solicitação de ordenação das tuplas e a ordem natural gerada pela solução, ou seja, soluções que atendam naturalmente à critérios de ordenação explícitos, tendem a ser mais baratas em algumas situações. Por fim, é apresentado a seguir um exemplo prático com o modelo de custo. Para tal, devem ser consideradas as informações apresentadas na Figura 24.

$T(REL_1)$	$TE(REL_1)$	ATR_1			
		PD	**TI**	**PB**	**HI**
10	10	Sim	Não	Sim	Não

(a) REL_1

$T(REL_2)$	$TE(REL_2)$	ATR_2			
		PD	**TI**	**PB**	**HI**
10	10	Sim	Não	Não	Não

(b) REL_2

Figura 24 – Exemplo Custo - Informações das relações REL_1 e REL_2.

De acordo com as tabelas da Figura 24, REL_1 tem 10 tuplas e os meios de acesso *PageDataIndex* e *PageBtreeIndex* disponíveis. Por outro lado, REL_2 possui

10 tuplas, mas apenas *PageDataIndex* como meio de acesso. A consulta exemplo é exibida na listagem 3.1.

Listing 3.1 – Exemplo Custo - Consulta SQL.

SELECT ATR_1, ATR_2 **FROM** REL_1, REL_2 **WHERE** $REL_1.ATR_1 = REL_2.ATR_2$

Com base na consulta SQL 3.1, duas possíveis soluções podem ser identificadas: $SOL_1 = [REL_1, REL_2]$ ou $SOL_2 = [REL_2, REL_1]$. A Tabela 5 resume o custo associado às soluções e a combinação dos métodos de acesso.

Solução	ATR_1		ATR_2		**Custo**
	PD	PB	PD	PB	
SOL_1	Sim		Sim		$C_{REL_1} = 10 \times (10 + 1000) = 10100$ $C_{REL_2} = 10 \times (10 + 1000) = 10100$ $C_{TOTAL} = 102020100$
SOL_2	Sim		Sim		$C_{REL_1} = 10 \times (10 + 1000) = 10100$ $C_{REL_2} = 10 \times (10 + 1000) = 10100$ $C_{TOTAL} = 102020100$
SOL_2		Sim	Sim		$C_{REL_1} = 10 \times 10 = 100$ $C_{REL_2} = 10 \times (10 + 1000) = 10100$ $C_{TOTAL} = 1020100$

Tabela 5 – Exemplo Custo - Resumo

Observa-se na Tabela 5 que a solução mais barata é SOL_2, com leitura sequencial na coluna ATR_2 e pesquisa com árvore-b na coluna ATR_1. Vale lembrar que o H2 escolhe a pesquisa sequencial automaticamente para a primeira relação da solução, ou seja, ele fixa PD para REL_1 na solução SOL_1 e REL_2 em SOL_2. Tal estratégia parte da ideia de que não há nenhum predicado de junção válido na primeira iteração, pois se trata da relação inicial. Assim, o único método de acesso compatível nessa situação, segundo o H2, é a pesquisa sequencial.

3.4 Resumo

Neste capítulo foi apresentada uma visão geral do planejamento e execução de consultas no SGBDR H2. Foram identificados os principais componentes que compõem as camadas planejamento e execução de consultas. Discorreu-se inicialmente sobre a preparação dos planos de consulta, a representação utilizada para os planos e a sua execução propriamente dita. A seguir, apresentou-se as metodologias empregadas pelo H2 na resolução do problemas de ordenação de junções. Por fim, detalhou-se

o componente responsável pela tarefa de estimar o custo de uma solução, isto é, o custo de um plano de consulta.

4 Revisão

A revisão bibliográfica abordada neste capítulo discorre sobre pesquisas no âmbito da otimização do problema de ordenação de junções. Na Seção 4.1 são apresentados vários trabalhos que contribuíram para o estudo de soluções relacionadas ao problema em questão. A Seção 4.1.1 apresenta uma discussão acerca dos trabalhos revisados em relação às decisões tomadas na presente Tese.

4.1 Trabalhos Relacionados

O problema de ordenação de junções é antigo e amplamente estudado na literatura. No final da década de 70, foi apresentado por 1979 SELINGER et al.; um trabalho que se tornou referência para muitas pesquisas nesta área. Foram abordadas características de um SGBD relacional da época, o *System-R* (ASTRAHAN et al., 1976). Nesse trabalho foi apresentado um método de otimização para o problema de ordenação de junções baseado em programação dinâmica com complexidade de tempo $O(N!)$ relativa ao número de relações. Além disso, o autor apresentou um modelo de custo para os planos de execução que inclui análise da utilização de *CPU* e *I/O*, bem como formas para se tratar o custo de expressões ligadas por meio de conectores lógicos *AND* e *OR*.

No trabalho desenvolvido por 1984 IBARAKI; KAMEDA;, os autores demonstraram que o problema de ordenação de junções com o objetivo de minimizar o número de páginas acessadas por meio das consultas, ou seja, diminuir a taxa de *I/O* gerada pela execução de uma consulta, é um problema da classe NP-Completo. Ressalta-se que os autores utilizaram como método de junção apenas a implementação *Nested-Loop Join* (Subseção 2.1.1). Foi discutida uma nova estrutura para o método de junção utilizado evitar leituras desnecessárias de páginas. Também apresentou-se dois algoritmos A e B, aplicados na minimização da função de custo definida para o problema em questão. Com base em um número N de relações presentes na consulta, o algoritmo A possui uma complexidade de tempo $O(N^3)$ e utiliza no processo de otimização uma estrutura de grafo rotulada pela seletividade dos predicados de junção envolvendo os pares de relações ou pares de nós do grafo. Já o algoritmo B, possui complexidade de tempo $O(N^2 \log N)$ e é aplicável apenas em consultas acíclicas ou *tree queries*. Em relação ao algoritmo A, é importante observar que o mesmo não garante o retorno da solução ótima, ao passo que o algoritmo B dá tal garantia.

Um algoritmo com complexidade de tempo quadrática $O(N^2)$, conhecido como KBZ, foi desenvolvido em 1986 KRISHNAMURTHY; BORAL; ZANIOLO;. Os autores

tomaram como base o método proposto por 1984 IBARAKI; KAMEDA;. O espaço de soluções se restringiu à árvores de crescimento à esquerda e o custo baseou-se em operações realizadas em memória principal, ou seja, custo de operações de *CPU*. Apenas o método de junção *Nested-Loop Join* foi utilizado. O algoritmo trabalha com uma representação de árvore de junções com o nó raiz pré-fixado (*rooted join tree*). Além disso, foi definida uma função de classificação para auxiliar na definição da melhor ordem de junção entre as relações. Assim, dois componentes principais podem ser citados, o primeiro é aplicado na definição da melhor ordem de junção de uma solução parcial com o nó raiz pré-definido, tal definição se apoia na função de classificação definida. O outro componente é o método principal, e que de fato explora o espaço de soluções. Em resumo, o algoritmo recebe um grafo de junções acíclico como entrada e trata de encontrar a melhor ordem para as relações por meio de chamadas ao primeiro componente com diferentes nós raízes selecionados.

Em 1987 IOANNIDIS; WONG; foi desenvolvido um algoritmo baseado na meta-heurística Recozimento Simulado (*Simulated Annealing –SA*) para otimização de consultas recursivas em SGBDs. A otimização nesse trabalho ignorou algumas variáveis de decisão comumente presentes no problema de ordenação de junções, dentre elas: métodos de acesso das relações, métodos de junção e a possibilidade de paralelização de algumas operações. O custo dos estados ou soluções foi baseado em um modelo simples de avaliação de operações de *I/O* em SGBDs. A vizinhança adotada para exploração do espaço de soluções se baseou em propriedades tais como associatividade e comutatividade dos operadores estudados (Junção, Projeção e União). Os experimentos trataram a otimização de algumas expressões e segundo os autores, o algoritmo foi capaz de encontrar as soluções ótimas em alguns dos problemas e se aproximou do ótimo em outros. Contudo, questionou-se a efetividade dos movimentos empregados para a avaliação do espaço de soluções.

Vários algoritmos foram aplicados na otimização do problema de ordenação de junções em 1988 SWAMI; GUPTA;. Para avaliação do espaço de soluções foram adotados movimentos de troca entre duas e três relações distintas. Incluindo as variações da meta-heurística Recozimento Simulado, foram testados um total de 6 algoritmos:

- ***Perturbation Walk (PW)*** - Realiza uma pesquisa no espaço de soluções por meio de avaliações de vizinhos selecionados aleatoriamente e retorna a melhor solução encontrada. Este método aplica a mesma estratégia utilizada no método de descida randômica;
- ***Quasi-random Sampling (QS)*** - Se assemelha ao *PW*. Entretanto, não avalia iterativamente soluções adjacentes (vizinhos) e sim outras soluções geradas aleatoriamente. Ao final da execução retorna a melhor solução visitada;

- ***Iterated Improvement (II)*** - Utiliza o método de descida completa para alcançar um mínimo local. Após isto, seleciona uma nova solução aleatoriamente e aplica novamente a busca local. Atendido o critério de parada, a melhor solução encontrada é retornada;

- ***Sequence Heuristic (SH)*** - Tal como no *II*, o método de descida completa é utilizada na busca local. Contudo, após encontrado o mínimo local, a seleção da próxima solução a ser refinada não é realizada aleatoriamente, e sim por meio de uma quantidade de movimentos na solução atual;

- ***Simulated Annealing (SA)*** - Aplica a conhecida técnica Recozimento Simulado. São utilizadas duas variações do Recozimento Simulado no trabalho, a variação SAJ (JOHNSON et al., 1989) e a SAH (HUANG; ROMEO; SANGIOVANI-VINCENTELLI, 1986).

Utilizou-se um planejamento fatorial de efeitos fixos para calibragem dos parâmetros dos algoritmos e a análise dos resultados baseou-se na análise de variância (ANOVA (MONTGOMERY, 2008)). Ressalta-se o desenvolvimento de uma metodologia para avaliação dos algoritmos, a qual é descrita na Seção 6.1. Foram realizados experimentos com problemas envolvendo de 10 a 100 relações e segundo, os autores, o algoritmo *II* foi superior em relação aos outros testados.

É apresentada em 1989 SWAMI; uma extensão do trabalho de 1988 SWAMI; GUPTA;. De acordo com os autores, três tipos de heurísticas foram utilizadas:

- ***Augmentation Heuristic*** - Heurística gulosa que trabalha com a construção incremental da solução por meio de inserções de relações na solução corrente de acordo com alguns critérios classificação (ex.: classificação por meio da cardinalidade das relações). Dentre os critérios avaliados, o que minimiza a seletividade do predicado de junção se mostrou melhor;

- ***KBZ Heuristic*** - Mesma metodologia descrita anteriormente proposta por 1986 KRISHNAMURTHY; BORAL; ZANIOLO;;

- ***Local Improvement*** - Realiza o refinamento de blocos de relações. Assim, dadas todas permutações válidas entre m relações, seleciona-se a melhor dentre elas. O processo é repetido para o próximo grupo de m relações e assim por diante.

Tal como em 1988 SWAMI; GUPTA;, os métodos *II*, *SA* e suas variações foram testados em 1989 SWAMI;. No total, 9 algoritmos foram implementados, sendo eles:

- ***SA*** - Recozimento Simulado;

- ***II*** - Melhora iterativa;
- ***SAA*** - Recozimento Simulado que utiliza a heurística *augmentation* para gerar a solução inicial;
- ***SAK*** - Recozimento Simulado que utiliza a heurística *KBZ* para gerar a solução inicial;
- ***IAI*** - Melhora iterativa que utiliza como ponto de partida para o refinamento uma solução gerada pela heurística *augmentation*;
- ***IKI*** - Melhora iterativa que utiliza como ponto de partida para o refinamento uma solução gerada pela heurística *KBZ*;
- ***IAL*** - Se assemelha ao método *IAI*, exceto pelo fato de que a heurística *local improvement* é aplicada ao melhor mínimo local visitado por *IAI*;
- ***AGI*** - Neste método, a heurística *augmentation* é aplicada para gerar diferentes soluções. Caso o critério de parada não tenha sido atendido, soluções aleatoriamente geradas são submetidas ao II. Ao final, a melhor solução encontrada é retornada;
- ***KBI*** - Se assemelha ao *AGI*, exceto que utiliza a heurística *KBZ* para gerar as soluções.

Em relação à metodologia experimental (mais detalhes na Seção 6.1), foram otimizados problemas envolvendo de 10 a 100 relações e utilizou-se dois modelos de custo diferentes, um que realiza os cálculos da função de custo com base na quantidade de operações de *CPU* em memória principal e outro na quantidade de operações de *I/O* (memória secundária). De acordo com os resultados, combinações com o método *II* e a heurística *augmentation* retornaram melhores resultados.

Com base nos algoritmos *II* e *SA*, foi proposto em 1990 IOANNIDIS; KANG; um algoritmo denominado *2PO* que faz uso das duas técnicas citadas. O algoritmo divide a otimização em duas fases diferentes. Na primeira, aplica-se o método *II* por um determinado tempo. A seguir, na segunda fase, executa-se a meta-heurística *SA*. O modelo de custo definido baseia seus cálculos na taxa de *I/O* requerida para a execução das soluções geradas. Foram utilizados os métodos de junção *Nested-loop join* e *Merge-Sort Join*. Para explorar o espaço de soluções definiu-se cinco movimentos diferentes, sendo eles:

1. Troca do método de junção - $A \bowtie_1 B \rightarrow A \bowtie_2 B$;
2. Troca de posição entre duas relações numa junção - $A \bowtie B \rightarrow B \bowtie A$;

3. Com base na propriedade de associatividade, troca-se a ordem de execução das junções - $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$;
4. Troca de junção à esquerda - $(A \bowtie B) \bowtie C \rightarrow (A \bowtie C) \bowtie B$;
5. Troca de junção à direita - $A \bowtie (B \bowtie C) \rightarrow B \bowtie (A \bowtie C)$;

Na seção experimental do trabalho foi proposta uma nova metologia de avaliação (uma avaliação mais detalhada é feita na Seção 6.1) dos 3 algoritmos citados no problema estudado. Tal metodologia envolveu a otimização de problemas com 6 a 101 relações. Segundo os resultados dos experimentos, o método *2PO* se mostrou melhor que os algoritmos *II* e *SA*, tanto no custo das soluções geradas, quanto no tempo gasto para otimização.

Um algoritmo genético aplicado à ordenação de junções foi desenvolvido em 1991 BENNETT; FERRIS; IOANNIDIS;. Considerou-se dois tipos de representação para o espaço de soluções: árvore de crescimento à esquerda (*left-deep tree* - LT) e árvores fechadas (*bushy tree* - BT). Adotou-se um esquema de vizinhança denominado *ring6*, o qual foi utilizado na definição da aptidão dos indivíduos e funcionamento do operador de cruzamento. Os cálculos de aptidão dos indivíduos foram restritos apenas a vizinhos e um método de escala linear foi utilizado para tal operação. A população inicial é gerada de forma aleatória. Para o operador de mutação na representação LT, utilizou-se duas técnicas: troca do método de junção e troca de posição entre duas relações adjacentes. Já na representação BT, as seguintes estratégias foram adotadas: troca do método de junção, troca da orientação utilizada para a definição da relação externa/interna na junção e uma reordenação dos genes da solução. Foram definidos dois métodos para a operação de cruzamento:

- ***Modified two swap* (representação LT)** - Dados dois indivíduos *A* e *B*, seleciona-se dois genes de *A* e substitui-se os mesmos em *B* mantendo a ordem de seleção de *A* de forma a se criar um novo descendente;
- ***CHUNCK* (representação BT)** - Dados dois indivíduos *A* e *B*, seleciona-se uma parte da solução de *A* e copia-se para a mesma posição no descendente. Para o restante dos genes, copia-se os de *B* mantendo a mesma ordem e eliminando os genes previamente inseridos.

O processo de seleção de indivíduos para posterior reprodução baseou-se no método da roleta. O critério adotado para composição das novas populações apoiou-se na substituição do pior indivíduo da população pelo descendente gerado (metodologia elitista). Os experimentos computacionais utilizaram consultas envolvendo de 6 a 17 relações, e custo das soluções baseou-se nas operações de *I/O*. Comparou-se

o algoritmo genético com o método exato proposto por 1979 SELINGER et al.;. De acordo com os resultados, o algoritmo desenvolvido se mostrou uma opção viável em relação ao de programação dinâmica e foi capaz de encontrar soluções para problemas mais complexos com qualidade comparável e em menor tempo.

Com vistas a resolver algumas limitações do algoritmo KBZ (KRISHNAMURTHY; BORAL; ZANIOLO, 1986), os autores em 1993 SWAMI; IYER;, propuseram uma extensão do algoritmo em questão com uma série de melhorias e complexidade de tempo $O(N^4)$. A extensão proposta combina randomização, pesquisa no espaço de soluções com base em vizinhanças e o algoritmo KBZ. Os métodos de junção *Nested-Loop Join* e *Merge-Sort Join* foram utilizados. Consequentemente, o modelo de custo foi adaptado para contemplar também o uso do *Merge-Sort Join*. A execução do algoritmo pode ser distribuída em cinco passos:

1. **Método randômico** - Para uma dada solução inicial ou grafo de junção, associa-se aleatoriamente o método de junção para as arestas do grafo. Ressalta-se, que este método é aplicado a várias soluções de partida;
2. **KBZ** - Aplica-se o algoritmo KBZ no resultado do passo 1;
3. **Mudança de ordem** - Aplica-se mudanças na ordem da solução por meio de mudanças de posição entre relações;
4. **Mudança de método de junção** - Percorre-se a solução modificando o método de junção das arestas na tentativa de minimizar o custo;
5. **Pós-processamento** - Novamente, a ordem das relações é alterada buscando a minimização do custo.

A fase experimental do trabalho operou sobre 200 problemas gerados aleatoriamente envolvendo a junção de 2 a 16 relações. Comparou-se os resultados dos seguintes algoritmos: AB, Guloso, Programação Dinâmica estilo *System-R*, KBZ, II e 2PO. Com base nos resultados, o algoritmo AB foi capaz de obter em média, soluções com custos menores que os outros métodos não-exaustivos. Utilizando menos memória e menos tempo de CPU, observou-se que os custos das soluções do AB foram quase tão bons quanto os do algoritmo de programação dinâmica.

No trabalho de 1993 GRAEFE; MCKENNA; foi proposto um novo tipo de mecanismo para otimização de consultas, o qual independe do modelo de dados. Esse trabalho faz parte de um projeto chamado *Volcano*, que provê um conjunto de ferramentas extensíveis e eficientes para o processamento de consultas. A abordagem utilizada para a construção do otimizador exige que sejam fornecidas informações acerca

da especificação do modelo de dados, dentre elas: conjunto de operadores lógicos, regras de transformação algébrica, algoritmos para os operadores lógicos (operadores físicos) e modelos de custo relacionados. Com base nas informações passadas, é produzido um otimizador de consultas baseado em programação dinâmica. Tal algoritmo utiliza uma abordagem *top-down*, que possibilita o reconhecimento do custo planos completos antes que na abordagem *bottom-up*. Esta característica permite a utilização de estratégias de eliminação de soluções parciais, e tal como na metodologia *branch-and-bound*, a avaliação de soluções que ultrapassem um limite corrente estabelecido, pode ser descartada. Além disso, os autores definiram um modelo simples composto por basicamente operadores lógicos de seleção, junção, algoritmo para leitura das relações e métodos de junção. Diante disso, tal especificação foi empregada na comparação do otimizador *Volcano* com outro projeto dos autores conhecido como *EXODUS* (GRAEFE; DEWITT, 1987). De acordo com os resultados, experimentos envolvendo consultas com 2 a 8 relações mostraram que o *Volcano* obteve tempos de otimização melhores e soluções de melhor qualidade.

O uso da meta-heurística Busca Tabu (GLOVER; LAGUNA, 1997) aplicado ao problema de ordenação de junções foi discutido em 1995 MATYSIAK;. A avaliação do espaço de soluções se baseou na vizinhança de troca, sendo utilizados três tipos diferentes de movimentos:

- **Troca do método de junção** - Altera o método de junção variando entre *Nested-Loop Join* e *Merge-Sort Join*;
- **Troca de argumento do método de junção** - Troca a posição das relações que compõem a junção.
- **Troca de operador de junção** - Dadas duas operações de junção distintas, onde uma seja argumento para outra, este movimento consiste em trocar as mesmas de posição.

Adotou-se como ponto de partida para o método, um ótimo local obtido por meio da execução do método de Descida Completa em uma solução aleatória. Configurou-se a lista tabu para armazenar o movimento aplicado para mudança do estado atual, ou seja, o movimento aplicado na solução corrente para seleção do novo vizinho. As operações de *I/O* foram priorizadas para definição do custos. Os experimentos compararam as soluções obtidas por três métodos: Busca Tabu, Recozimento Simulado e Melhora Iterativa. De acordo com os resultados dos testes envolvendo 11 a 101 relações, a Busca Tabu foi capaz de obter soluções melhores em quase todos os problemas. Contudo, para problemas de menor complexidade, o algoritmo proposto foi pior.

Um algoritmo exaustivo baseado em programação dinâmica, denominado *blitzsplit*, foi desenvolvido por 1996 VANCE; MAIER;. Diferentemente de outros trabalhos citados, os quais limitam o espaço de busca por meio da adoção de algumas restrições como as citadas na Seção 2.2, este algoritmo considera o uso de planos cartesianos se necessário. A complexidade de tempo do algoritmo é $O(3^N)$ e a complexidade de espaço é $O(2^N)$, sendo N o número de relações. Nos experimentos computacionais foram otimizados problemas com até 15 relações e utilizou-se três modelos de custo diferentes. Além disso, a construção dos problemas-teste seguiu a topologia de quatro tipos de grafo (CORMEN et al., 2001), sendo eles: corrente, circulo+3, estrela e clique. Maiores informações acerca da metodologia experimental desenvolvida são apresentadas na Seção 6.1. Segundo os autores, o algoritmo desenvolvido foi capaz de gerar soluções rapidamente e pouco esforço foi despendido para eliminar soluções muito custosas. Contudo, para situações onde as soluções possuem custos muito parecidos, o algoritmo requer mais esforço para sua execução. Também é relatado no trabalho que confinar o espaço de busca para consultas com até 15 relações a árvores de crescimento à esquerda podem contribuir pouco no ganho em tempo de otimização.

Um comparativo entre métodos heurísticos, randômicos e o algoritmo genético é apresentado em 1997 STEINBRUNN; MOERKOTTE; KEMPER;. Nesse trabalho foram explorados os seguintes algoritmos: Programação Dinâmica estilo *System-R*, KBZ, Seletividade Mínima, SA, II, 2PO e Genético. A heurística Seletividade Mínima se assemelha a *Augmentation Heuristic* (SWAMI, 1989) e utiliza como critério de classificação a seletividade da junção entre duas relações. Os métodos determinísticos executaram no espaço de soluções restritos à árvores de crescimento à esquerda, enquanto que alguns dos randômicos/genético operaram também no espaço de árvores fechadas. Foi utilizado um modelo de custo baseado na taxa de *I/O*. Os experimentos computacionais trataram problemas envolvendo de 5 a 30 relações. Guiou-se a construção do grafo de junções (CORMEN et al., 2001) dos problemas com os seguintes formatos: corrente, círculo, grade e estrela. Tal como em 1988 SWAMI; GUPTA;1990 IOANNIDIS; KANG;, foram implementadas variações para os métodos II e SA. De acordo com os resultados, os autores constataram que os algoritmos exaustivos são mais indicados para problemas menos complexos, que envolvam de 10 a 15 relações e operem no espaço de soluções restrito a árvores de crescimento à esquerda. Quanto ao KBZ, ele se mostrou mais competitivo no tempo de otimização apenas para problemas com o formato estrela. Para o espaço de árvores fechadas, os algoritmos exaustivos se mostraram desejáveis para problemas com até 7 relações. Ainda em relação a este espaço, verificou-se que os algoritmos randômicos e o genético são mais adequados. Ressalta-se que no tocante à qualidade das soluções e tempo de otimização, o 2PO apresentou melhor custo benefício. O SA foi capaz de obter melhores soluções mas gastando um tempo maior de otimização. Objetivando tempos de

otimização menores, os algoritmos II, Genético e 2PO se mostraram boas alternativas.

Uma interessante extensão do algoritmo clássico de programação dinâmica (SELINGER et al., 1979) (DP) foi apresentado em 2000 KOSSMANN; STOCKER;. O algoritmo clássico é capaz de retornar ótimos resultados, mas a medida em que a complexidade dos problemas aumenta, sua execução se torna proibitiva. Assim, de forma a contornar tal limitação, os autores desenvolveram uma técnica baseada em programação dinâmica e algoritmo guloso, denominada *Iterative Dynamic Programming* (IDP). Foram estudadas um total de 8 variações do IDP, mas a ideia básica é composta por três componentes:

1. **Algoritmo de Programação Dinâmica** - Considerando N relações e um limite K, onde $K \leq N$. O método é aplicado até que soluções com K relações sejam formadas. Observa-se que quando $K = N$, IDP executa da mesma maneira que DP;

2. **Metodologia Gulosa** - Quando $K < N$, o DP interrompe sua execução e uma abordagem gulosa é aplicada. Portanto, dentre todas as K-soluções, escolhe-se aquela melhor classificada de acordo com algum critério. Após isso, o DP retoma sua execução, mas fixa a solução parcial escolhida, ou seja, a construção dos blocos de solução pelo DP sempre irão considerar a solução parcial por inteiro, tal como escolhida na fase gulosa;

3. **Função de Classificação** - A fase gulosa do IDP pode usar vários critérios para selecionar a melhor solução parcial da próxima rodada, dentre eles: custo do plano, resultado intermediário e seletividade dos predicados. Os autores distribuíram os critérios de classificação em três grupos diferentes, que basicamente aplicam heurísticas distintas para a classificação das soluções.

Diante do que foi apresentado, as variações se diferem na ordem de execução do DP e metodologia gulosa, critério de avaliação, número de soluções parciais escolhidas durante as iterações do método e número de relações avaliadas em cada fase do algoritmo. Os experimentos computacionais focaram ambientes distribuídos e centralizados. O modelo de custo adotado foi o mesmo proposto por 1997 STEINBRUNN; MOERKOTTE; KEMPER;, mas estendido para bancos de dados distribuídos. Mais ainda, foram explorados problemas com 10 e 20 relações nos formatos corrente e estrela. Duas variações do IDP foram escolhidas e comparadas ao clássico DP e uma implementação do 2PO. De acordo com os resultados , o 2PO apresentou bons tempos de otimização, mas foi capaz de retornar planos aceitáveis, apenas em ambientes centralizados. Já o IDP, obteve tempos de otimização menores que DP tanto para 10

relações, quanto para 20. Vale ressaltar, que devido à complexidade dos problemas com 20 relações, o DP não foi utilizado nesses experimentos.

Um comparativo entre o uso de abordagens *bottom-up* e *top-down* em otimizadores de consultas é apresentado por 2001 SHAPIRO et al.;. Foi desenvolvido um algoritmo denominado *columbia* para o problema de ordenação de junções que utiliza a abordagem *top-down*. Ele se apoia em algumas estruturas de dados para sua execução, dentre elas:

- **Grupo** - Define uma classe de equivalência de expressões que produzem o mesmo resultado, como por exemplo permutações diferentes entre as mesmas relações;
- **Multi-expressões** - Consiste em um operador de junção que tem como entrada grupos de expressões.

O algoritmo desenvolvido emprega o uso de limites superior e inferior. Tais limites permitem eliminar grupos de expressões que não fazem parte da solução ótima. É importante ressaltar o uso do princípio da otimalidade para realização das podas de soluções. O método parte de um conjunto de multi-expressões e recursivamente encontra os vencedores do grupos. Grupos que extrapolam o limite superior são eliminados. Mais ainda, o limite inferior também permite a eliminação de avaliações sem perda de qualidade. Diferente de outros trabalhos citados, que na maioria avaliam o custo dos planos gerados, bem como o tempo gasto na otimização, este trabalho analisa o número de soluções ou multi-expressões avaliadas durante o processo de otimização. Foram geradas consultas envolvendo de de 4 ate 13 relações no formato dos grafos corrente e estrela (CORMEN et al., 2001). Os autores compararam o algoritmo proposto com outro da literatura, o *starburst*. Segundo eles, o *columbia* foi capaz de encontrar a solução ótima reduzindo a avaliação de multi-expressões de 60% a 98%.

No trabalho de 2007 GUTTOSKI; SUNYE; SILVA; é apresentada uma implementação do algoritmo de *Kruskal* (CORMEN et al., 2001) para a otimização de consultas em SGBDs. O algoritmo proposto foi incluído no SGBD relacional *PostgreSQL*[1] e comparado com outros dois métodos disponíveis neste SGBD, um de programação dinâmica e um algoritmo genético. Devido ao fato do algoritmo proposto ter sido implementado no *PostgreSQL*, o modelo de custo utilizado foi o fornecido pelo SGBDR, o qual leva em consideração, tanto operações de *CPU*, quanto de *I/O*. Também são disponibilizados pelo banco os três métodos de junção citados na Subseção 2.1.1. A

[1] PostgreSQL: www.postgresql.org

versão do SGBDR testada foi a $8.2 - devel$ e os experimentos computacionais utilizaram o *benchmark*[2] sintético TPC-H disponibilizado pela organização TPC [3], o qual é descrito na Seção 6.1. Ressalta-se que na versão do SGBDR utilizada, para consultas com até 12 relações o algoritmo de programação dinâmica é empregado, ao passo que para mais relações o genético é utilizado. Diante disto, foi necessário gerar consultas adicionais para comparação com o genético, devido ao fato de o TPC-H não fornecer consultas envolvendo o número mínimo de relações necessárias para execução do genético. Com base nos experimentos, verificou-se que o algoritmo proposto obteve bons resultados, inclusive alcançando soluções em tempos melhores ou quase próximos ao dos outros algoritmos.

Foi desenvolvido em 2007 DONG; LIANG; um algoritmo genético para tratar a otimização de consultas com muitas junções entre relações. Nesse trabalho foram codificadas quatro classes do algoritmo genético:

1. **Algoritmos genéticos básicos** - Esta classe define algoritmos genéticos clássicos. Várias opções foram testadas para a substituição de indivíduos e operadores de mutação e cruzamento. A fase de calibração optou pela versão com inicialização da população aleatória, substituição do pior indivíduo com réplica ou do pior indivíduo, caso contrário. Os operadores de mutação e cruzamento selecionados foram: troca entre duas relações da solução (escolhidas aleatoriamente) e cruzamento uniforme baseado na ordem (*uniform order-based crossover*), respectivamente;

2. **Algoritmos genéticos com restrição** - Diferentemente da classe básica, esta não trabalha com soluções contendo planos cartesianos. Assim, optou-se por métodos livres de planos cartesianos na fase de inicialização e operadores de mutação e cruzamento;

3. **Algoritmos genéticos heurísticos** - Adotou-se nesta classe uma heurística para guiar a inicialização da população baseada na cardinalidade das relações. Ressalta-se, ainda, a utilização das mesmas opções da classe 1 para as demais configurações;

4. **Algoritmos genéticos com busca local** - Cada descendente gerado nesta classe passa por um método de refinamento, o qual utiliza a vizinhança de troca para exploração do espaço de soluções. Ressalta-se que foram utilizados movimentos de troca entre duas e três relações durante a calibração. Optou-se pela

[2] *Benchmark* é um processo de avaliação que se apoia na execução de um conjunto de ações que buscam analisar o comportamento de um dado programa em relação a um conjunto de métricas.

[3] TPC: http://www.tpc.org

configuração do algoritmo com as mesmas escolhas da classe 1, acrescentando o refinamento com movimentos de troca entre três relações.

Os testes experimentais utilizaram consultas com 10 a 100 relações. O modelo de custo adotado considerou apenas as operações de disco em seus cálculos e o espaço de soluções limitou-se a árvores de crescimento à esquerda. Além disso, os problemas gerados foram distribuídos em seis modelos diferentes, três deles utilizando as ideias propostas em 1988 SWAMI; GUPTA;, e os outros três construídos com base em grafos do tipo estrela e duas variações do tipo estrela (CORMEN et al., 2001). Os autores concluíram que a eficiência do algoritmo está relacionada às diferenças entre os operadores/vizinhança adotados e os modelos de consultas, e a eficiência dos parâmetros também está relacionada aos modelos de consultas.

A tese de doutorado de 2007 MULERO; propõe um otimizador para o problema de ordenação de junções denominado *Carquinyoli Genetic Optimizer* – CGO. De acordo com o autor, o CGO baseia-se em conceitos de uma metodologia evolucionária conhecida como Programação Genética – GP. Embora seja baseado em GP, o CGO utiliza apenas operadores de cruzamento, mutação e seleção. A tese discute inicialmente uma versão básica do CGO (apenas operadores de cruzamento, mutação e seleção) e adiciona iterativamente novas características ao método. Assim, o CGO e suas variações podem ser distribuídos entre diferentes abordagens para os métodos de cruzamento, mutação, seleção de pares para cruzamento, construção da população inicial, heurísticas de transformação nos planos de consulta. Diante disso, a listagem seguinte resume um conjunto de metodologias que possibilita caracterizar todas as variações estudadas na tese:

- **Cruzamento** - O autor definiu duas metodologias diferentes. Primeiro tomam-se dois indivíduos P_1 e P_2 para gerar dois descendentes D_1 e D_2. A solução de D_1 é formada com base na seleção aleatória de parte da solução de P_1. A seguir, o restante da solução é carregado com as informações de P_2 não presentes em D_1, vale ressaltar que leitura em P_2 é feita em pós-ordem. A composição de D_2 é análoga a D_1, só que a solução é inicialmente carregada com P_2. A outra metodologia é chamada *Increasing Intensive Crossovers* – IIC, que basicamente aumenta o número de operações de cruzamento por iteração, isto é, dados dois ascendentes, aplica-se N operações de cruzamento (a mesma metodologia anterior) numa única iteração, o que leva a um total de $2 \times N$ descentes por operação ou iteração;

- **Mutação** - O CGO possui 5 métodos de mutação: troca de posição entre duas relações da solução, troca do método de acesso das relações, troca do método de junção, troca entre duas operações de junção imediatamente subsequentes e

geração aleatória de subárvore, onde seleciona-se randomicamente uma parte da solução de um dado indivíduo e insere-se as demais informações aleatoriamente;

- **Seleção dos Pares de Cruzamento** - A metodologia básica seleciona dois indivíduos aleatoriamente. Numa outra abordagem, a WE, utiliza-se o custo das soluções individuais para gerar a probabilidade de escolha das mesmas, ou seja, privilegiar a escolha de indivíduos com a aptidão pior avaliada e acelerar a convergência;

- **Construção da População Inicial** - São aplicadas duas técnicas. Na primeira, gera-se a população por meio da inclusão de indivíduos construídos aleatoriamente. Já na segunda, denominada HIP, a construção é realizada por um tempo pré-definido e emprega a heurística *Augmentation Heuristic* (SWAMI, 1989) para geração dos indivíduos;

- **Heurísticas de Transformação** - Foram apresentadas uma série de regras de transformação para melhorar as soluções (GEO). Tais regras se apoiam em características dos operadores físicos de seleção e junção. Assim, adotou-se a estratégia de aplicar as transformações possíveis nas soluções após a fase de mutação, que por sua vez antecede a seleção dos indivíduos para nova geração.

O algoritmo CGO foi testado num ambiente de simulação construído para este propósito. O ambiente é composto por alguns componentes que fornecem funcionalidades básicas para a otimização de junções. Além disso, considerou-se apenas a análise da qualidade das soluções e o tempo de otimização, sendo adotado um modelo de custo baseado em operações de *CPU* e *I/O*. A calibração de alguns dos parâmetros do CGO fundamentou-se em resultados de um planejamento fatorial e análise estatística da variância (ANOVA). A avaliação experimental foi dividida em vários experimentos, partindo de uma observação inicial do CGO básico a uma comparação final entre as variações do CGO e uma versão do 2PO. Quanto ao experimento final, foram estudadas as seguintes variações/métodos: CGO, CGO + WE, CGO + HIP, CGO + HIP + WE, GEO, GEO + WE, GEO + HIP, GEO + HIP + WE, 2PO e IAI (SWAMI, 1989). Esses métodos foram aplicados a problemas-teste envolvendo 50 relações no formato estrela em um banco de dados gerado aleatoriamente. Adotou-se como critério de parada para o processo de otimização um limite de 270 segundos. Os resultados após 30 segundos indicaram que o 2PO não é capaz de convergir tão rapidamente quanto os outros algoritmos, e que, em geral, as metodologias HIP, WE e GEO beneficiam a qualidade das soluções, sendo as variações GEO, GEO+HIP e CGO+HIP+WE responsáveis pelas soluções de melhor qualidade. A evolução observada entre 30 e 270

segundos mostra, claramente, que as variações GEO+WE e GEO+HIP+WE produzem soluções com os menores custos. Por fim, é válido notar que o texto da tese relaciona-se a vários trabalhos publicados pelo mesmo autor (MUNTES-MULERO; ZUZARTE; MARKL, 2006; MUNTéS-MULERO et al., 2006c; MUNTéS-MULERO et al., 2006; MUNTéS-MULERO et al., 2006b; MUNTéS-MULERO et al., 2006a; MUNTéS-MULERO et al., 2007).

Um modelo de paralelização do planejamento de consultas foi desenvolvido em 2008 HAN et al.;. O artigo citado apresentou uma forma de subdividir o problema de ordenação de junções e executar de forma paralela os subproblemas. Foi desenvolvida uma versão paralela do algoritmo clássico de programação dinâmica. Estudou-se várias metodologias para se dividir a avaliação do espaço de soluções entre diferentes *threads*. Além disso, foi introduzida uma estrutura chamada de *skip vector array*, que melhorou o desempenho do algoritmo por meio da eliminação eficiente de avaliações de soluções inválidas. O algoritmo proposto foi implementado no SGBDR *PostgreSQL*, versão 8.3. Os experimentos utilizaram problemas-teste envolvendo de 10 a 20 relações. Foram avaliadas consultas com as seguintes topologias: corrente, círculo, estrela e clique. Contudo, apenas os formatos estrela e clique foram discutidos, pois os outros formatos não apresentaram benefícios com a paralelização. Várias análises foram realizadas, dentre elas: modelos de divisão do espaço de busca, algoritmos/-variações implementados, formato das consultas e *speedup*. Focando a comparação dos diferentes algoritmos/variações desenvolvidos e o algoritmo de programação dinâmica do SGBDR, observou-se que a variação mais eficiente do algoritmo proposto foi capaz melhorar o tempo de otimização em até 133,3 vezes no formato estrela e no formato clique em até 547 vezes. Em relação à métrica *speedup*, o algoritmo obteve um *speedup* linear no formato estrela para problemas com 20 relações, obtendo um ganho de 100%. Já no formato clique, o algoritmo obteve um ganho médio melhor que no formato estrela. Contudo, alcançou o ganho máximo de 87,5% para 20 relações.

Na dissertação de mestrado de 2010 LANGE; foram revisados vários métodos da literatura aplicados à ordenação de junções. O autor implementou o algoritmo *2PO* proposto por 1990 IOANNIDIS; KANG; e o comparou com um algoritmo genético presente no SGBDR *PostgreSQL*. Foi proposta uma nova metodologia para geração do roteiro de consultas-teste aplicado na comparação dos algoritmos. De acordo com os resultados, a implementação *2PO* mostrou maior robustez em grande parte dos casos testados.

Tendo em consideração a otimização de consultas em bancos de dados distribuídos, destaca-se o estudo de 2011 SEVINC; COSAR;, onde foi desenvolvido um algoritmo genético denominado *NGA*. Tal algoritmo é composto por duas novas metodologias de cruzamento e mutação, denominados *new-crossover* e *new-mutation*,

respectivamente. O operador de cruzamento utiliza a estratégia de um ponto de corte para composição da solução descendente. A primeira parte da solução é carregada com a subsequência de genes consecutivos com o menor custo de junção de um dos ascendentes. A seguir, o restante da solução é formado com os genes do outro ascendente na ordem de leitura da solução do mesmo. Já a mutação utiliza o custo de cada gene para definir a probabilidade de seleção do mesmo. Uma vez escolhido o gene, seleciona-se aleatoriamente uma posição na solução e efetua-se o movimento de troca de posição entre os genes. O modelo de custo baseia-se em operações de *CPU*, I/O e Rede. Três algoritmos foram comparados com NGA: uma abordagem exaustiva (ESA), uma outra implementação do GA (RHO; MARCH, 1997) e um método randômico, que gera aleatoriamente um número de soluções igual à quantidade de soluções gerada por NGA. De acordo com os resultados, o NGA encontrou soluções aceitáveis quando comparado ao ESA e ao GA. Além disso, o NGA foi capaz de gerar soluções ótimas em 80% dos casos e melhorou os resultados do GA em cerca de 50%.

Ainda no âmbito de banco de dados distribuídos, um otimizador baseado na meta-heurística Colônia de Formigas (*Ant-Colony - ACO* (DORIGO; MANIEZZO; COLORNI, 1996)) foi estudado por 2014 GOLSHANARA; ROUHANI RANKOOHI; SHAHHOSSEINI; na ordenação de operações de junções com dados replicados em múltiplos nós. A abordagem proposta emprega o uso de quatro tipos de colônias diferentes, o que caracteriza uma versão multi-colônia. Cada colônia possui um tipo específico de formiga que contribui na pesquisa pela solução ótima, são elas:

- ***JoinAnt*** - Determina a ordem das operações de junção;
- ***SiteAnt*** - Determina o nó onde cada operação de junção será executada;
- ***ReplicaAnt*** - Determina para cada relação o nó onde sua réplica será utilizada;
- ***Semi-Ant*** - Determina o tipo de metodologia *semi-join* que será aplicada para cada operação de junção (sem *semi-join*, primeira relação enviando os dados do atributo de junção para o site da segunda relação ou a segunda relação enviando para a primeira).

Foram utilizados dois modelos de custo. O primeiro relaciona o custo ao total resultante da soma de todas as métricas envolvidas (*CPU*, *I/O* e *Rede*). O segundo é associado ao intervalo de tempo entre a submissão da consulta e o retorno da resposta, ou seja, tempo de resposta. A avaliação experimental foi dividida em quatro experimentos, utilizando-se problemas no formato corrente, árvore (acíclico e conexo) e consultas cíclicas e o espaço de solução limitado a representações do tipo árvores fechadas e árvore de crescimento à esquerda. O método proposto foi comparado aos

algoritmos genéticos GA (RHO; MARCH, 1997) e NGA (SEVINC; COSAR, 2011). Pelos resultados apresentados, verificou-se que, no geral, o GA comportou-se melhor que o NGA, tanto na qualidade das soluções, quanto no tempo de otimização. Em relação ao ACO, observou-se tempos de otimização menores em cerca de 80% dos casos. Contudo, a qualidade das soluções do ACO foi cerca de 14,3% pior que a do GA.

O uso de algoritmos genéticos e sistemas multiagente na otimização de consultas em bancos de dados distribuídos foi proposto por 2008 GHAEMI et al.;. Os autores compararam o algoritmo com um de programação dinâmica e, segundo eles, o algoritmo proposto requisitou um tempo de processamento menor que o algoritmo de programação dinâmica. O seguintes agentes foram definidos:

1. **Query Distributor Agent (QDA)**. Divide a consulta em *sub-queries* e distribui para os devidos LOAs;
2. **Local Optimizer Agents (LOAs)**. Os agentes locais aplicam um algoritmo genético para otimização das *sub-queries* e retornam o resultado para o GOA;
3. **Global Optimizer Agent (GOA)**. Responsável por encontrar a melhor ordem de junção entre os *sites*. O objetivo deste agente é minimizar o custo de comunicação.

Uma extensão do algoritmo proposto por 2008 GHAEMI et al.; foi desenvolvida em 2010 ZAFARANI et al.;2010 FEIZI-DERAKHSHI; ASIL; ASIL;. Nesse trabalho focou-se em construir um sistema adaptativo, que reduziu em até 29% o tempo de resposta. Foi adicionado um novo agente além dos propostos por 2008 GHAEMI et al.;, sendo ele:

1. **Adaptative Query Agent (AQA)**. Este agente reconhece similaridade entre consultas. Em caso de haver similaridade, um plano de execução previamente armazenado é reutilizado; caso contrário, a consulta é normalmente executada e o agente posteriormente atualizado. O agente adaptativo também fica responsável pela substituição de planos armazenados.

No campo de otimização com fontes RDF, 2013 HOGENBOOM; FRASINCAR; KAYMAK; propôs um algoritmo fundamentado em colônia de formigas ACO. Adotou-se um modelo de custo apoiado no dispêndio relacionado a transferência dos dados da fonte para um nó processador, bem como o custo associado ao processamento dos dados. A metodologia proposta foi comparada a um GA (HOGENBOOM et al., 2009) e um 2PO (STUCKENSCHMIDT et al., 2005). Foram utilizados problemas-teste no formato corrente envolvendo de 2 a 20 operações de junção. De acordo com os autores,

o ACO foi significantemente melhor que o GA e o 2PO em termos de qualidade da solução e tempo de otimização. Para consultas com até 15 junções, o ACO requereu, aproximadamente, 80% de tempo a menos para obter as melhores soluções. Em problemas mais complexos, o ACO melhorou a qualidade das soluções em comparação ao 2PO e o GA, mas demandou mais tempo de otimização que o GA.

Mais recentemente, o uso da Unidade de Processamento Gráfico (*Graphics Processing Unit - GPU*) em SGBDRs foi abordado por 2012 HEIMEL; MARKL;2013 HEIMEL;. Realizou-se uma discussão acerca de ideias iniciais para o projeto de um otimizador de consultas assistido pela *GPU*. Mais ainda, os autores destacam planos futuros para o desenvolvimento de um otimizador baseado no método de programação dinâmica (SELINGER et al., 1979) integrado à unidade de processamento gráfico.

Durante a análise de SGBDRs para implementação do algoritmo desenvolvido neste trabalho, foi possível verificar a estratégia usada por alguns SGBDRs na otimização de consultas. Com base nisso, a lista seguinte resume a metodologia aplicada por alguns desses sistemas:

- **H2** - O otimizador[4] da versão $1.3.174$ - 2013 usa para consultas com até 7 relações, uma versão força bruta (Subseção 3.2); ao passo que para consultas com mais relações, uma mistura entre otimização exaustiva, metodologia gulosa e avaliação aleatória é empregada (Subseção 3.2);
- ***PostgreSQL*** - A versão $9.2.6$ - 2013 emprega um algoritmo estilo *Selinger* (SELINGER et al., 1979) para ordenação de junções com até 11 relações e utiliza uma Algoritmo Genético[5] para consultas envolvendo 12 ou mais relações;
- ***MySQL*** - O otimizador[6] utilizado na versão $5.1.72$ - 2013 baseia-se no método busca em profundidade (CORMEN et al., 2001);
- ***Derby*** - O SGBDR *Apache Derby*[7], versão $10.9.1.0$ - 2012, também utiliza uma técnica do tipo busca em profundidade.

4.1.1 Discussão

De acordo com o que foi apresentado no Capítulo 2, o processamento de consultas pode ser dividido entre o planejamento e a execução do plano definido. Esta revisão descreveu uma série de trabalhos com contribuições importantes no âmbito da otimização de consultas em banco de dados, ou seja, contribuições relacionadas à

[4] H2: www.h2database.com/html/performance.html
[5] *PostgreSQL* - www.postgresql.org/docs/9.2/interactive/geqo.html
[6] *MySQL* - http://dev.mysql.com/doc/internals/en/optimizer.html
[7] *Derby* - http://db.apache.org/derby/papers/optimizer.html

fase de planejamento. Assim, foram abordados algoritmos exatos, heurísticos, meta-heurísticos e evolucionários para a otimização do problema de ordenação de junções. Além disso, também discorreu-se acerca das metodologias empregadas por alguns SGBDs relacionais conhecidos.

A despeito das várias arquiteturas de banco de dados, geralmente quando se fala de paralelização em SGBDR, refere-se à execução paralela do plano otimizado, sendo o planejamento em si, executado sequencialmente por um nó coordenador. Tal como apontado em 2008 HAN et al.;, há um maior esforço no sentido de paralelização da fase de execução das consultas, e não na fase de planejamento, a qual inclui o problema de otimização foco deste trabalho. De acordo com a revisão, apenas 2008 HAN et al.; focou a paralelização do processo de otimização em SGBDRs. Diante disto, o presente trabalho preenche esta lacuna, uma vez que os agentes rodam em paralelo. Desta maneira, a arquitetura empregada para a otimização de consultas pode ser aplicada tanto na direção do desempenho, quanto na melhora da eficiência de avaliação do espaço de soluções.

No que diz respeito aos trabalhos de 2008 GHAEMI et al.;2010 ZAFARANI et al.;2010 FEIZI-DERAKHSHI; ASIL; ASIL;, onde a metodologia proposta mescla o uso de sistemas multiagente e algoritmos genéticos, vale notar que tal abordagem se difere totalmente do algoritmo desenvolvido nesta tese, descrito no Capítulo 5. Primeiro, o algoritmo proposto por esses autores foca um SGBDR distribuído; segundo, executa fora do núcleo de otimização do SGBDR; e, por último, os agentes empregam uma forma mais simplificada de interação, se comparado ao algoritmo desenvolvido nesta tese. Basicamente, um agente (QDA) divide a consulta em problemas menores e distribui cada parte do problema para análise em alguma fonte de dados com um agente local registrado (LOA). O agente LOA executa uma versão independente do GA para determinar a ordem de junção. Finalmente, o último agente (GOA) tenta minimizar o tráfego de rede, executando um plano parcial definido por algum LOA, enviando o resultado parcial pela rede para outra fonte de dados, e combinando o resultado parcial enviado com outro resultado definido por um LOA. O processo termina quando todos os resultados parciais são combinados.

Em relação à escolha por uma abordagem baseada em sistemas multiagente evolucionários, é digno de nota que a união de sistemas multiagente e algoritmos evolucionários permite uma exploração eficiente e decentralizada do espaço de soluções. Um agente pode executar operadores genéticos comprovadamente eficientes em vários problemas. Além disso, cada agente também pode influenciar diretamente na execução dos demais, pois algumas ações são tomadas em conjunto com os outros agentes. Tal metodologia possibilita a definição de perfis de exploração variados, os quais viabilizam diferentes objetivos e implicam em formas distintas de avaliação do

espaço de soluções. Ainda em relação aos perfis, os mesmos não são restritos apenas a ações presentes em algoritmos evolucionários, isto é, a composição dos perfis de exploração pode se apoiar em heurísticas construtivas, de refinamento, entre outras. Uma característica importante herdada da metodologia evolutiva, é o fato de que agentes mais aptos tendem a executar por mais tempo, ou seja, os agentes com soluções melhores podem diminuir o número de iterações disponíveis de agentes com a aptidão pior. Esse mecanismo de pressão seletiva implica em um balanceamento natural entre as estratégias de busca local e global, posto que os agentes necessitam obter boas soluções para sobreviverem, mas não podem gastar muito tempo com certas estratégias custosas, pois podem ter o número de iterações disponíveis reduzido antes do término de tais ações. Diante do que foi apresentado, é inegável o potencial da metodologia escolhida com respeito à possibilidade de geração de soluções de boa qualidade e tempos de execução aceitáveis. Portanto, é importante notar que as características citadas e o fato de não ter sido encontrado nesta revisão um trabalho que utilizasse sistemas multiagente evolucionários, configuraram-se como o grande motivador para o desenvolvimento do algoritmo proposto.

Uma outra observação notável na revisão, é o fato de que alguns dos estudos apresentados utilizaram para exploração do espaço de soluções, movimentos de troca do método de acesso às relações e algoritmos de junção. Durante a avaliação das técnicas empregadas por alguns SGBDRs reais (H2, HSQL, Derby, MySQL e PostgreSQL), observou-se que tal movimento é desnecessário, implicando em desperdício de recursos computacionais. Tal afirmação se apoia no fato de que os otimizadores geram uma permutação e a repassam ao componente de custo, que por sua vez determina os meios de acesso das relações, métodos de junção, etc. Tal abordagem estabelece que, dada uma sequência ordenada e baseado nas métricas e fórmulas relacionadas, o componente de custo sempre escolherá os melhores algoritmos para as operações de seleção, junção, etc. Assim, qualquer troca na solução após a avaliação do componente de custo, implicará em piora da solução. Diante de tal informação, o uso de SGBDR real para implementação do trabalho se mostrou bastante pertinente, dada a maturidade dos componentes relacionados e a possibilidade de focar em movimentos eficazes na avaliação do espaço de soluções.

Também é válido destacar a disponibilidade do código-fonte de otimizadores propostos em algumas das pesquisas revisadas. Tal fator tem influência direta na escolha do SGBDR H2 em detrimento de trabalhos anteriores disponíveis. Pela análise realizada, verificou-se a existência de alguns projetos implementados no *PostgreSQL* (GUTTOSKI; SUNYE; SILVA, 2007; LANGE, 2010) e um ambiente de simulação para o problema de ordenação de junções (MULERO, 2007). Em relação ao *PostgreSQL* e de acordo com o que será observado no Capítulo 5, o algoritmo proposto depende do uso de *Threads* para sua execução. Assim, descartou-se o uso do *PostgreSQL*, visto

que o mesmo não oferece suporte a *Threads* ou a outro recurso equivalente. Por outro lado, o ambiente de simulação do CGO suporta o uso de *Threads*. Contudo, ele possui uma série de características que pesaram na escolha por um SGBDR real. A seguir, são destacados os principais pontos negativos observados no ambiente de simulação:

1. Não há a possibilidade de atestar a qualidade dos planos gerados, uma vez que só é possível planejá-los dentro do ambiente de simulação;
2. Os autores citam que o modelo de custo proposto apresentou em alguns momentos, erros significantes quando comparado ao modelo do IBM DB2;
3. Dificuldade na comparação com SGBDRs reais. O ambiente é focado apenas no IBM DB2 e acrescido de uma série de limitações;
4. O ambiente possui poucos algoritmos implementados para a realização de experimentos de comparação;
5. Necessidade de conversão das consultas SQL para uma representação interna adotada, o que aumenta ainda mais a dificuldade de integração com ambientes reais.

Por fim, concluiu-se que o ambiente de simulação não apresenta nenhuma vantagem expressiva em comparação ao SGBDR real escolhido. Dentre os vários aspectos positivos encontrados no H2, pode ser destacado que o padrão SQL é mantido com a utilização do H2. Mais ainda, experimentos envolvendo outros SGBDRs reais são perfeitamente possíveis e comparações com outros métodos no próprio H2, como em um ambiente de simulação, também é facilmente alcançável. O modelo de custo tem uma certa maturidade e não há a simplificação do funcionamento de outros componentes do SGBDR visando a facilitação da construção do ambiente de simulação. Há a possibilidade de experimentos com funcionalidades diferenciadas e sem a necessidade de integrar componentes adicionais para simular tais funcionalidades. Por último, trabalhar no núcleo de otimização do H2, resulta no uso de abordagens totalmente coerentes e pertinentes com a estratégia de otimização empregada por alguns SGBDRs conhecidos, o que aumenta a possibilidade da execução de tais algoritmos em situações reais.

4.2 Resumo

Neste capítulo foi apresentada uma revisão geral de trabalhos focados no problema de ordenação de junções. Além disso, são discutidos aspectos da revisão que influenciaram a escolha do SGBDR H2 para implementação do algoritmo proposto no Capítulo 5.

5 Otimizador Multiagente Evolucionário

Este capítulo descreve um algoritmo baseado em sistemas multiagente para o problema de ordenação de junções. O algoritmo desenvolvido pode ser classificado como Sistema Multiagente Evolucionário, pois ele aplica técnicas inspiradas em modelos evolucionários, tal como os algoritmos genéticos. Além disso, o ambiente é composto por um conjunto de agentes, onde cada um trabalha para encontrar a melhor ordem de junção entre as tabelas presentes na consulta a ser otimizada. Tal metodologia estende o comportamento de algoritmos genéticos e multiagente convencionais, dado que os agentes podem agir de forma pró-ativa e reativa. Eles podem tomar decisões aplicando mecanismos diferenciados de interação com outros agentes e explorar o espaço de soluções de forma inteligente por meio do uso de operadores empregados nos algoritmos genéticos. Em 2010 DREZEWSKI; OBROCKI; SIWIK; são citados dois mecanismos diferentes de evolução, sendo eles: Seleção Mútua e Hospedeiro-Parasita. Nesse trabalho foi aplicado o método de seleção mútua como mecanismo de evolução, isto é, cada agente seleciona individualmente outro agente para executar as operações disponíveis no ambiente multiagente. O algoritmo desenvolvido é descrito em detalhes na Seção 5.1.

5.1 Algoritmo Desenvolvido

O algoritmo desenvolvido (GONÇALVES; GUIMARÃES; SOUZA, 2013; GONÇALVES; GUIMARÃES; SOUZA, 2014) é composto por um ambiente, o qual possui uma série de agentes que trabalham na otimização de uma consulta. Nesse ambiente, cada agente representa uma solução. O sistema desenvolvido pode ser denotado pela seguinte expressão:

$$EMA = \langle E, \Gamma, \Omega \rangle \tag{5.1}$$

em que E é o ambiente do sistema, Γ o conjunto de recursos do sistema e Ω os tipos de informações disponíveis para os agentes no sistema. O ambiente é do tipo acessível e neste caso, os agentes podem obter informações precisas, atualizadas e completas sobre o ambiente. Tal ambiente também pode ser considerado não-determinístico e dinâmico, pois não há certeza sobre estado resultante devido a uma ação executada

e os agentes podem modificar o ambiente durante sua execução. Ainda em relação ao ambiente E, o mesmo pode ser expresso da seguinte forma:

$$E = \langle T^E, \Gamma^E, \Omega^E \rangle \tag{5.2}$$

Na expressão anterior, T^E representa a topografia do ambiente, Γ^E os recursos do ambiente e Ω^E os tipos de informação existentes no ambiente. Foi adotado apenas um tipo de recurso e um tipo de informação no ambiente. O tipo de recurso definido foi a vida (*life*) destinada a cada agente. Γ^E foi definido como as informações completas sobre cada agente e a melhor solução corrente. A topografia é dada pela seguinte notação:

$$T^E = \langle A, l \rangle \tag{5.3}$$

em que A representa o conjunto de agentes presentes no ambiente e l uma função que possibilita localizar um agente específico. Cada agente a pode ser descrito como segue:

$$a = \langle sol^a, Z^a, \Gamma^a, PR^a \rangle \tag{5.4}$$

sendo sol^a a solução do agente, a qual é formada por um vetor de *TableFilter* (maiores detalhes na Seção 3.1), com cada elemento do vetor representando uma relação da consulta. Este vetor define a ordem de junção entre as relações. O conjunto de ações do agente é dado por Z^a. Γ^a define a quantidade de vida e PR^a o perfil do agente.

O conjunto de ações possíveis foi definido como segue:

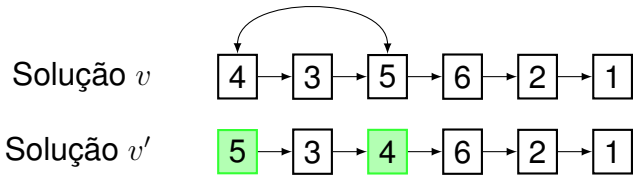
- ***getLife* (*gl*)** - Utilizado para obter pontos de vida de outro agente e somá-los à própria vida;
- ***giveLife* (*vl*)** - Transfere para outro agente parte dos pontos da própria vida;
- ***lookWorse* (*lw*)** - Procura entre todos os agentes aquele com a pior aptidão avaliada;
- ***lookPartner* (*lp*)** - Procura por um parceiro no conjunto de agentes no ambiente;
- ***crossover* (*cr*)** - Aplica o operador de cruzamento para gerar um descendente;
- ***mutation* (*mt*)** - Aplica o operador de mutação;

- ***becomeBestSol* (*bb*)** - Atualiza a própria solução com a melhor solução corrente;
- ***updateBestSol* (*ub*)** - Tenta atualizar a melhor solução corrente;
- ***randomDescent* (*rd*)** - Aplica o método de descida randômica para refinamento da solução;
- ***parallelCompleteDescent* (*pd*)** - Aplica uma abordagem paralela do método de descida completa para refinamento da solução;
- ***semiGreedyBuild* (*gb*)** - Gera uma solução por meio de uma heurística construtiva semi-gulosa;
- ***processRequests* (*pr*)** - Avalia e processa mensagens pendentes;
- ***tryChangeProfile* (*cp*)** - Tenta mudar de perfil corrente do agente;
- ***stop* (*st*)** - Para a sua execução momentaneamente, mas não interrompe totalmente como na ação *die*;
- ***die* (di)** - Ação de morrer; neste caso, os pontos de vida do agente terminaram.

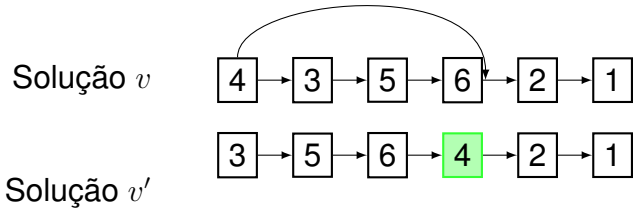
Cada agente foi definido como sendo uma *thread* de execução. Estes agentes podem ser classificados como híbridos, mesclando entre reativos sem estado interno e deliberativos com metas de atualizar a melhor solução corrente e não morrer. As seguintes características podem ser citadas:

- **Reatividade** - Reagem a estímulos do ambiente;
- **Proatividade** - São capazes de tomar decisões;
- **Habilidade Social** - Tem a capacidade de se comunicar com outros agentes;
- **Veracidade** - Se comunicam de forma verdadeira;
- **Benevolência** - Sempre tentam fazer o que foi solicitado;

Com respeito às ações *mutation* e *randomDescent*, foram utilizados movimentos de troca e realocação para explorar o espaço de soluções em tais ações. Na mutação, fixou-se por padrão um total de 5 movimentos em cada execução da ação com relação a algumas das vizinhanças. Já na operação *randomDescent*, adotou-se como critério de parada para o método de refinamento Descida Randômica, um número máximo de iterações sem melhora dado por $RD_{mov} = RDE \times IL$, sendo RDE um



(a) Movimento de Troca - V^T



(b) Movimento de Realocação - V^R

Figura 25 – Tipos de Movimento

coeficiente de esforço e IL a vida inicial do agente (Expressão (5.5)). Os movimentos aplicados na avaliação da vizinhança são ilustrados na Figura 25.

Três operadores de cruzamento foram implementados: *Ordered Crossover* - OX (DAVIS, 1985), *Sequential Constructive Crossover* - SCX (AHMED, 2010) e um operador proposto neste trabalho, denominado de Pandora-Aquiles *Greedy Crossover* - PAGX. Os exemplos de cruzamento apresentados a seguir utilizam como entrada as soluções da Figura 26.

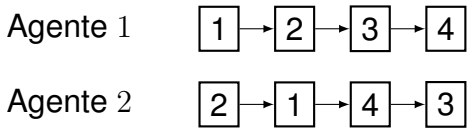


Figura 26 – Agentes Exemplo

No cruzamento OX, um ponto de corte é selecionado aleatoriamente. Este ponto define a parte da solução do Agente 1 que será repassada ao descendente (a solução parcial é repassada na mesma ordem de leitura do Agente 1). O restante da solução é retirado do Agente 2 e adicionado ao descendente na mesma ordem em que os elementos forem lidos, sendo as repetições ignoradas. Considerando os agentes na Figura 26 e um ponto de corte após a segunda relação, o descendente resultante é apresentado na Figura 27.

Na estratégia SCX, o custo das operações de junção entre cada par de relações é gravado em uma matriz de custos. O método inicia adicionando a primeira relação do Agente 1 no descendente. A seguir, o descendente recebe a próxima relação de um dos ascendentes cujo custo de junção seja o menor, ou seja, considerando

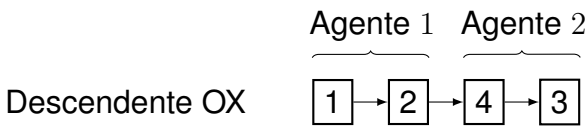
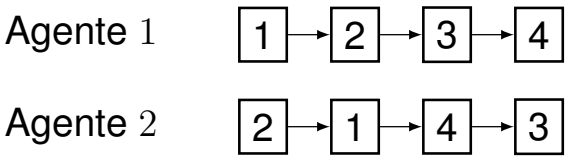


Figura 27 – Resultado Cruzamento OX

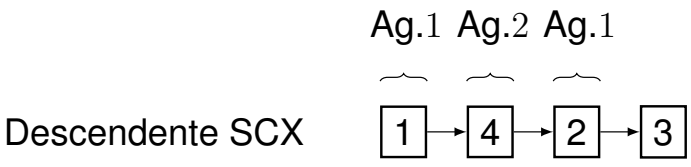
a inserção da relação A no descendente em uma iteração anterior, e os agentes ascendentes 1 e 2 possuam em sua solução uma junção com A dada por $A \bowtie B$ e $A \bowtie C$, respectivamente. A relação escolhida dentre B e C, será aquela com o menor custo de junção. Mais ainda, empates são resolvidos de forma aleatória e nos casos em que A for a última relação em um dos ascendentes, considera-se a junção de A com a primeira relação ainda não inserida no descendente, onde a relação não inserida é selecionada a partir da primeira posição da solução no ascendente. Com base nos agentes da Figura 26 e a matriz de custos seguinte, a solução resultante é dada pela Figura 28.

Matriz de Custos				
	R1	R2	R3	R4
R1	99	10	20	5
R2	10	99	15	35
R3	20	15	99	45
R4	5	35	45	99

(a) Matriz de Custos



(b) Soluções de Entrada



(c) Resultado

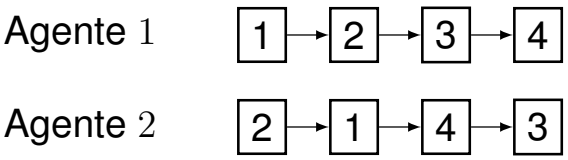
Figura 28 – Resultado Cruzamento SCX

Por fim, no operador PAGX há uma mistura entre aleatoriedade e determinismo. Inicialmente são escolhidos aleatoriamente dois pontos de corte na solução dos ascendentes. Diferentemente do método OX, a solução parcial não é formada necessariamente com uma sequência iniciando da primeira relação até o ponto de corte. Além disso, o descendente recebe a solução parcial de menor custo entre os ascendentes. No caso de empate, a solução parcial do primeiro agente avaliado é selecionada. A seguir, o restante das relações ainda não inseridas são repassadas ao descendente de forma gulosa com base no custo de junção. Para tal, é criada uma lista L com tais

relações. O algoritmo percorre L a cada iteração, e remove da mesma, a relação cujo custo de junção seja o menor. Em caso de empate, o primeiro elemento da lista é removido. A solução final será gerada quando L estiver vazia. Considerando os agentes da Figura 26, os dois pontos de corte englobando a segunda e terceira relações e a matriz de custos apresentada no exemplo do SCX, o descendente resultante é exibido na Figura 29.

Matriz de Custos				
	R1	R2	R3	R4
R1	99	10	20	5
R2	10	99	15	35
R3	20	15	99	45
R4	5	35	45	99

(a) Matriz de Custos



(b) Soluções de Entrada

(c) Resultado

Figura 29 – Resultado Cruzamento PAGX

A ação *semiGreedyBuild* é baseada em uma heurística semi-gulosa (gulosa-randômica), a qual introduz aleatoriedade numa heurística construtiva gulosa. Uma solução é construída inserindo-se as relações passo a passo. Para cada iteração, a lista de relações não inseridas na solução, ou lista de candidatos LC, é ordenada de acordo com um critério de classificação. Neste trabalho foram adotados dois critérios distintos de classificação:

1. **Número de Arestas** - Este critério utiliza o número de arestas de ligação ou conexões como critério de classificação. Dado que o problema em questão pode ser visualizado como um grafo (tal como no TSP (CORMEN et al., 2001)), em que os nós são relações e as arestas de ligação representam predicados de junção entre os nós. Assim, relações com menos conexões são priorizadas, tal estratégia se apoia na ideia de que uma relação com menos arestas de ligação pode gerar planos cartesianos (junção entre duas relações sem predicado de junção), pois tem menos possibilidades de conexão com as relações da solução parcial. Este tipo de estratégia é utilizado no operador de cruzamento ERX (WHITLEY; STARKWEATHER; SHANER, 1990);

2. **Custo de Junção** - Nesse critério é utilizado o custo de inclusão da relação em questão na solução, ou seja, o custo de junção da mesma na solução parcial. Tal estratégia ordena os custos de forma crescente, priorizando relações com menor custo de junção.

Os melhores elementos de LC formam a lista restrita de candidatos LRC, a qual é utilizada para inserção dos elementos na solução. O novo elemento a ser adicionado é selecionado aleatoriamente de LRC. A cada inserção de uma nova relação a lista LRC é atualizada. Quando todas as relações tiverem sido inseridas, a nova solução é retornada. Por fim, foram aplicadas duas estratégias de execução na ação *semiGreedyBuild*, classificadas como:

1. **ERX** - Nesta estratégia é utilizado o critério de classificação baseado no número de arestas e a lista LRC composta por 50% dos elementos melhores classificados em LC;
2. **SCX** - Esta estratégia mescla entre uma abordagem totalmente gulosa e o procedimento descrito anteriormente e utiliza como critério de classificação o custo de junção. Desta maneira, a lista LRC é configurada para conter apenas o melhor elemento de LC durante inclusão de 50% das relações. Para as demais relações, configura-se LRC para conter 50% dos melhores elementos de LC e executa-se tal como descrito anteriormente.

Uma solução refinada pelo método de Descida Completa é dita ser um ótimo local com relação a vizinhança adotada (aleatoriamente escolhida entre V^T ou V^R). Neste trabalho desenvolveu-se uma versão paralela para o método de refinamento. Ressalta-se que tal implementação foi totalmente facilitada pela infra-estrutura já montada para utilização do sistema multiagente. Assim, necessitou-se basicamente incluir uma ação chamada *parallelCompleteDescent* no conjunto de ações dos agentes. Esta ação recebe uma lista contendo orientações a respeito da forma de avaliação do espaço de soluções. Esta lista é compartilhada por todos os agentes; consequentemente, deve possuir mecanismos de sincronização de acesso. Para garantir que a solução processada paralelamente entre os agentes seja um ótimo local, duas restrições foram adotadas, são elas:

1. **Solução de Partida** - A solução inicial ou solução de partida para o refinamento deve ser a mesma para todos os agentes;
2. **Lista de Tarefas** - A lista de tarefas compartilhada entre agentes deve garantir que a parcela do espaço de soluções analisada seja suficiente para afirmar que a solução refinada seja um ótimo local.

A Descida Completa paralela implementada é executada ao final do processo de otimização, como descrito a seguir. Utilizou-se como solução de partida a melhor solução corrente. Sabe-se que o refinamento em questão avalia para cada elemento da solução todos os seus vizinhos, e ao final é retornado o melhor vizinho avaliado. Desta maneira, para refinar uma solução com n relações, é necessário avaliar os vizinhos da primeira à n-ésima relação da solução, o que pode ser naturalmente representado por uma lista LA de n elementos, em que cada elemento desta lista é uma relação da solução. Diante das informações apresentadas, a lista LA foi adotada como a lista de tarefas compartilhadas entre os agentes. Por fim, durante o processo de refinamento, adotou-se a estratégia de excluir da lista compartilhada o item de avaliação selecionado por cada agente, e assim, obrigar os agentes a interromper sua execução quando a lista estiver vazia. É importante ressaltar que os agentes sempre tentam atualizar a melhor solução corrente quando geram uma nova solução ou vizinho; portanto, a melhor solução corrente será o melhor vizinho avaliado por um dos agentes.

A ação *processRequests* processa diferentes tipos de mensagens definidas no ambiente. Por exemplo, se um agente A requisitar pontos de vida de um agente B, a execução da ação *processRequests* pelo agente B implicará na execução da ação *giveLife* em resposta a requisição gerada pela ação *getLife* de A.

Um sistema multiagente possibilita que os agentes executem diferentes objetivos, os quais podem ser: sobreviver, gerar uma solução em conjunto com outro agente, refinar sua própria solução e assim por diante. Para trabalhar esta característica, foram definidos uma série de perfis com propósitos diferenciados, são eles:

- ***RESOURCE*** - O agente tenta aumentar seus pontos de vida por meio da procura de agentes com piores avaliações na aptidão e posterior requisição de pontos de vida dos mesmos;
- ***REPRODUCER*** - O agente busca parceiros para gerar novos descendentes por meio de cruzamento;
- ***MUTANT*** - O agente sofre mutações e tenta explorar outras partes do espaço de soluções;
- ***RANDOM_DESCENT*** - O agente tenta refinar a própria solução aplicando um método de busca local;
- ***SEMI-GREEDY*** - O agente gera novas soluções por meio de uma heurística semi-gulosa;

- ***SLAVE_COMPLETE_DESCENT*** - O agente utiliza o método descida completa para refinar a própria solução. A análise da vizinhança é direcionada a uma parte específica dos vizinhos.

Tendo em consideração os perfis definidos, é importante notar que na tentativa de se evitar uma alta taxa de aleatoriedade e um alto esforço de refinamento, as probabilidades de escolha dos perfis *MUTANT* e *RANDOM_DESCENT* foram configuradas em 10% para ambos os perfis. Por outro lado, buscando uma pesquisa inteligente e efetiva do espaço de soluções, os perfis *REPRODUCTION* e *SEMI-GREEDY* receberam uma probabilidade de escolha igual a 60% e 15%, respectivamente. Finalmente, a fim de impor alguma pressão seletiva para o algoritmo, a probabilidade de seleção do perfil *RESOURCE* foi definida como 5%.

O pseudocódigo do otimizador multiagente evolucionário é apresentado no bloco de código 8. A lógica desenvolvida inicializa o ambiente na linha 2. A seguir, todos os agentes são criados com o perfil *REPRODUCER* (linhas 4 e 6). É válido notar que o primeiro agente é construído com as relações da solução ordenadas de acordo com a leitura realizada durante a fase de *parsing* (linha 4), e os demais são carregados com uma solução gerada por meio da heurística semi-gulosa (linha 6). A inicialização dos agentes e o refinamento final da melhor solução corrente são feitos nas linhas 9, 11 e 20. Uma vez inicializados os agentes, o processo evolucionário também começa, visto que eles podem evoluir por meio da execução de ações, as quais podem ser puramente individuais ou tomadas em conjunto com outros agentes. A cada iteração do agente em um dado perfil, parte de seus pontos de vida são decrementados. Quando um agente não tiver mais pontos de vida ele executa a ação *stop*. Quando todos os agentes pararem, o algoritmo executa e descida completa paralela e por fim todos os agentes morrem (ação *die*) e o algoritmo de fato encerra sua execução.

Algoritmo 8: MAQO

```
   Data: QueryRelations
   Result: sol*
1  begin
2      Environment E ← initializeEnvironment(QueryRelations);
3      //Create the agents with the profile REPRODUCTION;
4      createAgentWithInitialSolution(E, REPRODUCTION);
5      for i=2 to E →MAXAGENTS do
6          createAgentWithSemiGreedySolution(E, REPRODUCTION);
7      end for
8      //Initialize threads;
9      initializeAgents(E);
10     //Wait threads finish;
11     waitAgents(E);
12     //All agents update their solutions;
13     for i=1 to E →MAXAGENTS do
14         E →findAgent(i)→becomeBestSol();
15     end for
16     //Generate the tasks that will be executed by the agents;
17     List TASKS ← generateTasks();
18     //Parallel Best Improvement in the best current solution;
19     for i=1 to E →MAXAGENTS do
20         E →findAgent(i)→parallelCompleteDescent(TASKS);
21     end for
22     return E →bestSolution;
23 end
```

Na Figura 30 é exibido o relacionamento entre o comando *Select* e o otimizador descrito nessa Seção. Pode-se observar a presença da função *calculateMultiAgent()* em *Optimizer*, a qual implementa toda a lógica descrita no Algoritmo 8.

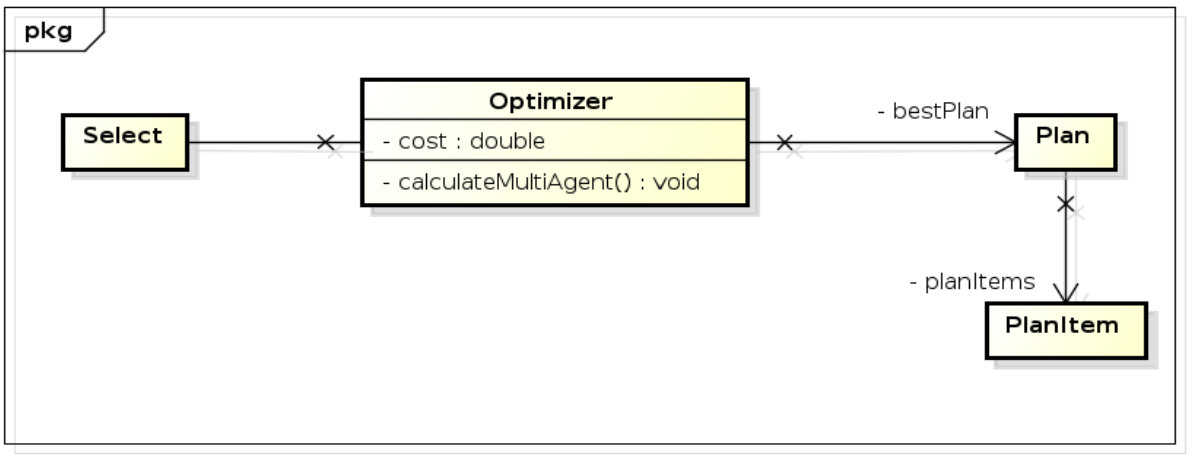


Figura 30 – Relacionamento - Comando *Select* e o Otimizador.

É apresentado, de forma resumida na Figura 31, a estrutura geral do otimizador multiagente evolucionário desenvolvido e o seu relacionamento com a classe

Optimizer. Vale lembrar que tal abordagem é aplicada na resolução de problemas envolvendo 8 ou mais relações e assim como os otimizadores apresentados na Seção 3.2, a classe *Optimizer* é responsável por sua chamada.

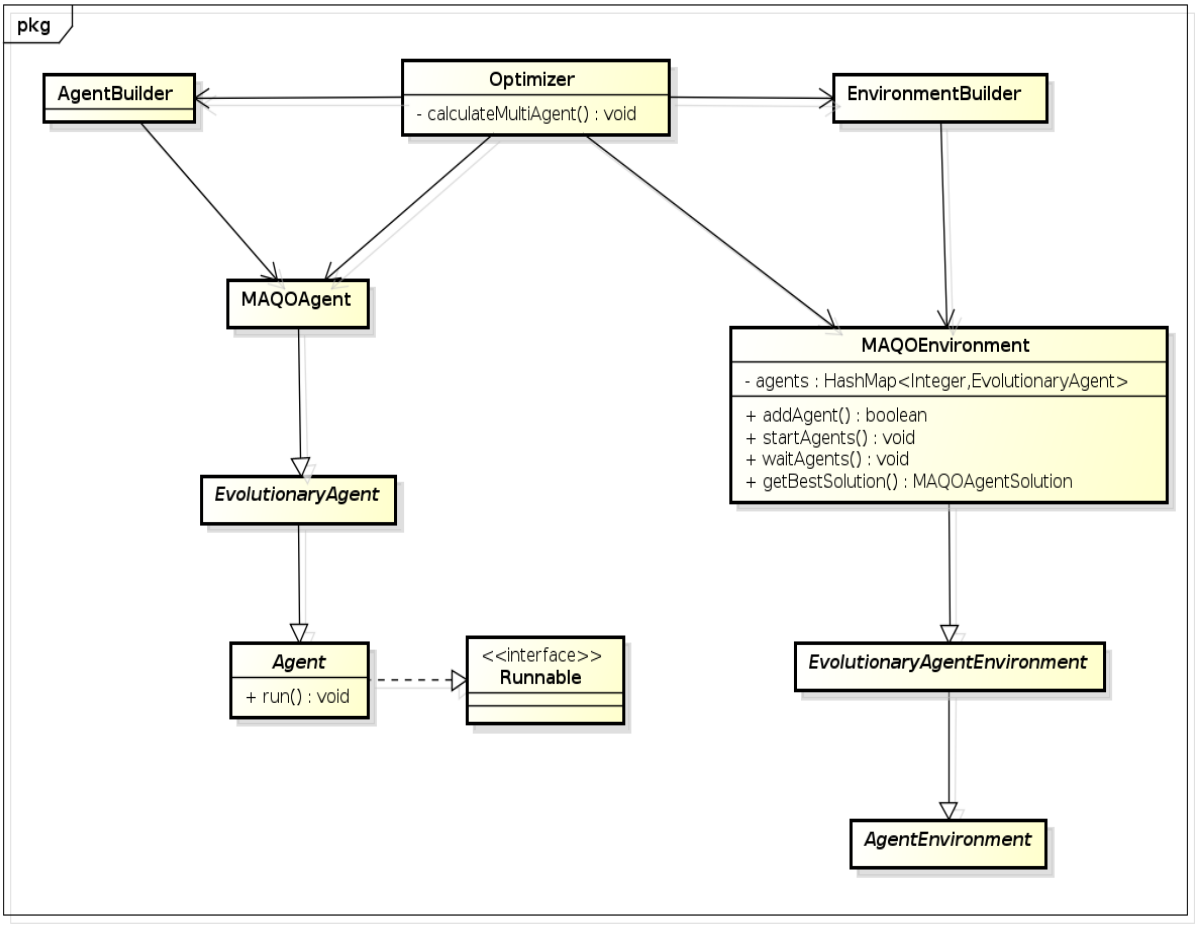


Figura 31 – Estrutura - Otimizador Multiagente Evolucionário.

No que concerne ao sistema multiagente, podem ser destacados dois componentes principais:

- **Ambiente** - Esse componente define o ambiente onde os agentes executarão. A classe *MAQOEnvironment* contem toda a lógica necessária aos agentes no que diz respeito ao ambiente de execução;
- **Agente** - Como não poderia deixar de ser, o componente agente tem um papel fundamental no otimizador. Os agentes são de fato os responsáveis pela pesquisa efetiva do espaço de soluções. As classes *EvolutionaryAgent* e *MAQOAgent* são umas das classes responsáveis pela implementação da abordagem inerente à lógica de otimização discutida nesta seção.

O componente **Agente** é detalhado na Figura 32. A classe *Agent* é a base para todos os agentes do sistema, possuindo informações comuns a todos os agentes. Já a classe *EvolutionaryAgent* inclui os métodos inspirados em modelos evolucionários e contém informações acerca dos perfis possivelmente assumidos pelos agentes. Por fim, a classe principal *MAQOAgent* reúne toda a lógica inerente às ações dos agentes e implementa o restante necessário. Observa-se, ainda, que *MAQOAgent* delega a implementação dos operadores genéticos, heurísticas de refinamento, heurísticas construtivas às classes *MAQOGenetic* e *MAQOHeuristics*.

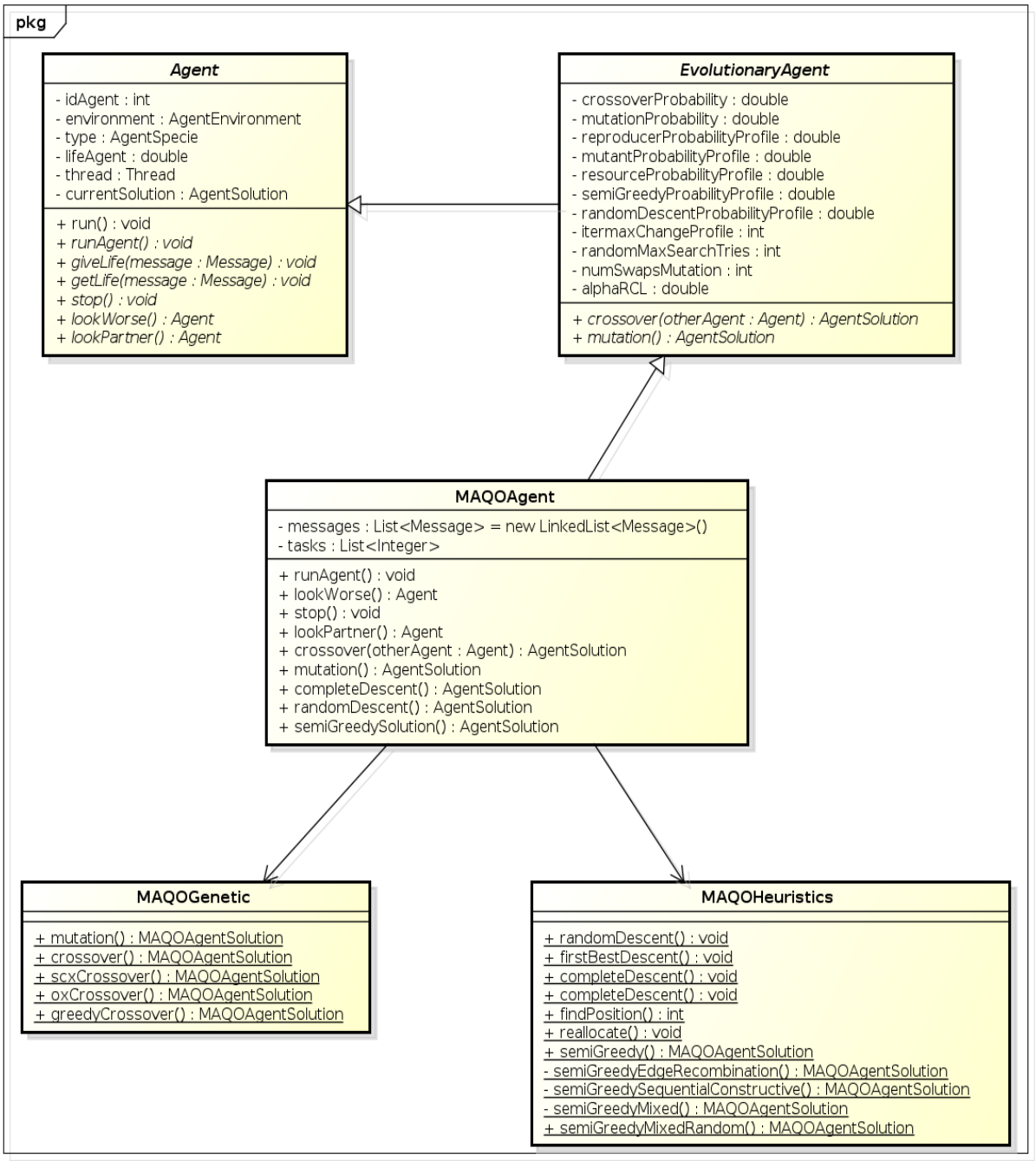


Figura 32 – Componente Agente.

Em relação à inicialização da execução de um agente, destaca-se a existência de um método utilizado para este propósito, denominado *runAgent()*. Como dito anteriormente, os agentes são *threads* de execução, o que pode ser comprovado na Figura 32, por meio da implementação da interface *Runnable* (interface padrão para programação *multithread* em Java). As *threads* são iniciadas através da invocação do método *run()*, que por sua vez chamará a função *runAgent()*.

Uma característica importante e que vale a pena ser discutida, diz repeito à vida inicial destinada aos agentes (atributo *lifeAgent* da classe *Agent*). Neste trabalho utilizou-se uma relação entre a complexidade do problema a ser otimizado e os pontos de vida de cada agente. Assim, a vida de inicial de cada agente é descrita pela expressão a seguir:

$$IL = LFC \times SZP \tag{5.5}$$

em que LFC representa um coeficiente de vida com valor padrão igual a 1 e SZP o número de relações presentes na consulta em questão. Com base na expressão, observa-se que a vida de um agente aumenta na medida em que o problema se torna mais complexo.

O método *runAgent()*, responsável pela inicialização do agente, basicamente instancia um perfil e o executa. Vale observar que cada agente está associado a um perfil corrente, o qual é utilizado para executar um objetivo específico. Durante a execução do otimizador, é permitido aos agentes trocarem esse perfil. Contudo, tais mudanças são autorizadas apenas após um número mínimo de iterações no perfil corrente. Quando os pontos de vida de um agente alcançam um nível crítico (10% da vida inicial), o agente migra automaticamente para o perfil *RESOURCE*. A distribuição das ações apresentadas entre os perfis definidos é descrita a seguir:

- **RESOURCE** - lookWorse, processRequests, getLife, tryChangeProfile, stop and die;
- **REPRODUCTION** - lookPartner, processRequests, crossover, updateBestSol, tryChangeProfile, stop and die;
- **MUTANT** - mutation, processRequests, updateBestSol, tryChangeProfile, stop and die;
- **RANDOM_DESCENT** - becomeBestSol, randomDescent, processRequests, updateBestSol, tryChangeProfile, stop and die;
- **SEMI-GREEDY**- semiGreedyBuild, processRequests, updateBestSol, tryChangeProfile, stop and die;

Os diferentes perfis citados foram estruturados de acordo com a Figura 33.

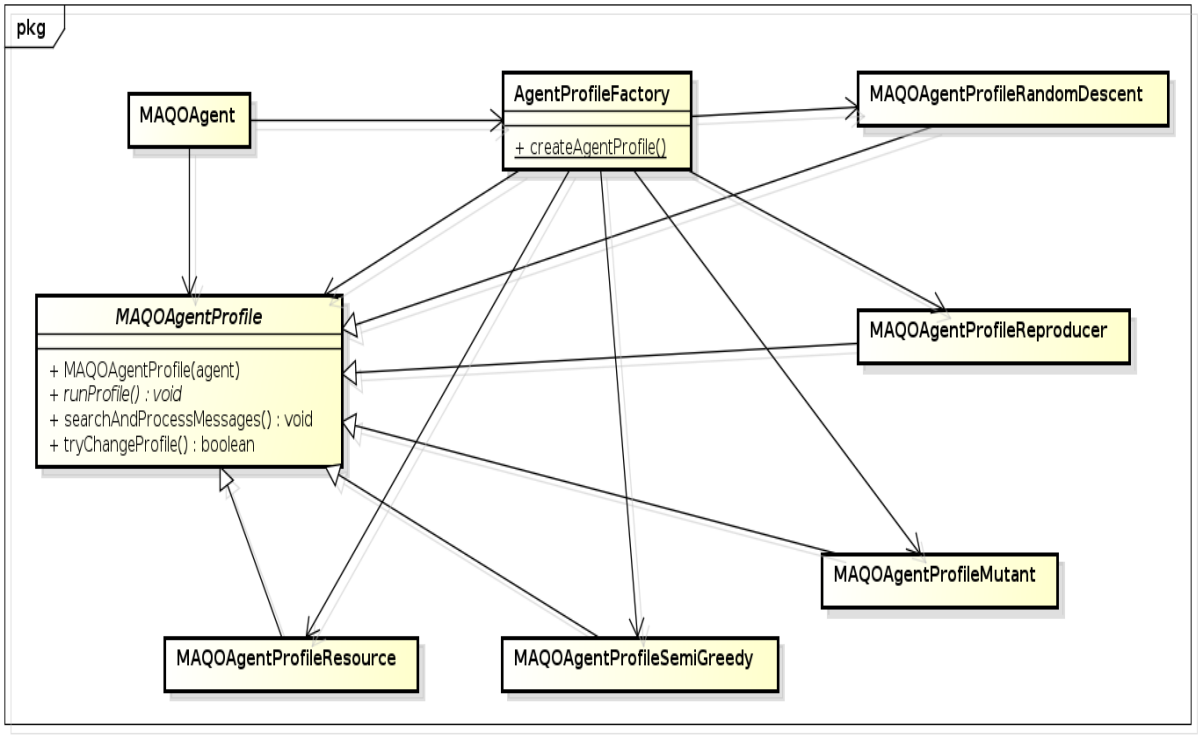


Figura 33 – Estrutura - Perfis dos Agentes.

Observa-se na Figura 33 que *MAQOAgentProfile* representa a classe base necessária na implementação da lógica relacionada a cada perfil definido. Essa classe possui um método abstrato chamado *runProfile()*, o qual deve ser implementado por todas as classes concretas relacionadas. Para uma melhor compreensão da lógica inerente ao método *runProfile()*, é apresentado no Algoritmo 9, o pseudo-código do perfil *RESOURCE*. Este perfil utiliza algumas das ações citadas anteriormente para executar o seu objetivo, que é procurar outros agentes com soluções piores e solicitar pontos de vida dos mesmos. Quanto ao método, pode-se verificar que enquanto o agente possuir pontos de vida (linha 2), ele irá procurar por agentes com soluções piores (linha 4) e obter pontos de vida desses agentes (linha 9). Contudo, quando não houver agentes com soluções piores, o agente diminuirá a própria vida (linha 12). Ao longo de sua execução, ele também tentará mudar de perfil (linha 15). Ainda em relação ao Algoritmo 9, três pontos podem ser destacados:

- **Pontos de vida solicitados a outro agente** - Quando um agente executa ação *getLife()*, é solicitado a outro agente uma quantidade *quantumLife* de pontos de vida. Esta quantidade foi definida como sendo 1% da vida inicial destinada aos agentes;

- **Pontos de vida decrementados** - Em todos os perfis desenvolvidos, existe um bloco específico para decrementar os pontos de vida do agente (Exemplo: linha 12 do Algoritmo 9). Isto é necessário para que o otimizador encerre sua execução em algum momento, visto que ele somente terminará seu processamento quando todos os agentes não possuírem mais pontos de vida. Diante disto, assim como na ação *getLife()*, *quantumLife* pontos de vida são decrementados quando necessário. A quantidade de pontos decrementados por iteração é dado por $DL = DCL \times IL$, onde DCL é um coeficiente relacionado e IL a vida inicial do agente.
- **Tentativa de mudança do perfil** - Tal como o item anterior, a tentativa de mudança do perfil corrente está presente em todos perfis desenvolvidos. Portanto, adotou-se como regra para a mudança, um número mínimo de iterações no perfil corrente dado por 5% dos pontos de vida iniciais de um agente.

Algoritmo 9: RESOURCE

```
   Data:
   Result:
 1 begin
 2     while Agent is alive do
 3         //Find worse agents;
 4         Agent worse ← lookWorse();
 5         //Process the messages;
 6         processMessages();
 7         if exists worse then
 8             //Get life from the agent;
 9             getLife(worse);
10         else
11             //Decrements his own life;
12             life ← life - quantumLife;
13         end if
14         //Tries to change his profile;
15         tryChangeProfile();
16     end while
17     //Executes the stop action;
18     stop();
19 end
```

5.2 Resumo

Foi discutido neste capítulo o otimizador multiagente evolucionário desenvolvido neste trabalho. Descreveu-se como o ambiente de execução foi implementado, bem como os agentes responsáveis pela otimização da consulta. Foram discutidas todas as ações disponíveis para uso dos agentes. Além disso, foram apresentados os

perfis definidos para exploração do espaço de soluções. Por fim, discorreu-se acerca de alguns detalhes inerentes à estrutura geral desenvolvida na implementação do algoritmo proposto.

6 Resultados

Este capítulo tem como foco principal a discussão dos resultados dos experimentos realizados para analisar o algoritmo desenvolvido neste trabalho, descrito no Capítulo 5. Contudo, antes da avaliação propriamente dita, se faz necessária uma revisão de técnicas disponíveis na literatura para avaliação de algoritmos no campo de ordenação de junções em SGBDRs. Além disso, a revisão é pré-requisito para discorrer-se acerca da metologia de avaliação empregada neste trabalho. Assim, na Seção 6.1 é apresentada uma revisão sobre trabalhos importantes no desenvolvimento de metodologias de avaliação do problema otimização em questão. A lógica desenvolvida para a geração dos problemas-teste é discutida na Seção 6.2. Os experimentos computacionais da Seção 6.3 foram distribuídos em 3 subseções. Na Subseção 6.3.1 é realizada a calibração do algoritmo proposto. A seguir, apresenta-se na Subseção 6.3.2, uma avaliação da capacidade do algoritmo desenvolvido em encontrar soluções ótimas. Por fim, experimentos de comparação com outros SGBDRs do mercado são tratados na Subseção 6.3.3. Ao final do capítulo é realizada uma análise geral dos resultados (Seção 6.4).

6.1 Revisão de Metodologias de Avaliação

O problema estudado neste trabalho diz respeito à ordenação de junções em SGBDRs discutida na Subseção 2.2. O processo de avaliação de otimizadores para este tipo de problema envolve a otimização e execução de um roteiro de consultas SQL. A criação do roteiro de consultas pode ser influenciada por uma série de fatores, dentre eles:

1. Quantidade de relações ou junções entre relações a serem ordenadas;
2. Cardinalidade das relações da consulta;
3. Intervalo e distribuição dos valores das colunas das relações das consultas;
4. Seletividade dos predicados de junção;
5. Uso de índices;
6. Métodos de junção disponíveis;
7. Formato das consultas.

8. Presença de pré-otimizadores que buscam otimizar a consulta por meio da reescrita da mesma;

Em relação à lista anterior, ressalta-se que a quantidade de junções a serem ordenadas, a cardinalidade das relações, o intervalo e distribuição dos valores das colunas e o uso de índices têm grande importância na dificuldade do problema de otimização, uma vez que podem influenciar significantemente na taxa de *CPU* e *E/S* requeridos para execução de um planejamento para a consulta. Outro fator com influência na dificuldade do problema é o tamanho de resultados intermediários gerados durante o processamento da cadeia de operações de junção relacionadas. Como dito anteriormente, o número de tuplas resultantes de uma junção pode ser estimado com base na seletividade dos predicados de junção. Em alguns trabalhos (SWAMI; GUPTA, 1988; SWAMI, 1989), a seletividade dos predicados é dada por constantes pré-definidas. Por outro lado, a seletividade pode ser calculada de acordo com o que foi apresentado na Subseção 2.1.2.

Os métodos de junção aumentam a complexidade do espaço de soluções, visto que uma mesma operação de junção pode ser executada de formas diferentes de acordo com os métodos de junções disponíveis no SGBDR. Vale a pena observar que a escolha do método de junção é influenciada pelos itens da lista apresentada discutidos anteriormente.

Vários trabalhos na literatura (SWAMI, 1989; IOANNIDIS; KANG, 1990; STEINBRUNN; MOERKOTTE; KEMPER, 1997; LEE; SHIH; CHEN, 2001; DONG; LIANG, 2007; MULERO, 2007; HAN et al., 2008; LANGE, 2010) utilizam a representação de grafos para definir o formato das consultas. Nesta representação, os vértices definem as relações da consulta e as arestas de ligação representam os predicados de junção. A Figura 34 apresenta um exemplo.

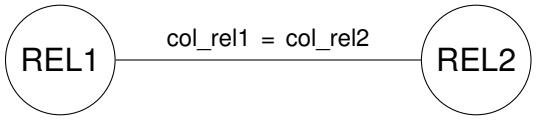
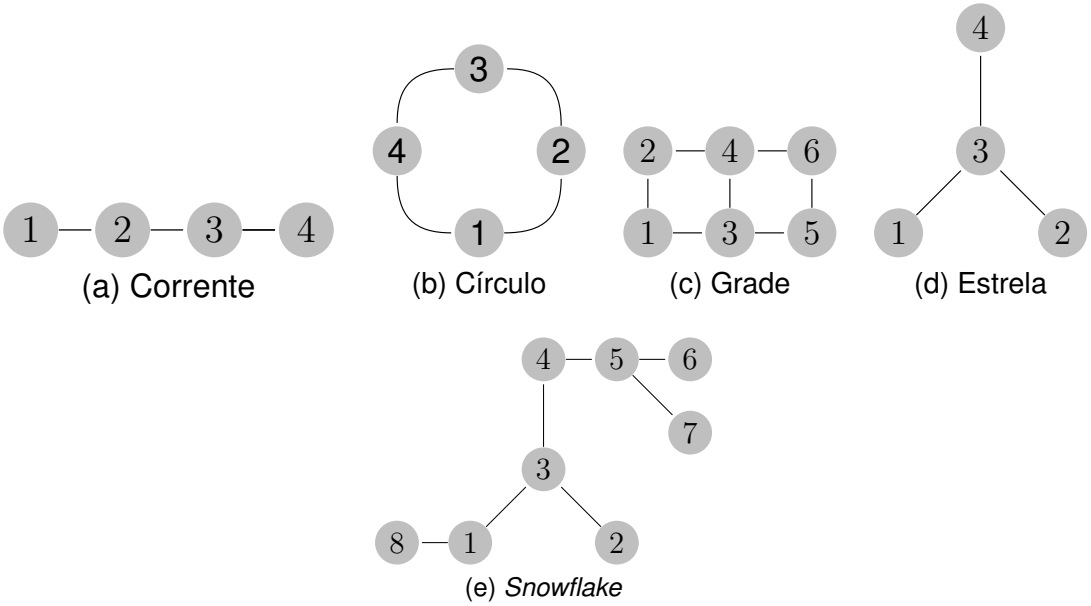


Figura 34 – Exemplo - Formato Grafo.

A utilização de consultas formatadas como grafos de junção leva à herança de propriedades inerentes à teoria de grafos (CORMEN et al., 2001). As seguintes propriedades são citadas por 1984 JARKE; KOCH;: representação visual para entender características estruturais das consultas, análise automática para se descobrir por exemplo ciclos e árvores nas consultas. Os autores em 2007 GUTTOSKI; SUNYE; SILVA; usufruíram da representação em questão para implementar um algoritmo aplicado na formação da árvore geradora mínima ou plano de execução ótimo. Na Figura

35 são apresentados vários tipos de grafos comumente utilizados na literatura para expressar consultas:

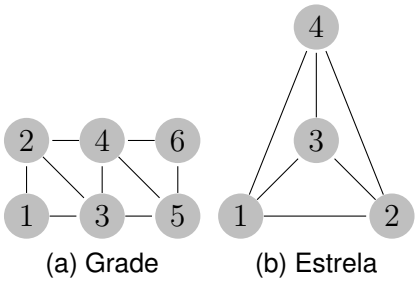


(a) Corrente (b) Círculo (c) Grade (d) Estrela

(e) *Snowflake*

Figura 35 – Tipos de Grafo.

Sistemas de banco de dados relacionais normalmente aplicam antes do planejamento, uma fase de pré-otimização, conhecida como **Reescrita**. Tal abordagem tem como objetivo a transformação da consulta em outra equivalente com alguma otimização. As transformações realizadas na fase de reescrita podem tratar vários aspectos, sendo alguns deles: corrigir alguns possíveis problemas na sintaxe (ex.: transformar o predicado $1 > coluna$ para $coluna < 1$), reduzir a complexidade da consulta (ex.: reduzir as sub-consultas) e aumentar as possibilidades de junção. Dentre as transformações citadas anteriormente, destaca-se a geração predicados de junção implícitos por meio da propriedade de transitividade dos predicados, o que aumenta as possibilidades de junção. Contudo, o aumento das opções de junção implica em uma maior complexidade para o problema. Para exemplificar o processo de geração de predicados implícitos, destaca-se na Figura 36, o resultado da transformação realizada em grafos de junção do tipo grade e estrela. Vale a pena ressaltar que, dentre os tipos de grafo apresentados, o formato estrela pós-transformado oferece uma maior quantidade de possibilidades de junção e, consequentemente, consultas neste formato tornam-se mais complexas, pois possuem um espaço de soluções maior.



(a) Grade (b) Estrela

Figura 36 – Grafos de Junção Transformados.

Tal como em outros campos da otimização, que tratam de problemas práticos, a avaliação do desempenho de SGBDRs pode ser realizada por meio de *benchmarks* padronizados na indústria, que tentam sintetizar experimentos mais realistas. Um exemplo que vale a pena ser ressaltado é o TPC (*Transaction Processing Performance Council*)[1]. De acordo com informações disponíveis no sítio apresentado, a organização TPC é definida como uma corporação sem fins lucrativos fundada para criar *benchmarks* para banco de dados, disseminar o objetivo proposto e disponibilizar de forma confiável resultados acerca de desempenho de SGBDRs do mercado. Os *benchmarks* trabalham a análise do desempenho sob avaliação de pontos tais como: número de usuários simultâneos, consulta de informações e alteração das informações. A avaliação dos resultados pode considerar o custo dos planos, tempo de otimização, tempo total para otimização e execução, etc. Nesta organização há uma coleção de *benchmarks* mantidos com propósitos diferentes, onde busca-se atacar diversas áreas de negócio através de diferentes formas de avaliação. Dentre as opções disponibilizadas, podem ser citadas:

- **TPC-C** - Trabalha a comparação de desempenho de SGBDs no campo de processamento de transações em tempo real (*Online Transaction Processing - OLTP* (ELMASRI; NAVATHE, 2010a));
- **TPC-E** - Simula a carga de trabalho *OLTP* de uma empresa corretora da bolsa de valores;
- **TPC-H** - Trabalha o apoio à tomada de decisões, sendo o mesmo criado para ter uma maior relevância com a indústria;
- **TPC-DS** - Este *benchmark* também é relacionado ao suporte à tomada de decisões e foi desenhado para representar sistemas de apoio modernos. De acordo com 2007 POESS; NAMBIAR; WALRATH;, o *TPC-DS* trata-se da próxima geração de *benchmarks* para apoio à tomada de decisões e é um substituto natural

[1] TPC: http://www.tpc.org

ao *TPC-H*, o que torna seu uso mais interessante em comparação ao *TPC-H*. Maiores informações do *TPC-DS* estão disponíveis no sítio da organização[2] e nos trabalhos de 2006 NAMBIAR; POESS;2007 POESS; NAMBIAR; WALRATH;.

É importante observar que a corporação TPC não disponibiliza o software de *benchmark* em si, e sim uma documentação completa contendo uma série de informações sobre a metodologia de avaliação, e em alguns casos, também libera um programa que gera a base de dados e as consultas relacionadas. Em relação ao conteúdo da documentação, pode-se destacar: descrição e escopo do *benchmark*, esquema de dados e consultas-teste distribuídas de acordo com forma de avaliação proposta. De posse dessas informações, é possível construir um *benchmark* que defina o roteiro de consultas a serem executadas na avaliação. Tal metodologia é de fato utilizada por outros SGBDRs, os seguintes exemplos podem ser citados: *pg_bench* do *PostgreSQL* e módulo de *benchmark* do *H2*.

Os *benchmarks* da organização TPC baseiam-se, em sua grande maioria, em problemas reais da indústria. Deste modo, o roteiro de avaliação é composto por um conjunto de consultas pré-definidas. Entretanto, tais *benchmarks* não representam as únicas alternativas disponíveis para a avaliação de otimizadores de consulta. Uma outra abordagem digna de destaque é a metodologia que utiliza *benchmarks* sintéticos, isto é, um processo de avaliação composto por uma base de dados e problemas-teste construídos de forma aleatória ou também determinística. Tendo em consideração os *benchmarks* sintéticos, destaca-se que tal abordagem não implica necessariamente em um processo de avaliação sem nenhum significado prático, ou seja, tanto a base de dados, quanto os problemas, podem ser gerados em conjunto com parâmetros que simulem um ambiente real. Assim, pode-se verificar um grau de liberdade para construção dos roteiros de testes para a avaliação em diferentes ambientes. Mais ainda, tal método possibilita inclusive que sejam modelados problemas ausentes nos *benchmarks* TPC. A seguir, são destacadas algumas contribuições da literatura com repeito ao uso de *benchmarks* sintéticos.

Em 1988 SWAMI; GUPTA; empregou-se a metodologia aleatória para a construção de consultas envolvendo de 10 a 100 junções, onde N junções implica em consultas com $N + 1$ relações. O processo de avaliação baseou-se nas seguintes informações:

- **Distribuição da Cardinalidade das Relações em Número de Tuplas** - $[10,\ 100)$ - 20%, $[100,\ 1000)$ - 64%, $[1000,\ 10000)$ - 16%;

[2] *TPC-DS*: http://www.tpc.org/tpcds/

- **Seletividade dos Predicados de Junção** - O número de condições de junção por relação foi definido entre 0 a 2 predicados. Além disso, a seletividade dos predicados foi escolhida aleatoriamente dentre os seguintes valores: 0.0001, 0.01, 0.1, 0.2, 0.34, 0.34, 0.34, 0.34, 0.5, 0.5, 0.5, 0.67, 0.8, 1.0.
- **Valores Distintos das Colunas** - $[0,\ 0.2)$ - 75%, $[0.2,\ 1)$ - 5%, 1 - 20%;
- **Índices** - 25% das colunas dos predicados foram configuradas para ter índice.

O processo de construção inicia-se com a seleção aleatória de N relações dentre as disponíveis. Definidas as relações, prossegue-se com a geração de um grafo conectado, ou seja, cria-se predicados de junção entre as relações de forma a se obter uma solução sem planos cartesianos. A seguir, N predicados de junção adicionais são criados. Ressalta-se que formação dos predicados também utilizou uma escolha aleatória das colunas para compor a condição. Através da seleção de diferentes sementes de aleatorização, foram construídas 50 consultas para cada problema envolvendo um número de junções $N = 10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90,\ 100$, totalizando 500 consultas distintas. Quanto à qualidade das soluções, os autores utilizaram uma escala de custo para avaliar este ponto. Para tal, dividiu-se os custos das soluções de cada algoritmo pela melhor solução encontrada para problema em questão. Ressalta-se ainda, que o custo relativo a uma solução foi calculado com base em operações na memória principal (SWAMI, 1989). Por fim, na fase experimental foi desenvolvido um planejamento fatorial de efeitos fixos 2^N com 4 replicações completas. Os fatores estudados foram: o algoritmo e o tempo máximo para otimização. A análise dos resultados foi baseada na variância (ANOVA (MONTGOMERY, 2008)).

Em um outro trabalho (SWAMI, 1989), foi utilizada novamente a técnica de construção aleatória de consultas e da base de dados. Este trabalho se espelha nas ideias discutidas anteriormente e apresentadas por 1988 SWAMI; GUPTA;. As informações padronizadas acerca da cardinalidade das relações, seletividade dos predicados de junção e valores distintos são as mesmas de 1988 SWAMI; GUPTA;. Entretanto, existem algumas diferenças entre os trabalhos no que diz respeito ao processo de geração inicial das consultas, algoritmos avaliados, variação das distribuições dos valores padronizados em diferentes grupos de avaliação e o modelo de custo para avaliar a qualidade das soluções. O processo para geração inicial das consultas inicia-se com a seleção aleatória de N relações e posterior construção do grafo conectado. Após isto, é utilizado um conjunto S de relações inicializado com uma única relação dentre as N selecionadas. Desta maneira, as demais N_i relações são percorridas em ordem para possivelmente formarem novos predicados de junção. A formação dos predicados se dá entre cada relação N_i e outra selecionada aleatoriamente de S. Contudo, o predicado só é aceito se um dado número gerado aleatoriamente for me-

nor que a propriedade de corte JC, setada inicialmente com 0.01. As relações N_i são adicionadas a S na medida em que são lidas na construção dos predicados. Um outro ponto que merece destaque é a divisão dos experimentos em diferentes grupos de consultas-teste, 3 grupos para ser mais exato. Tal divisão se baseia na alteração dos valores padronizados e na forma de geração das consultas apresentados na Tabela 6:

Grupo	**Cardinalidade das Relações (tuplas)**	**Valores Distintos (%)**	**Formato Consulta**
1	$[10, 10^3)$ - 20%; $[10^3, 10^4)$ - 60%; $[10^4, 10^5)$ - 20%;	$(0, 0.2]$ - 80%; $(0.2, 1)$ - 16%; 1.0 - 4%;	Método padrão explicado
2	$[10, 10^4)$ - distribuição uniforme	$(0, 0.1]$ - 90%; $(0.1, 1)$ - 9%; 1.0 - 1%;	Direcionado ao grafo tipo estrela
3	$[10, 10^5)$ - distribuição uniforme	$(0, 0.1]$ - 80%; $(0.1, 1)$ - 16%; 1.0 - 4%;	Direcionado ao grafo tipo corrente

Tabela 6 – Grupos de Consultas-teste

Os resultados foram analisados em duas etapas. Na primeira, os testes utilizaram o mesmo tipo de planejamento e modelo de custo apresentado em 1988 SWAMI; GUPTA;. Já a segunda forma de avaliação, teve modificada a forma de se quantificar o custo de uma solução, a qual baseou-se em um modelo focado em operações na memória secundária similar ao proposto por 1984 BRATBERGSENGEN;.

A metodologia de avaliação proposta por 1990 IOANNIDIS; KANG; baseou-se também na geração aleatória de consultas. Para tal, foram utilizados 3 catálogos diferentes, cada qual contendo relações com cardinalidade e valores distintos selecionados aleatoriamente de intervalos pré-definidos. A Tabela 7 resume as informações sobre os intervalos:

Catálogo	**Cardinalidade das Relações (tuplas)**	**Valores Distintos (%)**
relcat1	1000	$[90, 100]$
relcat2	$[1000, 100000]$	$[90, 100]$
relcat3	$[1000, 100000]$	$[10, 100]$

Tabela 7 – Catálogos das Consultas-teste

Cada relação foi definida com 4 colunas, sendo uma delas indexada. Os demais atributos foram configurados para terem índices mediante a probabilidade de 50%. A construção das consultas considerou as seguintes dimensões para o número de junções $[5, 10, 20, 40, 60, 80, 100]$, sendo geradas para cada catálogo, 20 consultas com até 40 junções e 5 consultas com mais de 40 junções. A seletividade dos predicados de junção foi definida tal como explicado na Subseção 2.1.2. Os formatos utilizados para guiar a construção das consultas foram: grafo tipo estrela e árvores de junção aleatórias. O modelo de custo adotado considerou em seus cálculos apenas as operações de *E/S*, sendo utilizados os seguintes algoritmos de junção: *Nested-Loop Join* e *Merge-Sort Join*. Novamente foi aplicada uma escala de custo para avaliação dos algoritmos selecionados nos experimentos computacionais, escala esta, igual a proposta em 1988 SWAMI; GUPTA;. Os algoritmos testados foram executados um total de 5 vezes em cada um dos problemas gerados nos diferentes catálogos. A análise dos resultados avaliou os algoritmos com base no tempo de otimização e na qualidade das soluções.

No trabalho de 1996 VANCE; MAIER; é apresentada uma metodologia determinística para geração do roteiro de consultas-teste. O estudo em questão analisa o comportamento de algoritmos exaustivos na ordenação de junções. Portanto, o número máximo relações do problema foi definido em 15. As consultas-teste foram modeladas de acordo com os tipos de grafo a seguir:

- **Corrente** - As consultas do tipo corrente tiveram os predicados de junção definidos de acordo com a seguinte permutação: $R_0 \bowtie R_8 \bowtie R_1 \bowtie R_9 \bowtie R_2 \bowtie R_{10} \bowtie R_3 \bowtie R_{11} \bowtie R_4 \bowtie R_{12} \bowtie R_5 \bowtie R_{13} \bowtie R_6 \bowtie R_{14} \bowtie R_7$;
- **Círculo**$+3$ - Neste tipo de grafo, a permutação apresentada no tipo corrente foi aproveitada e acrescida de mais 4 predicados, dados por: $R_0 \bowtie R_7$, $R_8 \bowtie R_{14}$, $R_1 \bowtie R_6$ e $R_9 \bowtie R_{13}$;
- **Estrela** - Configurou-se neste formato o nó central como sendo a relação R_{14} e predicados de junção dele com as demais relações;
- **Clique** - O formato clique é dado por um grafo completo, ou seja, existem predicados de junção entre todos os pares de relações.

Embora as consultas sejam pré-fixadas, a metodologia se apoia em outros fatores na fase experimental. Tais fatores podem ser trabalhados de forma independente o que leva a vários possíveis cenários de testes. Os seguintes fatores podem ser destacados:

- **Média Geométrica** - Este item é utilizado na definição da cardinalidade das relações. A média geométrica é definida por: $M = (\prod_{i=0}^{14} |R_i|)^{\frac{1}{15}}$. Durante os experimentos computacionais, os autores utilizaram valores para média geométrica variando de 1 a 10^8;

- **Variação** - Este fator diz respeito a variação das cardinalidades das relações. Os autores adotaram valores $V \in [0, 1]$. A variação igual a zero indica uma distribuição uniforme das cardinalidades das relações. No caso geral, a cardinalidade de R_0 é dada por $(M)^{1-V}$ e a das demais relações R_i distribuídas de forma que $\frac{|R_i|}{|R_{i-1}|}$ seja constante, ou seja, uma progressão geométrica das cardinalidades de R_0 a R_{14}.

A seletividade dos predicados entre duas relações R_i e R_j foi definida como $M^{\frac{1}{k}} \times |R_i|^{\frac{-1}{k_i}}$
$\times |R_j|^{\frac{-1}{k_j}}$, sendo M a média geométrica, k o número total de predicados, k_i e k_j o número de predicados envolvendo as relações R_i e R_j, respectivamente. Foram adotados 3 modelos de custo diferentes: um modelo definido no próprio trabalho, um baseado no *merge join* e outro baseado no *nested-loop join*. Foi avaliado o comportamento do algoritmo desenvolvido em relação à métrica tempo computacional gasto na otimização da consulta.

Em 2001 SHAPIRO et al.; foi desenvolvida outra pesquisa focando a avaliação de métodos exaustivos. Os autores seguiram as mesmas ideias propostas em 1996 VANCE; MAIER;. A metodologia avaliou consultas envolvendo de 4 a 13 relações. Assim como em 1996 VANCE; MAIER;, a média geométrica e um fator de variação foram utilizados para calcular a cardinalidade das relações. De acordo com os autores, a média geométrica foi definida de forma fixa como 4096 ou 2^{12}. O fator de variação denominado *LOGRATIO*, foi configurado como $\log_2(\frac{|R_1|}{|R_n|})$ ou log_2 da divisão entre o valor da maior cardinalidade e menor cardinalidade. Novamente, a razão $\frac{|R_i|}{|R_{i-1}|}$ se manteve constante para todas as relações. Neste trabalho os autores avaliaram a capacidade dos otimizadores em eliminar a avaliação desnecessária de soluções, medida pelo número de multi-expressões avaliadas nos experimentos. As consultas foram geradas nos seguintes formatos:

- **Corrente** - Foi assumido para este formato, um arranjo sequencial de junções, começando pela relação de maior cardinalidade (R_1). A seguinte permutação representa a expressão para 5 relações: $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5$. Destaca-se que o número esperado de tuplas resultante de uma junção é dado pela seguinte expressão: $|T_i \bowtie T_{i+1}| = |T_i|$;

- **Estrela** - No formato estrela, foi selecionada a relação de maior cardinalidade

(R_1) para ser o nó central, e formando consequentemente predicados de junção com as demais relações.

A metodologia proposta por 2010 LANGE; tomou como base os trabalhos (VANCE; MAIER, 1996; SHAPIRO et al., 2001). Foram comparados dois algoritmos não-exaustivos em problemas-teste com 10, 20, 50 e 100 relações. Tal como nos trabalhos de 1996 VANCE; MAIER;2001 SHAPIRO et al.;, o autor utilizou a média geométrica e um fator de variação *LOGRATIO* para controlar a cardinalidade das relações. O roteiro de testes foi distribuído em consultas nos formatos corrente, círculo, grade e estrela. Mais ainda, diferente dos trabalhos de 1996 VANCE; MAIER;2001 SHAPIRO et al.;, que utilizaram uma abordagem determinística para a seleção das relações em cada um dos problemas-teste, 2010 LANGE; empregou uma metodologia de seleção aleatória para a composição do arranjo das relações no tipo de grafo relacionado.

É importante ressaltar que o desenvolvimento deste trabalho foca o problema de ordenação de junções com muitas relações, e de acordo com a discussão da Seção 2.3, consultas neste contexto trabalham com no mínimo 10 relações. Com base na revisão bibliográfica realizada, nota-se um consenso entre os autores, onde avalia-se o problema de ordenação de junções envolvendo muitas relações como característico de sistemas de apoio à tomada de decisões e mineração de dados. Um exemplo que resume a importância de otimizadores não-exaustivos na ordenação de muitas junções está disponível no sítio do SGBDR *PostgreSQL*[3]. Pode-se verificar que o desenvolvimento do atual otimizador não-exaustivo do *PostgreSQL* (Algoritmo Genético citado na Seção 4.1) foi motivado para suportar um sistema de apoio à decisão relacionado à manutenção de redes elétricas, pois as consultas relacionadas extrapolavam o limite aceitável pelo otimizador exaustivo (Algoritmo de Programação Dinâmica citado na Seção 4.1). Outro campo de aplicação para o problema em questão que merece destaque, está relacionado a aplicações avançadas como *ERP* (*Enterprise Resources Planning*). Tais sistemas são compostos por milhares de relações e possuem consultas que extrapolam a junção de 10 relações. Conforme apresentado em 2000 KOSSMANN; STOCKER;2007 MULERO;, o *ERP SAP R/3* possui consultas envolvendo a junção de mais de 20 relações. Diante do que foi discutido, pode-se observar que há várias aplicações que se beneficiam do estudo do problema em questão, o que caracteriza as aplicações alvo deste trabalho.

Com relação à metodologia de avaliação utilizada neste trabalho, ressalta-se a ausência de um *benchmark* padronizado pela indústria focado no problema de ordenação de junções com muitas relações. Um exemplo claro disso é o *benchmark TPC-DS*, o qual é indicado para o apoio à tomada de decisões, mas que possui apenas uma consulta envolvendo a junção de 10 ou mais relações. Vale ressaltar que

[3] http://www.postgresql.org/docs/9.3/interactive/geqo-intro.html

o *TPC-DS* é composto por um total de 99 consultas. Portanto, a sua utilização na avaliação de algoritmos não-exaustivos se mostrou inviável, uma vez que não possui problemas-teste suficientes. Tal limitação do *TPC-DS* foi o principal motivador para a criação de um *benchmark* sintético. Mais ainda, embora existam trabalhos como os de 2007 GUTTOSKI; SUNYE; SILVA;2009 CHEN et al.;, que tentaram contornar a limitação do número de relações do esquema padrão *TPC* por meio da inclusão de novas relações e geração de novas consultas com mais junções, o autor entende que tal alteração pode descaracterizar o objetivo primário do *benchmark*. Além disso, tal abordagem pode ser bastante dispendiosa e não garante que a base de dados e as consultas continuam expressando problemas reais que originaram o *benchmark*. Ante os fatos apresentados, optou-se por um processo de avaliação experimental dividido em dois experimentos diferentes:

1. **Comparação com um Algoritmo Exaustivo** - Os benefícios relacionados ao uso de *benchmarks* TPC são inquestionáveis. A execução do algoritmo proposto em situações consideradas mais realistas tem grande valia na avaliação do mesmo. Portanto, a Subseção 6.3.2 propõe um experimento que emprega o *TPC-DS* para validar a capacidade do otimizador proposto em encontrar ótimos globais, quando comparado a uma metodologia exaustiva;

2. **Comparação com outros SGBDRs** - Face as limitações expostas no *TPC-DS*, foi executado na Subseção 6.3.3, um experimento com a comparação de várias metodologias aplicadas à otimização de consultas com muitas junções. O experimento foi realizado com base em um *benchmark* sintético construído de acordo as informações da Seção 6.2.

6.2 Metodologia de Avaliação Proposta

Nesta seção é discutida a lógica implementada para a construção do *benchmark* sintético empregado nos experimentos da Subseção 6.3.3. Vale ressaltar que a abordagem proposta foi integrada ao componente do H2 responsável pela avaliação de desempenho. A Figura 37 descreve de forma geral algumas das classes responsáveis pela avaliação de desempenho do H2.

De acordo com o projeto apresentado na Figura 37, verifica-se a existência da interface *Bench*, que é a base para todos os *benchmarks*. As classes *BenchA*, *BenchB*, *BenchC* e *BenchDS* representam implementações para o *TPC-A*, *TPC-B*, *TPC-C* e *TPC-DS*, respectivamente. Todo o conteúdo discutido nesta seção está presente na classe *BenchLJQO* (*Benchmark for Large Join Query Optimization*). Mais ainda, a classe *BenchLJQO* faz uso constante de *BenchmarkEnvironment*, que é responsável

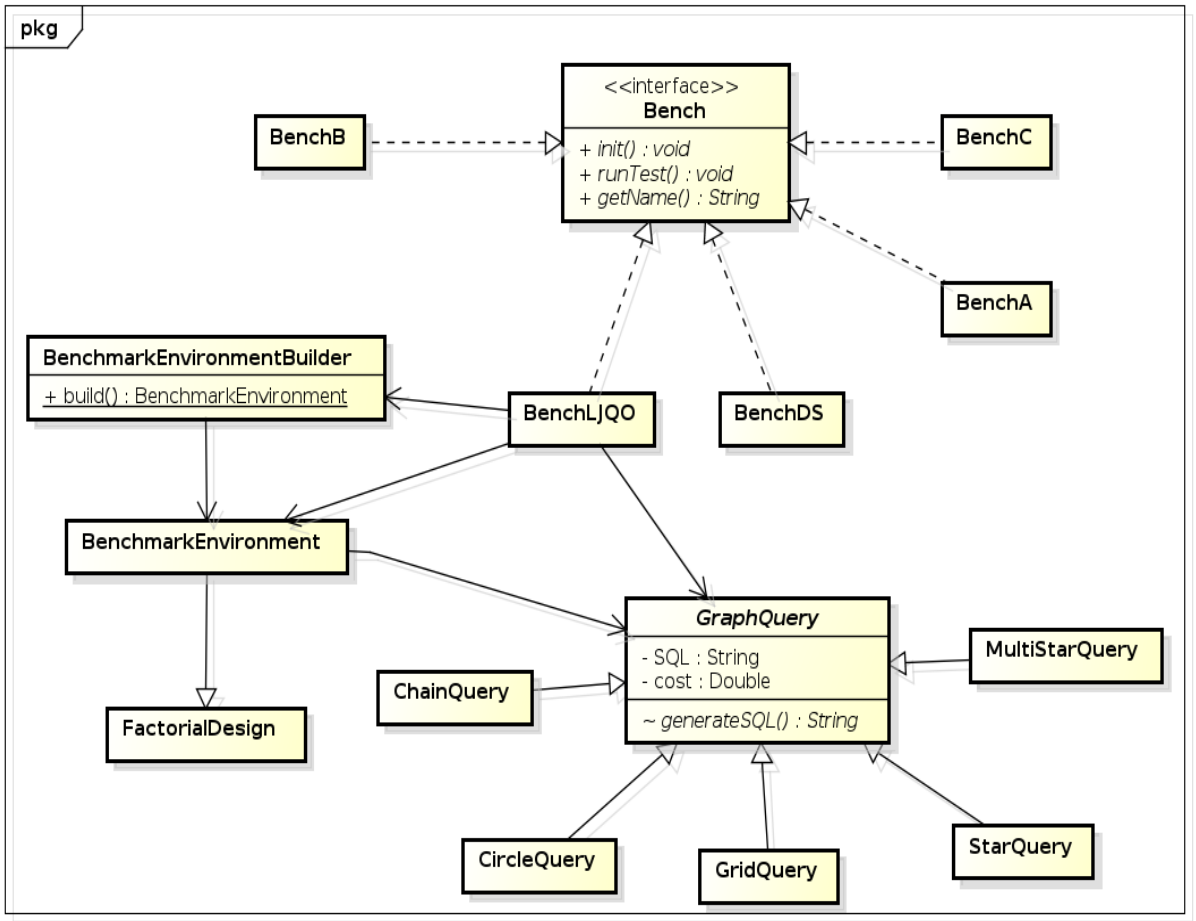


Figura 37 – Modelo de Avaliação de Desempenho do H2.

por gerar a base de dados, as consultas, realizar a integração entre diferentes SGBDRs, salvar resultados, etc. Assim, *BenchmarkEnvironment* produz todas as consultas relacionadas ao roteiro, as quais tem *GraphQuery* como classe base. É digno de nota que a infra-estrutura fornecida pelo H2 facilita a criação de novos *benchmarks*, bem como a sua comparação com outros SGBDRs, o que caracteriza um grande motivador para o uso de tal modelo. Com base na infra-estrutura apresentada, toda a lógica aplicada na construção do roteiro de testes desta seção é discutida a seguir.

O método de avaliação desenvolvido neste trabalho se inspirou em algumas das pesquisas descritas na seção anterior, a qual gera a base de dados e o roteiro de consultas de forma aleatória em conjunto com alguns critérios específicos. Portanto, tomando como base o número mínimo de relações necessárias para execução do otimizador não-exaustivo do *PostgreSQL* e o número máximo de adotado por 2007 MULERO;, definiu-se pela criação de consultas envolvendo 12, 14, 16, 18, 20, 30, 40 e 50 relações. É interessante notar que alguns trabalhos (DONG; LIANG, 2007; LANGE, 2010) exploraram problemas envolvendo a junção de até 100 relações. Contudo, primando por uma avaliação mais realista, decidiu-se por limitar os problemas à junção

de no máximo 50 relações. Além disso, consoante com aspectos práticos levantados por 2000 KOSSMANN; STOCKER;2007 MULERO;, concentrou-se a maior parte da avaliação em consultas com até 20 relações, mas também buscou-se analisar o comportamento dos otimizadores em casos mais extremos, chegando à junção entre 50 relações.

A composição do roteiro de testes em questão é dividida em duas fases; na primeira, constrói-se e povoa-se a base de dados; já na segunda, produz-se as consultas-teste relacionadas. No tocante à construção da base de testes, destaca-se a utilização dos critérios seguintes. Mais ainda, na tentativa de se expressar problemas mais realistas, os valores adotados para os critérios foram baseados em distribuições reais de um banco de dados de produção do centro de tecnologia da informação (TI) de uma universidade brasileira.

- **Distribuição da Cardinalidade das Relações em Número de Tuplas** -$[10,\ 100)$ - 30%, $[100,\ 1000)$ - 16%, $[1000,\ 3000000]$ - 54%;
- **Distribuição de Valores Distintos das Colunas Não-Chave** - $[0,\ 0.2)$ - 30%, $[0.2,\ 1)$ - 16%, 1 - 54%;

No total, 60 relações foram construídas e povoadas de acordo com as distribuições anteriores, sendo a cardinalidade das tabelas e os valores distintos de cada coluna selecionados aleatoriamente dentre intervalos dados. Quanto à carga dos dados, há programas como o *DTM Data Generator*[4], que tem a capacidade de gerar e povoar banco de dados de teste. Contudo, a integração de tais programas com o componente de avaliação de desempenho do H2, aliado às particularidades do *benchmark* proposto, dificulta o uso tais programas.

No que se refere ao esquema das relações criadas, definiu-se para cada uma das tabelas, um total de 3 colunas, sendo a primeira delas reservada à chave-primária da relação (as colunas primárias são compostas em sua totalidade por valores distintos entre si). Vale ressaltar que o número de colunas escolhido pode não refletir situações mais realistas relacionadas ao trabalho que o SGBDR tem para ler e manter as páginas de dados na sua área de cache, quando a mesma é insuficiente para carregar todas as informações (tuplas maiores, contendo mais informações). Assim, uma forma de estudar o efeito do fator memória principal, é destinar uma quantidade de memória menor que o tamanho total dos dados, forçando o SGBDR a utilizar sua política de descarte de páginas. Um outro aspecto importante associado ao efeito do tamanho das tuplas como resultado do número de colunas projetadas das relações, é a escolha de métodos de junção e ordenação, pois cada processo responsável por

[4] http://www.sqledit.com/dg/

uma consulta, normalmente tem sua execução limitada a uma quantidade de memória principal, o que influencia diretamente na escolha de tais métodos. Por outro lado, mesmo com um número pequeno de colunas, a escolha dos métodos de junção e ordenação ainda podem ser afetados, pois por exemplo, problemas envolvendo 50 relações podem gerar tuplas com aproximadamente $6KB$ e um resultado com 1 milhão de tuplas chega a quase $6GB$. Com respeito às colunas não-chave, adotou-se a estratégia de criação de índices e chaves estrangeiras mediante a probabilidade de 90% e 20%, respectivamente.

O domínio escolhido para representar as colunas das relações foi o tipo inteiro. Ressalta-se que valores do tipo inteiro são comuns e facilmente generalizáveis (HOLLOWAY; DEWITT, 2008). Além disso, observou-se nos SGBDRs avaliados na revisão, que o processo para o cálculo da seletividade dos predicados de junção é o mesmo para tipos numéricos e alfanuméricos/textuais. Diante disso, verificou-se que o domínio não interfere no processo de otimização, e que o uso de tipos textuais com a mesma cardinalidade e distribuição de valores distintos, não tem qualquer influência no processo de planejamento. Assim, caso necessário, o domínio pode ser alterado para representar outros problemas sem nenhuma perda para o otimizador. Vale lembrar que o processo de otimização tratado neste trabalho atua na ordenação das junções, delegando ao componente de custo a tarefa de estimativa do custo relacionado ao número de colunas projetadas, seletividade dos predicados de junção, formas de acesso as relações, métodos de junção, etc.

Com respeito ao processo de carga dos valores das colunas, verifica-se que os atributos chaves são compostos por inteiros variando de 0 à sua cardinalidade, sem repetições. Já as demais colunas são compostas por valores distribuídos de acordo com a cardinalidade e a porcentagem de valores distintos escolhidos. Observa-se que os grupos de repetições são gerados com base em intervalos definidos de acordo com a cardinalidade, sendo os demais valores distintos escolhidos fora do intervalo de repetição. A formação dos predicados de junção utilizou apenas operações de igualdade, conector lógico *AND* e a seletividade destes predicados seguiu a definição proposta na Subseção 2.1.2.

Embasado nas informações de distribuição citadas e em um número máximo de 60 relações, a construção do esquema de dados se inicia com a definição do número de tabelas a serem criadas em cada um dos seguintes níveis:

- **Cardinalidade Baixa** - 26% das relações com um número de linhas variando entre 10 e 99;
- **Cardinalidade Média** - 27% das relações com um número de linhas variando entre 100 e 999;

- **Cardinalidade Alta** - 47% das relações com um número de linhas variando entre 1000 e 5000000.

Após definido o número de relações em cada nível de cardinalidade, seleciona-se aleatoriamente do devido intervalo o número de linhas que cada relação terá. A seguir, calcula-se novamente o número de tabelas a serem incluídas em cada uma das opções abaixo:

- **Poucos Valores Distintos** - 30% das relações com 0% a 19% de valores distintos nas colunas;
- **Alguns Valores Distintos** - 16% das relações com 20% a 99% de valores distintos nas colunas;
- **Muitos Valores Distintos** - 54% das relações com 100% de valores distintos nas colunas.

Para cada uma das relações distribuídas nos níveis de valores distintos especificados, seleciona-se novamente de forma aleatória a porcentagem de valores distintos para elas. Após a definição da cardinalidade e valores distintos, inicia-se a fase de configuração dos índices e chaves estrangeiras.

A geração de consultas-teste de acordo com representações de grafos é um processo que está presente em quase todas as abordagens com *benchmarks* sintéticos estudados. É notável a grande utilização do modelo estrela, o qual é justificado pela semelhança com problemas práticos comumente encontrados em sistemas de apoio à decisão. Mais ainda, uma variação do modelo estrela que merece atenção é conhecida como *snowflake*, e que segundo 2010a ELMASRI; NAVATHE;, também pode ser utilizada para representar problemas no contexto de apoio à decisões e extração de conhecimento. Por fim, constatou-se que o modelo corrente é uma representação comum no campo de otimização RDF, sendo utilizados nos trabalhos de 2013 HOGENBOOM; FRASINCAR; KAYMAK;2009 HOGENBOOM et al.; para ordenação de até 20 operações de junção. Portanto, a partir das informações apresentadas, os formatos de grafo escolhidos para guiar a construção das relações foram: corrente, estrela, e *snowflake*. A Figura 35 anterior exibe a disposição dos nós e das arestas para os formatos citados. Como dito anteriormente, a construção do esquema de dados e das consultas-teste são realizadas de forma aleatória com base em diferentes sementes de aleatorização. Consequentemente, é possível gerar esquemas de dados compostos por variadas distribuições de cardinalidades e valores distintos, bem como diferentes consultas.

Parte-se, agora, para o segundo passo da geração do roteiro de testes, que é de fato a criação das consultas-teste. O número de total de consultas construídas está intrinsecamente ligado ao planejamento experimental desenvolvido na Subseção 6.3.3. A construção de cada consulta segue o formato grafo especificado. Assim, o procedimento seleciona inicialmente o número de relações a serem incluídas na consulta, podendo variar de 12 a 50 tabelas. Determinado o número de relações, escolhe-se aleatoriamente dentre as 60 relações disponíveis o número pré-fixado de tabelas. Por fim, os predicados de junção entre as relações são criados por meio da escolha aleatória das colunas das relações envolvidas na junção. Ressalta-se que a escolha das tabelas para formarem os predicados de junção segue os formatos de grafo apontados anteriormente. A seguir, são apresentados exemplos para geração das consultas.

Para as explanações acerca da construção dos problemas, tomou-se a seguinte permutação de relações: $P = R_1 - R_7 - R_2 - R_{10} - R_{30} - R_{50}$. Nesta permutação, R_1 indica a relação de número 1 do esquema de dados, R_7 a relação de número 7 e assim por diante. O formato de grafo do tipo corrente consiste na junção contínua das relações. A consulta SQL exibida no trecho de código 6.1 apresenta um possível exemplo de construção para este formato.

Listing 6.1 – Exemplo - SQL Formato Corrente.

SELECT * **FROM** R_1, R_7, R_2, R_{10}, R_{30}, R_{50} **WHERE** $R_1.col_1 = R_7.col_2$ **AND** $R_7.col_3 = R_2.col_2$ **AND** $R_2.col_1 = R_{10}.col_3$ **AND** $R_{10}.col_1 = R_{30}.col_3$ **AND** $R_{30}.col_2 = R_{50}.col_1$

Consultas no formato estrela contam com um nó ou relação central, a qual possui predicados de junção com cada uma das demais relações. O trecho de código 6.2 exibe uma consulta SQL para elucidar o grafo de junção tipo estrela.

Listing 6.2 – Exemplo - SQL Formato Estrela.

SELECT * **FROM** R_1, R_7, R_2, R_{10}, R_{30}, R_{50} **WHERE** $R_1.col_1 = R_7.col_2$ **AND** $R_1.col_3 = R_2.col_2$ **AND** $R_1.col_1 = R_{10}.col_3$ **AND** $R_1.col_1 = R_{30}.col_3$ **AND** $R_1.col_2 = R_{50}.col_1$

A variação *snowflake* é exemplificada no código SQL 6.3.

Listing 6.3 – Exemplo - SQL Formato *Snowflake*.

SELECT * **FROM** R_1, R_7, R_2, R_{10}, R_{30}, R_{50} **WHERE** $R_1.col_1 = R_7.col_2$ **AND** $R_1.col_3 = R_2.col_2$ **AND** $R_1.col_1 = R_{10}.col_3$ **AND** $R_{10}.col_3 = R_{30}.col_3$ **AND** $R_{30}.col_2 = R_{50}.col_1$

É importante lembrar que a propriedade de transitividade citada no início deste capítulo contribui para a geração de predicados de junção implícitos para os formatos apresentados. Diante disto, pode-se dizer que o consultas no estrela oferecem possibilidades de junção entre todos os pares de relações, o que tende a tornar problemas deste tipo e suas variações mais complexos de se resolver devido ao maior grau combinatório.

6.3 Experimentos Computacionais

Foi empregado nas subseções 6.3.1 e 6.3.3 um planejamento de fatorial de efeitos fixos e análise dos resultados baseada no Teste de Permutações (*Permutations Test for a Analysis of Variance Design* (ANDERSON; BRAAK, 2003)). Já a subseção 6.3.2 aplicou um estudo das médias com base no teste-t pareado. Para a análise estatística, utilizou-se os aplicativos *R* e *Minitab*. Os experimentos foram executados numa máquina *Core i7-2600 CPU 3.40*GHz, com 16GB de *RAM*, Sistema Operacional *Ubuntu* 10.10-x86_64 e Máquina Virtual Java *HotSpot(TM)* 64-*Bit* 1.6. Destinou-se uma quantidade padrão de 1.6GB ao cache de páginas dos SGBDRs e 10MB para a memória reservada aos processos responsáveis pelo processamento das consultas. A memória total das máquinas virtuais Java foi limitada a 9.6GB. O número máximo de tuplas retornadas pelas consultas das subseções 6.3.1 e 6.3.3 foi restrito a no máximo 10 milhões. O tempo máximo de processamento para as consultas em todos os experimentos foi configurado em 60 minutos. Finalmente, antes da execução de todas as consultas, adotou-se a estratégia de descarregar todas as páginas do sistema operacional presentes na memória principal para o disco, foçando sua posterior leitura do disco.

As tabelas ANOVA apresentadas nas subseções 6.3.1 e 6.3.3 são organizadas da seguinte maneira, a coluna DF representa os graus de liberdade (*Degrees of Freedom*), SS representa a soma de quadrados (*Sum of Squares*), MS representa os quadrados médios (Mean Square), *Iter* é o número de iterações realizadas até o cumprimento do critério relacionado e *Pr* refere-se ao *p-value* do teste estatístico relacionado. Já as tabelas apresentadas na Subseção 6.3.2 tem a seguinte estrutura, N representa o tamanho da amostra, *Mean* a média, *StDev* o desvio padrão e *SE Mean* o erro padrão da média (medida de variabilidade). O intervalo de confiança adotado em todos os experimentos foi definido em 95%, implicando em um nível de significância $\alpha = 0.05$. Todas as observações dos experimentos foram totalmente aleatorizadas.

Por fim, vale lembrar que o SGBDR escolhido para implementar o otimizador proposto foi o H2 *Database Engine*, versão $1.3.174$.

6.3.1 Calibração do Algoritmo

A calibração do algoritmo utilizou uma consulta-teste envolvendo 40 relações nos formatos corrente e estrela, respectivamente. Para a configuração, realizou-se um planejamento fatorial de efeitos fixos. Os fatores analisados, ou seja, os parâmetros configurados no algoritmo foram: o número máximo de agentes (*THREADS*), tipo de movimento para a mutação (*MUTATION*), método de cruzamento (*CROSSOVER*), tipo de movimento para método de descida randômica (*RD*), método da heurística semi-gulosa (*SEMIGREEDY*), coeficiente *LFC* da vida inicial do agente representado por *LC* (Expressão (5.5)), coeficiente *DCL* para decremento de pontos de vida do agente representado por *LQ* (Item da Seção 5.1 relacionado à quantidade de pontos de vida decrementada do agente por iteração nos perfis) e coeficiente *RDE* de esforço do método de descida randômica representado por *RDP* (Expressão relacionada ao número máximo de iterações da heurística de refinamento apresentada na Seção 5.1). Em relação aos níveis de avaliação selecionados para o fator *THREADS*, cabe ressaltar que optou-se para o primeiro nível, uma quantidade mínima de agentes igual ao número de núcleos de processamento disponíveis na máquina de testes, já para o segundo, adotou-se o dobro de agentes do primeiro valor. A Tabela 8 descreve os níveis de cada fator.

Tabela 8 – Fatores e Níveis

Fator	**Nível**
THREADS	8, 16
LC	1, 1.5
LQ	0.03, 0.05, 0.1
RDP	0.3, 0.5
MUTATION	*Troca*, *Realocação*
RD	*Troca*, *Realocação*
CROSSOVER	*PAGX*, *SCX*, *OX*
SEMIGREEDY	*ERX*, *SCX*

Foram consideradas duas replicações completas, totalizando assim 2304 observações. O número de observações é calculado como segue: $N_t = N_c \star N_r \star N_q = 1152 \star 2 \star 1 = 2304$, onde N_c é o número de combinações possíveis, N_r o número de replicações e N_q o número de consultas testadas por combinação. As variáveis de saída coletadas foram o custo de execução dos planos de consulta gerados e o tempo de planejamento dos mesmos (em milissegundos). A formulação da análise dos fatores estudados neste trabalho é apresentada no modelo de efeitos da Equação (6.1).

$$y_{ijklmnop} = \mu + \tau_i + \beta_j + \gamma_k + \lambda_l + \pi_m + \sigma_n + \zeta_o + \eta_p +$$

$$(\tau\beta)_{ij} + (\tau\gamma)_{ik} + (\tau\lambda)_{il} + (\tau\pi)_{im} + (\tau\sigma)_{in} + (\tau\zeta)_{io} +$$

$$+(\tau\eta)_{ip} + \ldots + (\tau\beta\gamma\lambda\pi\sigma\zeta\eta)_{ijklmnop} +$$

$$\epsilon_{ijklmnopq} \begin{cases} i = 1, \ldots, a \\ j = 1, \ldots, b \\ k = 1, \ldots, c \\ l = 1, \ldots, d \\ m = 1, \ldots, e \\ n = 1, \ldots, f \\ o = 1, \ldots, g \\ p = 1, \ldots, h \\ q = 1, \ldots, n \end{cases} \tag{6.1}$$

Na Equação 6.1, μ é a média geral. As variáveis τ_i, β_j, γ_k, λ_l, π_m, σ_n, ζ_o e η_p representam os efeitos principais dos níveis dos fatores *THREADS*, *LC*, *LQ*, *RDP*, *MUTATION*, *RD*, *CROSSOVER* e *SEMIGREEDY*, respectivamente. O restante da equação representa o efeito das interações entre os fatores estudados. As hipóteses de teste foram definidas como sendo:

1. $\mathbf{H_0}$: $\tau_1 = \tau_2 = \ldots = \tau_a = 0$ (sem efeito no fator principal *THREADS*);
   $\mathbf{H_1}$: Pelo menos um $\tau_i \neq 0$.

2. $\mathbf{H_0}$: $\beta_1 = \beta_2 = \ldots = \beta_b = 0$ (sem efeito no fator principal *LC*);
   $\mathbf{H_1}$: Pelo menos um $\beta_j \neq 0$.

3. $\mathbf{H_0}$: $\gamma_1 = \gamma_2 = \ldots = \gamma_c = 0$ (sem efeito no fator principal *LQ*);
   $\mathbf{H_1}$: Pelo menos algum $\gamma_k \neq 0$.

4. $\mathbf{H_0}$: $\lambda_1 = \lambda_2 = \ldots = \lambda_d = 0$ (sem efeito no fator principal *RDP*);
   $\mathbf{H_1}$: Pelo menos algum $\lambda_l \neq 0$.

5. $\mathbf{H_0}$: $\pi_1 = \pi_2 = \ldots = \pi_e = 0$ (sem efeito no fator principal *MUTATION*);
   $\mathbf{H_1}$: Pelo menos algum $\pi_m \neq 0$.

6. $\mathbf{H_0}$: $\sigma_1 = \sigma_2 = \ldots = \sigma_f = 0$ (sem efeito no fator principal *RD*);
   $\mathbf{H_1}$: Pelo menos algum $\sigma_n \neq 0$.

7. $\mathbf{H_0}$: $\zeta_1 = \zeta_2 = \ldots = \zeta_g = 0$ (sem efeito no fator principal *CROSSOVER*);
   $\mathbf{H_1}$: Pelo menos algum $\zeta_o \neq 0$.

8. $\mathbf{H_0}$: $\eta_1 = \eta_2 = \ldots = \eta_h = 0$ (sem efeito no fator principal *SEMIGREEDY*);
   $\mathbf{H_1}$: Pelo menos algum $\eta_p \neq 0$.

9. $\mathbf{H_0}$: $(\tau\beta)_{11} = (\tau\beta)_{12} = \ldots = (\tau\beta)_{ab} = 0$ (sem efeito na interação dos fatores);
   $\mathbf{H_1}$: Pelo menos algum $(\tau\beta)_{ij} \neq 0$.

10. $\mathbf{H_0}$: $(\tau\gamma)_{11} = (\tau\gamma)_{12} = \ldots = (\tau\gamma)_{ac} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\tau\gamma)_{ik} \neq 0$.

11. $\mathbf{H_0}$: $(\tau\lambda)_{11} = (\tau\lambda)_{12} = \ldots = (\tau\lambda)_{ad} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\tau\lambda)_{il} \neq 0$.

12. $\mathbf{H_0}$: $(\tau\pi)_{11} = (\tau\pi)_{12} = \ldots = (\tau\pi)_{ae} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\tau\pi)_{im} \neq 0$.

13. $\mathbf{H_0}$: $(\tau\sigma)_{11} = (\tau\sigma)_{12} = \ldots = (\tau\sigma)_{af} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\tau\sigma)_{in} \neq 0$.

14. $\mathbf{H_0}$: $(\tau\zeta)_{11} = (\tau\zeta)_{12} = \ldots = (\tau\zeta)_{ag} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\tau\zeta)_{io} \neq 0$.

15. $\mathbf{H_0}$: $(\tau\eta)_{11} = (\tau\eta)_{12} = \ldots = (\tau\eta)_{ah} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\tau\eta)_{ip} \neq 0$.

16. $\mathbf{H_0}$: $(\beta\gamma)_{11} = (\beta\lambda)_{12} = \ldots = (\beta\gamma)_{bc} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\beta\gamma)_{jk} \neq 0$.

17. $\mathbf{H_0}$: $(\beta\lambda)_{11} = (\beta\lambda)_{12} = \ldots = (\beta\gamma)_{bd} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\beta\lambda)_{jl} \neq 0$.

18. $\mathbf{H_0}$: $(\beta\pi)_{11} = (\beta\pi)_{12} = \ldots = (\beta\pi)_{be} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\beta\pi)_{jm} \neq 0$.

19. $\mathbf{H_0}$: $(\beta\sigma)_{11} = (\beta\sigma)_{12} = \ldots = (\beta\sigma)_{bf} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\beta\sigma)_{jn} \neq 0$.

20. $\mathbf{H_0}$: $(\beta\zeta)_{11} = (\beta\zeta)_{12} = \ldots = (\beta\zeta)_{bg} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\beta\zeta)_{jo} \neq 0$.

21. $\mathbf{H_0}$: $(\beta\eta)_{11} = (\beta\eta)_{12} = \ldots = (\beta\eta)_{bh} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\beta\eta)_{jp} \neq 0$.

22. $\mathbf{H_0}$: $(\gamma\lambda)_{11} = (\gamma\lambda)_{12} = \ldots = (\gamma\lambda)_{cd} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\gamma\lambda)_{kl} \neq 0$.

23. $\mathbf{H_0}$: $(\gamma\pi)_{11} = (\gamma\pi)_{12} = \ldots = (\gamma\pi)_{ce} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\gamma\pi)_{km} \neq 0$.

24. $\mathbf{H_0}$: $(\gamma\sigma)_{11} = (\gamma\sigma)_{12} = \ldots = (\gamma\sigma)_{cf} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\gamma\sigma)_{kn} \neq 0$.

25. $\mathbf{H_0}$: $(\gamma\zeta)_{11} = (\gamma\zeta)_{12} = \ldots = (\gamma\zeta)_{cg} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\gamma\zeta)_{ko} \neq 0$.

26. $\mathbf{H_0}$: $(\gamma\eta)_{11} = (\gamma\eta)_{12} = \ldots = (\gamma\eta)_{ch} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\gamma\eta)_{kp} \neq 0$.

27. $\mathbf{H_0}$: $(\lambda\pi)_{11} = (\lambda\pi)_{12} = \ldots = (\lambda\pi)_{de} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\lambda\pi)_{lm} \neq 0$.

28. $\mathbf{H_0}$: $(\lambda\sigma)_{11} = (\lambda\sigma)_{12} = \ldots = (\lambda\sigma)_{df} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\lambda\sigma)_{ln} \neq 0$.

29. $\mathbf{H_0}$: $(\lambda\zeta)_{11} = (\lambda\zeta)_{12} = \ldots = (\lambda\zeta)_{dg} = 0$ (sem efeito na interação dos fatores);
$\mathbf{H_1}$: Pelo menos algum $(\lambda\zeta)_{lo} \neq 0$.

30. $\mathbf{H_0}$: $(\lambda\eta)_{11} = (\lambda\eta)_{12} = \ldots = (\lambda\eta)_{dh} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\lambda\eta)_{lp} \neq 0$.

31. $\mathbf{H_0}$: $(\pi\sigma)_{11} = (\pi\sigma)_{12} = \ldots = (\pi\sigma)_{ef} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\pi\sigma)_{mn} \neq 0$.

32. $\mathbf{H_0}$: $(\pi\zeta)_{11} = (\pi\zeta)_{12} = \ldots = (\pi\zeta)_{eg} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\pi\zeta)_{mo} \neq 0$.

33. $\mathbf{H_0}$: $(\pi\eta)_{11} = (\pi\eta)_{12} = \ldots = (\pi\eta)_{eh} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\pi\eta)_{mp} \neq 0$.

34. $\mathbf{H_0}$: $(\sigma\zeta)_{11} = (\sigma\zeta)_{12} = \ldots = (\sigma\zeta)_{fg} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\sigma\zeta)_{no} \neq 0$.

35. $\mathbf{H_0}$: $(\sigma\eta)_{11} = (\sigma\eta)_{12} = \ldots = (\sigma\eta)_{fh} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\sigma\eta)_{np} \neq 0$.

36. $\mathbf{H_0}$: $(\zeta\eta)_{11} = (\zeta\eta)_{12} = \ldots = (\zeta\eta)_{gh} = 0$ (sem efeito na interação dos fatores);
    $\mathbf{H_1}$: Pelo menos algum $(\zeta\eta)_{op} \neq 0$.

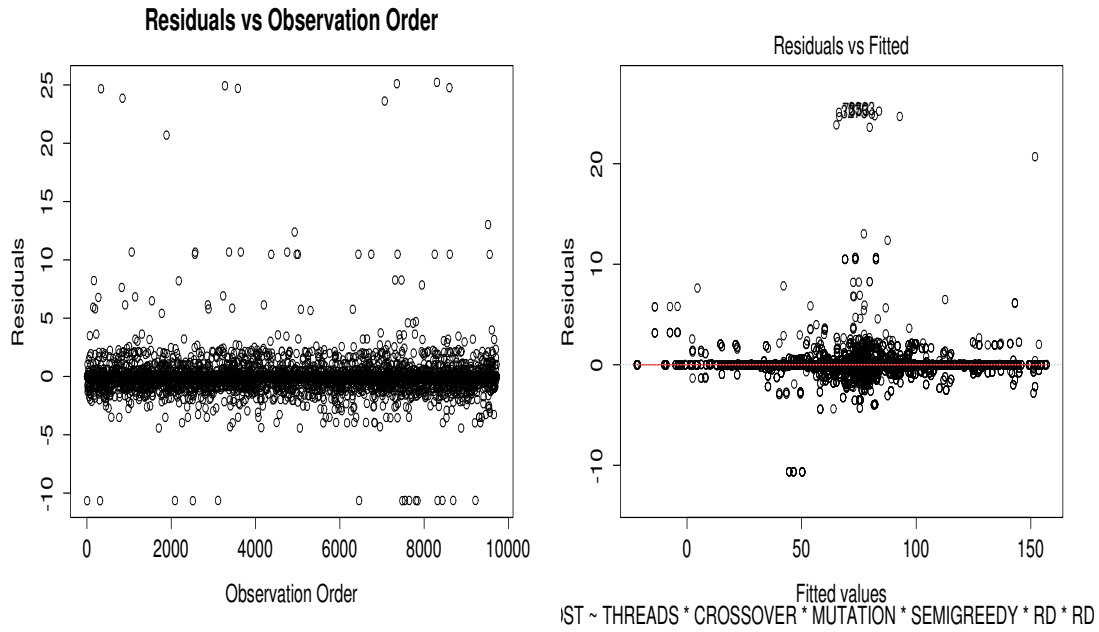
É possível observar nas hipóteses apresentadas que foram estudadas apenas interações entre dois fatores. O planejamento baseou-se no pressuposto de que interações de maior grau tendem a ser menos significantes. Assim, focou-se apenas nas interações de 2º grau. A Tabela 9 apresenta um resumo das informações em relação ao custo dos planos de execução das soluções em escala logarítmica. Ressalta-se que interações de maior grau foram omitidas das tabelas seguintes. De acordo com a tabela 9, é possível verificar que dentre todos os fatores principais, apenas *MUTATION* e *RDP* não apresentaram significância. Contudo, algumas interações com *MUTATION* e *RDP* foram significativas. Por consequência dos resultados da tabela apresentada, vários fatores principais e suas interações explicam parte da variabilidade dos resultados. Diante disto, pode-se rejeitar as hipóteses nulas referentes a estes fatores a um grau de confiança de 95$\%$.

Tabela 9 – ANOVA - Custo dos Planos

Source	**DF**	**SS**	**MS**	**Iter**	**Pr**
THREADS	1	48.9	48.89	5000	< 2e-16 ***
CROSSOVER	2	175.9	87.95	5000	< 2e-16 ***
THREADS:CROSSOVER	2	9.0	4.52	5000	0.15900
MUTATION	1	2.5	2.45	1109	0.08296 .
THREADS:MUTATION	1	3.9	3.94	3639	0.02693 *
CROSSOVER:MUTATION	2	1.7	0.85	51	1.00000
THREADS:CROSSOVER:MUTATION	2	1.9	0.93	907	0.14443
SEMIGREEDY	1	171.6	171.63	5000	< 2e-16 ***
THREADS:SEMIGREEDY	1	5.7	5.71	51	1.00000
CROSSOVER:SEMIGREEDY	2	173.4	86.71	5000	< 2e-16 ***
MUTATION:SEMIGREEDY	1	0.7	0.72	117	0.46154
RD	1	965.6	965.57	5000	< 2e-16 ***
THREADS:RD	1	1.3	1.33	343	0.22741
CROSSOVER:RD	2	169.7	84.87	5000	< 2e-16 ***
MUTATION:RD	1	0.4	0.41	55	0.65455
SEMIGREEDY:RD	1	69.1	69.11	5000	< 2e-16 ***
RDP	1	2.8	2.75	341	0.22874
THREADS:RDP	1	0.4	0.44	112	0.47321
CROSSOVER:RDP	2	3.7	1.83	2382	0.04030 *
MUTATION:RDP	1	0.7	0.66	69	0.59420
SEMIGREEDY:RDP	1	2.1	2.12	88	0.53409
RD:RDP	1	1.6	1.58	750	0.11867
LC	1	249.9	249.87	5000	< 2e-16 ***
THREADS:LC	1	12.1	12.09	1700	0.05588 .
CROSSOVER:LC	2	50.4	25.20	5000	< 2e-16 ***
MUTATION:LC	1	2.2	2.21	579	0.14853
SEMIGREEDY:LC	1	95.0	95.05	5000	< 2e-16 ***
RD:LC	1	1.2	1.20	51	0.92157
RDP:LC	1	1.2	1.17	51	1.00000
LQ	1	10.9	10.87	5000	< 2e-16 ***
RD:LQ	1	19.8	19.81	5000	0.00020 ***
RDP:LQ	1	1.1	1.13	51	1.00000
LC:LQ	1	3.1	3.12	508	0.16535
Residuals	1920	7523.6	3.92		

Signif. códigos: 0 '*' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1**

Com o intuito de avaliar a adequação do modelo em relação às premissas relacionadas ao teste de permutações, são apresentados na Figura 38 os gráficos necessários. Nesta Figura, a avaliação da premissa de independência e dada pela Figura 38a e a de homoscedasticidade pela Figura 38b. De acordo com a análise gráfica, a premissa de independência está atendida, pois não há a formação clara de uma estrutura padrão em relação ao espalhamento dos resíduos. Ressalta-se que as observações foram totalmente aleatorizadas, o que tende a evitar estruturas nos resíduos. Em relação a homoscedasticidade, também não há um padrão claro da igualdade de variância em relação aos valores ajustados que indique a violação.



(a) *Independência* (b) *Homoscedasticidade*

Figura 38 – Premissas Modelo de Calibração - Custo dos Planos.

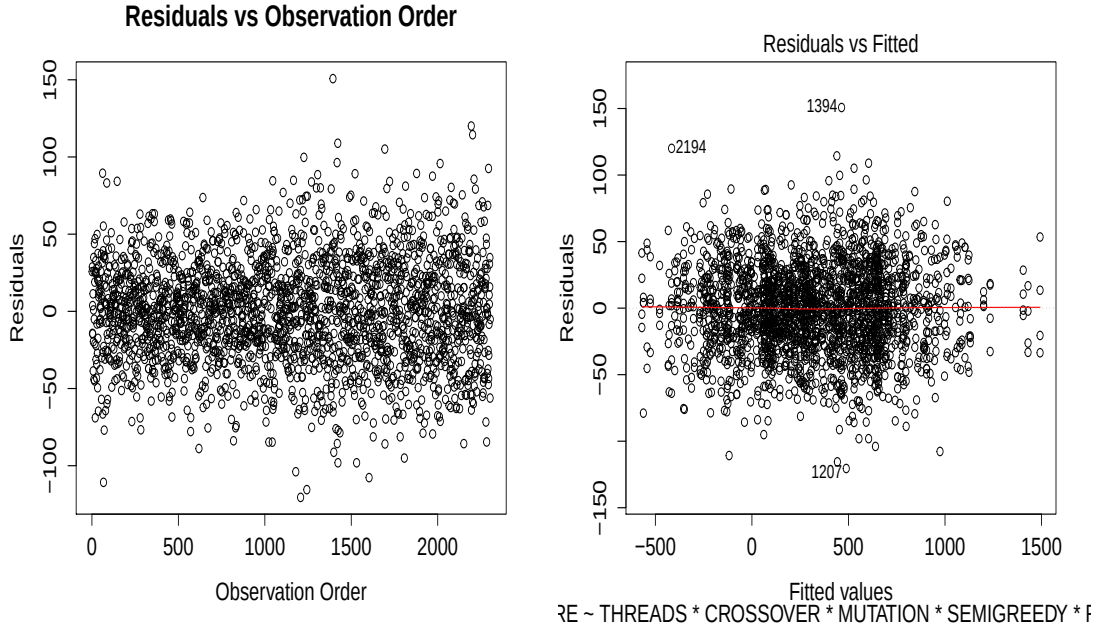
Analogamente à Tabela 9, relativa ao custo dos planos gerados, os resultados com respeito ao tempo de preparação dos planos são exibidos na Tabela 10. Pode-se observar que parte da variabilidade é devida aos fatores principais *THREADS*, *CROSSOVER*, *MUTATION*, *RD*, *SEMIGREEDY*, *LC* e *LQ*, bem como algumas interações de segunda ordem. Tal verificação permite a rejeição das hipóteses nulas relacionadas a um grau de confiança de 95%.

Tabela 10 – ANOVA - Tempo de Preparação

Source	**DF**	**SS**	**MS**	**Iter**	**Pr**
THREADS	1	3325076	3325076	5000	< 2e-16 ***
CROSSOVER	2	43739	21870	5000	< 2e-16 ***
THREADS:CROSSOVER	2	5974	2987	649	0.15100
MUTATION	1	6120	6120	3220	0.03012 *
THREADS:MUTATION	1	1	1	51	1.00000
CROSSOVER:MUTATION	2	2284	1142	1378	0.09797 .
SEMIGREEDY	1	2476886	2476886	5000	< 2e-16 ***
THREADS:SEMIGREEDY	1	24944	24944	5000	< 2e-16 ***
CROSSOVER:SEMIGREEDY	2	2171	1085	993	0.58006
MUTATION:SEMIGREEDY	1	0	0	51	1.00000
RD	1	757516	757516	5000	< 2e-16 ***
THREADS:RD	1	113443	113443	5000	< 2e-16 ***
CROSSOVER:RD	2	8201	4100	5000	< 2e-16 ***
MUTATION:RD	1	110	110	51	1.00000
SEMIGREEDY:RD	1	1386	1386	51	1.00000
RDP	1	95	95	51	1.00000
THREADS:RDP	1	538	538	1113	0.08266 .
CROSSOVER:RDP	2	1808	904	51	1.00000
MUTATION:RDP	1	17	17	51	1.00000
SEMIGREEDY:RDP	1	4	4	51	0.70588
RD:RDP	1	690	690	51	1.00000
LC	1	359475	359475	5000	< 2e-16 ***
THREADS:LC	1	55470	55470	5000	< 2e-16 ***
CROSSOVER:LC	2	10381	5190	5000	< 2e-16 ***
MUTATION:LC	1	52	52	51	1.00000
SEMIGREEDY:LC	1	237	237	51	1.00000
RD:LC	1	9069	9069	5000	< 2e-16 ***
RDP:LC	1	23	23	51	1.00000
LQ	1	830234	830234	5000	< 2e-16 ***
THREADS:LQ	1	15928	15928	5000	< 2e-16 ***
CROSSOVER:LQ	2	8376	4188	536	0.51866
MUTATION:LQ	1	18	18	51	1.00000
SEMIGREEDY:LQ	1	3333	3333	3103	0.03126 *
RD:LQ	1	10267	10267	5000	0.01760 *
RDP:LQ	1	60	60	516	0.16279
MUTATION:RDP:LQ	1	3234	3234	4692	0.02089 *
LC:LQ	1	6044	6044	5000	< 2e-16 ***
Residuals	1920 2592321	1350			

Signif. códigos: 0 ’*’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1**

Buscando novamente a análise da adequação do modelo gerado quanto as premissas inerentes ao tipo de teste realizado, apresenta-se na Figura 39, os gráficos de independência (Figura 39a) e homoscedasticidade (Figura 39b). Tal como no estudo anterior, não há nenhuma formação que evidencie alguma violação.



(a) *Independência* (b) *Homoscedasticidade*

Figura 39 – Premissas Modelo de Calibração - Tempo de Preparação.

Por fim, são exibidos nas Figuras 40 e 41 seguintes, o comportamento dos efeitos principais. É importante frisar que a escolha dos parâmetros foi realizada levando-se em conta a redução do custo e o tempo despendido para tal redução. Assim, com base nas significâncias apresentadas, primou-se por uma escolha balanceando entre ambos os fatores de saída analisados. Está claro que o número máximo de agentes igual a 8 implica em menos tempo de preparação e custos relativamente comparáveis aos resultados com 16 agentes, deste modo, escolheu-se um limite máximo de 8 agentes. Pode-se observar uma vantagem do método de cruzamento *PAGX* em relação aos outros, entretanto, optou-se por configurar a probabilidade de escolha em 50% para os métodos *PAGX* e *OX*. Distribuiu-se uma probabilidade de escolha igual a 50% para cada um dos métodos de mutação. De acordo com os gráficos relacionados ao fator *SEMIGREEDY*, o método *SCX* é o mais indicado. Com respeito ao fator *RD*, na tentativa de balancear entre custos mais baixos (movimento de realocação) e tempos de preparação menores (movimento de troca), partilhou igualmente a probabilidade de escolha entre ambos os movimentos. Sem significância na variabilidade dos resultados, o fator *RDE* foi fixado no valor 0.3. Por tempos de preparação menores e

custos competitivos, escolheu-se os valores 1 e 0.1 para os coeficientes *LFC* e *DCL*, respectivamente.

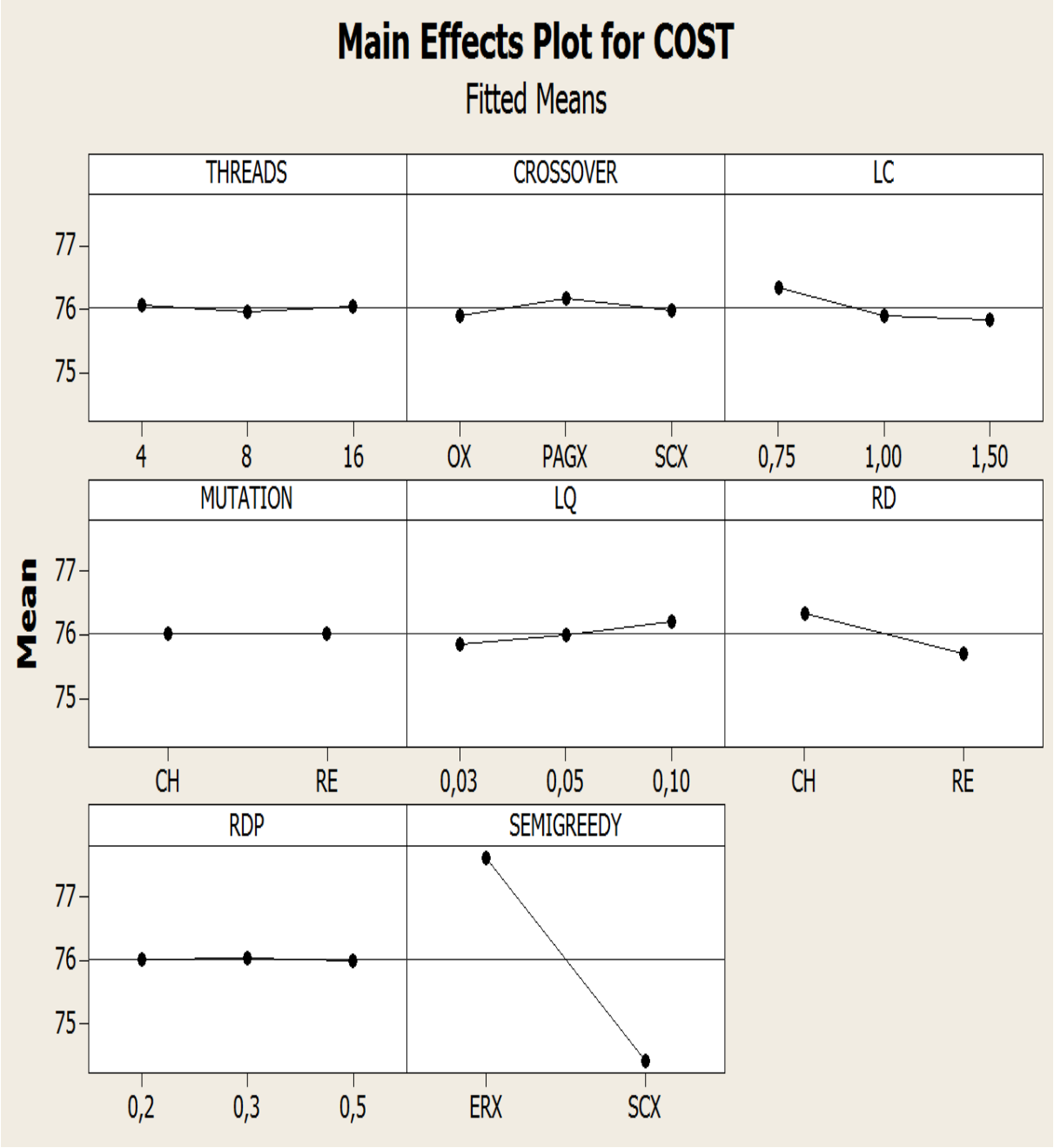


Figura 40 – Comportamento do Efeitos Principais - Custo dos Planos.

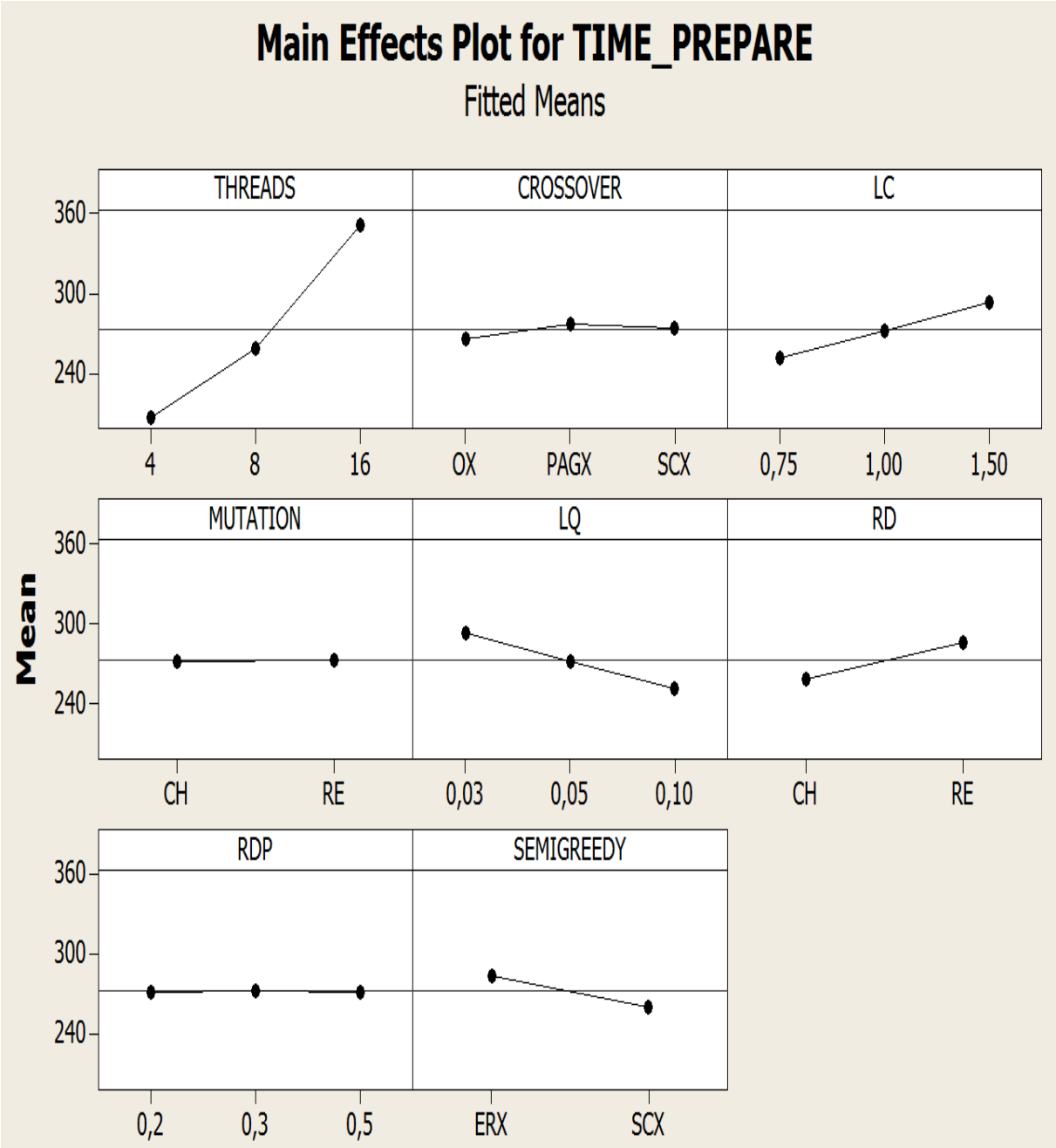


Figura 41 – Comportamento do Efeitos Principais - Tempo de Preparação.

6.3.2 Comparação com um Algoritmo Exaustivo

O propósito principal desta subseção é avaliar a capacidade do otimizador proposto em encontrar soluções ótimas ou ótimos globais. Um outro aspecto importante deste experimento, diz respeito ao uso do *TPC-DS* e, como dito anteriormente, tal *benchmark* relaciona-se ao suporte à tomada de decisões. O *TPC-DS* representa um modelo maduro e validado pela comunidade científica, o que acrescenta valor a análise experimental. Para alcançar o objetivo proposto, estudou-se dois algoritmos de ordenação de junções no SGBDR H2, o método exaustivo descrito na Seção 3.2 e o otimizador multiagente proposto.

Examinou-se nesta subseção o comportamento dos algoritmos em relação ao custo dos planos de execução gerados (em escala logarítmica), bem como o tempo gasto na preparação dos mesmos (em milissegundos). Desta forma, aplicou-se o teste-t pareado no estudo da média de amostras de 2 algoritmos diferentes. Por questões de compatibilidade com os recursos SQL disponíveis no H2 relacionados às funções de agregação do tipo *Window Functions* (**??**) , das 99 consultas presentes no *TPC-DS*, apenas 54 foram utilizadas. Realizou-se 10 replicações completas, totalizando 1080 execuções, distribuídas da seguinte forma: $N_c \times N_m \times N_r = 54 \times 2 \times 10 = 1080$, sendo N_c o total de consultas por método (54 consultas), N_m o número de métodos avaliados e N_r a quantidade de replicações. Vale ressaltar que um valor médio relativo à variável de saída foi utilizado como resultado, isto é, calculou-se para cada replicação o valor médio do custo e do tempo de preparação de todas as execuções, sendo o resultado deste cálculo repassado ao teste estatístico realizado. Isto significa que o comportamento dos algoritmos foi avaliado com base na média da execução de 54 consultas diferentes em 10 replicações. A hipótese testada em todas as comparações foi a não existência de diferença entre os dois algoritmos com relação às variáveis de saída.

Destaca-se a utilização do programa fornecido pela organização TPC para a construção do banco de dados e a geração das consultas-teste. Vale notar que o banco de testes construído alcançou um tamanho de 2.4GB. A Tabela 11 apresenta um resumo das informações retornadas pelo teste estatístico.

Tabela 11 – Teste-t - Exaustivo vs Multiagente - Custo dos Planos

Source	**N**	***Mean***	***StDev***	***SE Mean***
EXAUSTIVO	10	31,8572	0,0000	0,0000
MULTIAGENTE	10	31,8572	0,0000	0,0000
Difference	10	0,000000	0,000000	0,000000

95% CI for mean difference: (0,000000; 0,000000)
T-Test of mean difference = 0 (vs not = 0): T-Value = * P-Value = *

Pelos resultados apresentados na Tabela 11, verifica-se que não há diferença entre as metodologias em relação ao custo. Ambos os algoritmos possuem, em média, o mesmo custo para os planos gerados. Diante disto, aceita-se a hipótese testada com um grau de 95% de confiança, pois não há significância na variabilidade dos dados verificada pelo teste. A seguir, é apresentada a tabela de comparação entre as metodologias com relação ao tempo de preparação.

Tabela 12 – Teste-t - Exaustivo vs Multiagente - Tempo de Preparação

Source	**N**	***Mean***	***StDev***	***SE Mean***
EXAUSTIVO	10	8,80	4,29	1,36
MULTIAGENTE	10	7,30	1,06	0,33
Difference	10	1,50	4,65	1,47

95% CI for mean difference: (-1,83; 4,83)
T-Test of mean difference = 0 (vs not = 0): T-Value = 1,02 P-Value = 0,334

Pela Tabela 12 observa-se que não há significância estatística relativa à variabilidade dos dados. Portanto, aceita-se a hipótese de teste relacionada com 95% de confiança. Mais ainda, pode-se verificar que o tempo médio de preparação do otimizador multiagente é ligeiramente menor. Os dados permitem estimar a magnitude desta diferença em 1,5 milissegundos, com o intervalo de confiança de 95% variando de $-1,83$ a $4,83$.

Diante dos resultados apresentados, pode-se constatar que o otimizador proposto se comportou bem em problemas menos complexos, o quais envolvem a junção de até 8 relações. Além disso, o algoritmo multiagente apresentou um melhor desempenho que a outra abordagem, pois foi capaz de encontrar soluções tão boas quanto o exaustivo, mas em menor tempo. Analisando as consultas individualmente, comprovou-se que o algoritmo proposto foi capaz de encontrar as soluções ótimas em todas as execuções e obteve tempos de preparação menores que o método exaustivo em 31% dos casos.

6.3.3 Comparação com outros SGBDRs

O objetivo desta subseção é avaliar o comportamento do algoritmo proposto em relação à versão não-exaustiva padrão do H2 e outras abordagens presentes em 3 SGBDRs diferentes. Tal análise visa estudar as contribuições do algoritmo desenvolvido relacionadas ao tempo total gasto entre o processo de planejamento e posterior execução de um conjunto de consultas-teste envolvendo a ordenação de muitas junções. Deste modo, foram escolhidos 3 SGBDRs implementados na linguagem Java

e 1 sistema nativo implementado em C, são eles: H2[5], HSQLDB[6], Derby[7] and PostgreSQL[8], respectivamente.

A construção da base de dados, assim como o roteiro de problemas-teste seguiu as instruções apresentadas na Seção 6.2. De acordo com a metodologia proposta, a base de dados sintética gerada apresentou um tamanho de 2.7GB no H2, sendo ligeiramente maior que o tamanho da base padrão do *TPC-DS* no H2. Conforme o que foi introduzido da seção experimental, a área de cache dos SGBDRs foi fixada em 1.6GB, sendo insuficiente para comportar todos os dados da base de testes gerada. É importante notar que o tamanho das páginas carregadas no cache do H2 não coincide com o tamanho das páginas do sistema operacional (16KB), ou seja, o H2 utiliza outros componentes além dos 16KB, para estimar o tamanho de uma página, o que implica numa redução prática da área de cache. Além disso, todas as consultas obrigatoriamente acessam inicialmente as tuplas das relações do disco. Tais informações apenas destacam a tentativa de minimização da interferência do fator memória principal disponível para os experimentos.

No tocante ao roteiro de testes, foram aplicadas de 3 replicações completas. Além disso, um conjunto de 3 consultas-teste diferentes foi gerado por problema envolvendo a junção de 12, 14, 16, 18, 20, 30, 40 e 50 relações, sendo as mesmas distribuídas entre os formatos corrente, estrela e *snowflake*. No total, foram realizadas 1080 execuções computadas da seguinte forma: $N_t = N_c \times N_p \times N_f \times N_a \times N_r = 3 \times 8 \times 3 \times 5 \times 3 = 1080$, onde N_c é o número de consultas agrupadas por combinação, N_p o número de dimensões dos problemas testados, N_f o número de formatos dos problemas, N_a o número de algoritmos testados e N_r o número de replicações. O SGBDR H2 foi testado com dois otimizadores diferentes, sendo um deles o algoritmo não-exaustivo oficial descrito na Seção 3.2 e o outro, o método proposto neste trabalho, doravante denominados H2QO e MAQO, respectivamente. O planejamento fatorial do experimento em questão estuda os seguintes fatores principais descritos na Tabela 13.

Tabela 13 – Fatores e Níveis - Experimento Geral

Fator	**Nível**
DATABASE	H2QO, MAQO, HSQLDB, DERBY e POSTGRESQL
SHAPE	CHAIN, STAR e SNOWFLAKE
SIZE	12, 14, 16, 18, 20, 30, 40 e 50

O modelo de efeitos dos fatores é apresentado pela Equação (6.2).

[5] H2 - Version $1.3.174$, www.h2database.com
[6] HSQLDB - Version $2.0.0$, http://hsqldb.org/
[7] Derby - Version $10.9.1.0$, http://db.apache.org/derby
[8] PostgreSQL - Version $9.2.6$ - www.postgresql.org

$$y_{ij} = \mu + \tau_i + \beta_j + \gamma_k + (\tau\beta)_{ij} + (\tau\gamma)_{ik} + (\beta\gamma)_{jk} + (\tau\beta\gamma)_{ijk} +$$

$$\epsilon_{ijkl} \begin{cases} i = 1, \ldots, a \\ j = 1, \ldots, b \\ k = 1, \ldots, c \\ l = 1, \ldots, n \end{cases} \tag{6.2}$$

Na Equação 6.2, μ é a média geral. As variáveis τ_i, β_j e γ_k representam os efeitos principais dos níveis dos fatores *DATABASE*, *SHAPE* e *SIZE*, respectivamente. Já os componentes $(\tau\beta)_{ij}$, $(\tau\gamma)_{ik}$, $(\beta\gamma)_{jk}$ e $(\tau\beta\gamma)_{ijk}$ representam os efeitos das interações entre os fatores principais e ϵ_{ijk} é o resíduo. Definiu-se as hipóteses de teste como segue:

1. $\mathbf{H_0}$: $\tau_1 = \tau_2 = \ldots = \tau_a = 0$ (sem efeito no fator principal DATABASE);
   $\mathbf{H_1}$: Pelo menos um $\tau_i \neq 0$.

2. $\mathbf{H_0}$: $\beta_1 = \beta_2 = \ldots = \beta_b = 0$ (sem efeito no fator principal SHAPE);
   $\mathbf{H_1}$: Pelo menos um $\beta_j \neq 0$.

3. $\mathbf{H_0}$: $\gamma_1 = \gamma_2 = \ldots = \gamma_c = 0$ (sem efeito no fator principal SIZE);
   $\mathbf{H_1}$: Pelo menos um $\gamma_k \neq 0$.

4. $\mathbf{H_0}$: $(\tau\beta)_{11} = (\tau\beta)_{12} = \ldots = (\tau\beta)_{ab} = 0$ (sem efeito na interação dos fatores);
   $\mathbf{H_1}$: Pelo menos algum $(\tau\beta)_{ij} \neq 0$.

5. $\mathbf{H_0}$: $(\tau\gamma)_{11} = (\tau\gamma)_{12} = \ldots = (\tau\gamma)_{ac} = 0$ (sem efeito na interação dos fatores);
   $\mathbf{H_1}$: Pelo menos algum $(\tau\gamma)_{ik} \neq 0$.

6. $\mathbf{H_0}$: $(\beta\gamma)_{11} = (\beta\gamma)_{12} = \ldots = (\beta\gamma)_{bc} = 0$ (sem efeito na interação dos fatores);
   $\mathbf{H_1}$: Pelo menos algum $(\beta\gamma)_{jk} \neq 0$.

7. $\mathbf{H_0}$: $(\tau\beta\gamma)_{111} = (\tau\beta\gamma)_{112} = \ldots = (\tau\beta\gamma)_{abc} = 0$ (sem efeito na interação dos fatores);
   $\mathbf{H_1}$: Pelo menos algum $(\tau\beta\gamma)_{ijk} \neq 0$.

Destaca-se, ainda, que foi utilizado como resultado para cada iteração dos algoritmos, o valor médio da variável de saída referente ao grupo de 3 consultas-teste.

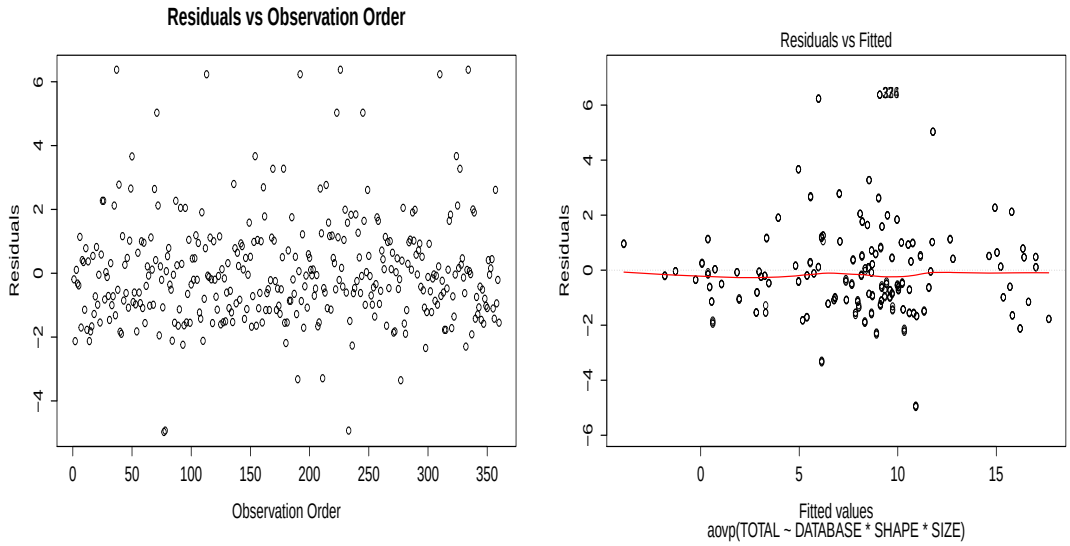
A execução do experimento em questão gastou cerca de 13 dias para seu encerramento. Finalmente, a Tabela 14 apresenta o resultado do teste estatístico realizado em relação ao tempo total gasto na otimização e execução das consultas.

Tabela 14 – ANOVA - Experimento Geral

Source	**DF**	**SS**	**MS**	**Iter**	**Pr**
DATABASE	4	1.05e+14	2.63e+13	5000	<2e-16 ***
SHAPE	2	3.05e+11	1.53e+11	326	0.45
DATABASE:SHAPE	8	1.74e+12	2.17e+11	5000	0.11
SIZE	1	6.91e+12	6.91e+12	5000	<2e-16 ***
DATABASE:SIZE	4	2.96e+13	7.40e+12	5000	<2e-16 ***
SHAPE:SIZE	2	3.11e+12	1.56e+12	5000	<2e-16 ***
DATABASE:SHAPE:SIZE	8	7.29e+12	9.11e+11	5000	<2e-16 ***
Residuals	330	5.15e+13	1.56e+11		

Signif. códigos: 0 '*' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1**

A tabela de resultados anterior mostra que o fator principal *DATABASE* e suas interações apresentaram significância, sugerindo que parte da variabilidade dos dados inerente ao tempo de execução das consultas é explicada pelos sistemas de banco de dados testados e suas interações com o formato das consultas e as dimensões dos problemas. Deste modo, pode-se rejeitar com 95% de confiança as hipóteses nulas relacionadas. Além disso, como pode ser visto nas Figuras 42a e 42b, não há uma formação clara que invalide as premissas de independência e homoscedasticidade relacionadas à adequação do modelo gerado.



(a) *Independência* (b) *Homoscedasticidade*

Figura 42 – Premissas Modelo Experimento Geral - Tempo Total.

O comportamento médio dos fatores principais estudados é descrito na Figura 43. Pelo gráfico apresentado, é notável que o SGBDR *DERBY* foi o que mais demandou tempo para execução das consultas. Além disso, observa-se no fator *DATABASE* que o nível MAQO requereu menos tempo para execução do roteiro de testes, isto é, o SGBDR H2 utilizando o otimizador proposto foi, em média, o sistema mais rápido dos experimentos computacionais realizados. Com relação à dimensão dos problemas-teste, destaca-se que consultas envolvendo a junção de 20 ou mais relações consumiram um maior tempo médio de execução, com destaque para 40 relações, a dimensão com maior demanda na média. Por fim, o gráfico do fator *SHAPE* apresenta uma ligeira diferença entre os formatos avaliados.

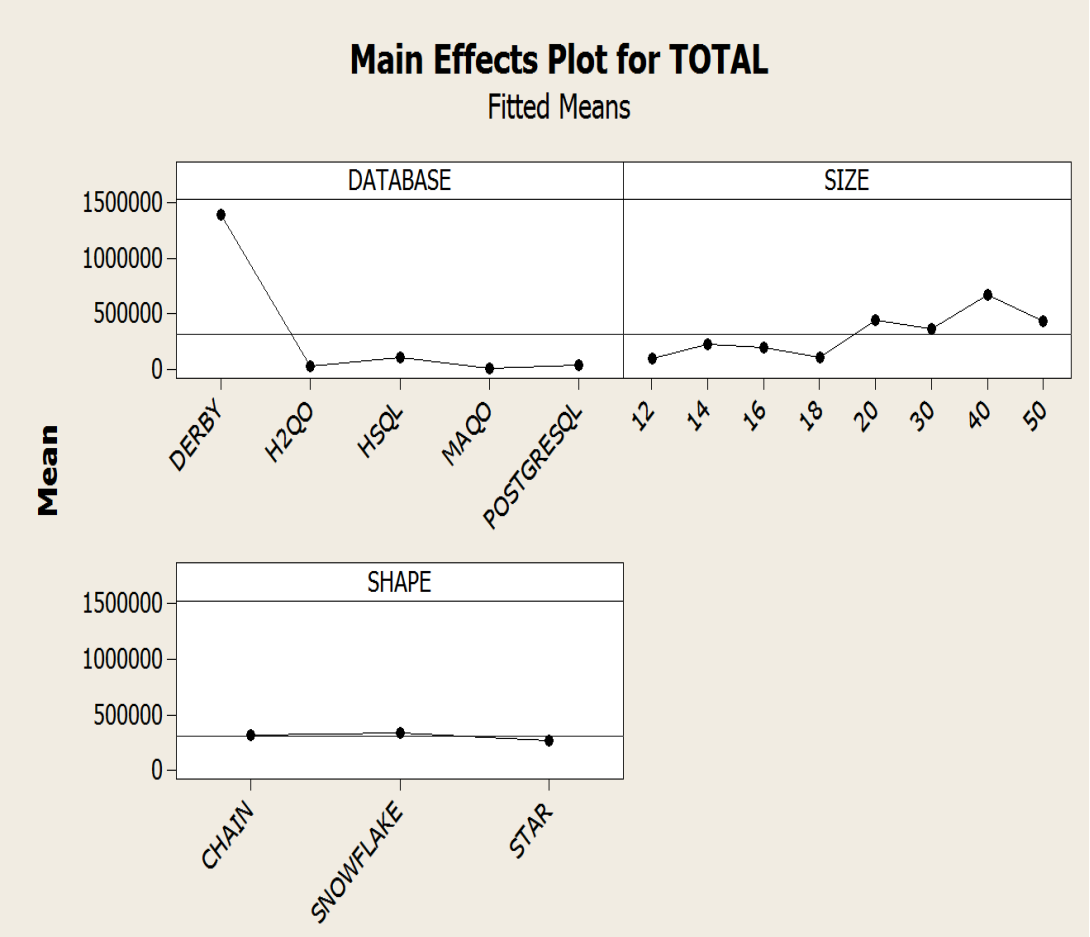


Figura 43 – Comportamento dos Efeitos Principais - Tempo Total.

Os gráficos das interações entre os fatores principais são exibidos na Figura 44. O comportamento apresentado reforça que o *DERBY* gastou mais tempo de execução que os outros SGBDRs em todas as dimensões de problemas e formatos de grafo testados. Além disso, pode-se constatar maiores tempos de processamento requeridos em problemas com as dimensões 14 e 20 para o *HSQL* e para problemas com a junção de 30 relações no *POSTRESQL*. Na interação específica entre os sistemas testados e o formato das consultas, está claro que o *DERBY* gastou mais tempo em consultas do tipo *CHAIN* e menos tempo no formato *STAR*. Porém, não é possível avaliar mais seguramente o comportamento dos demais SGBDRs. Em relação à interação entre os fatores *SHAPE* e *SIZE*, é possível notar um uso maior do tempo de processamento em problemas de dimensão 40 e 50 no nível *CHAIN* e no formato *STAR* para problemas com 40 relações.

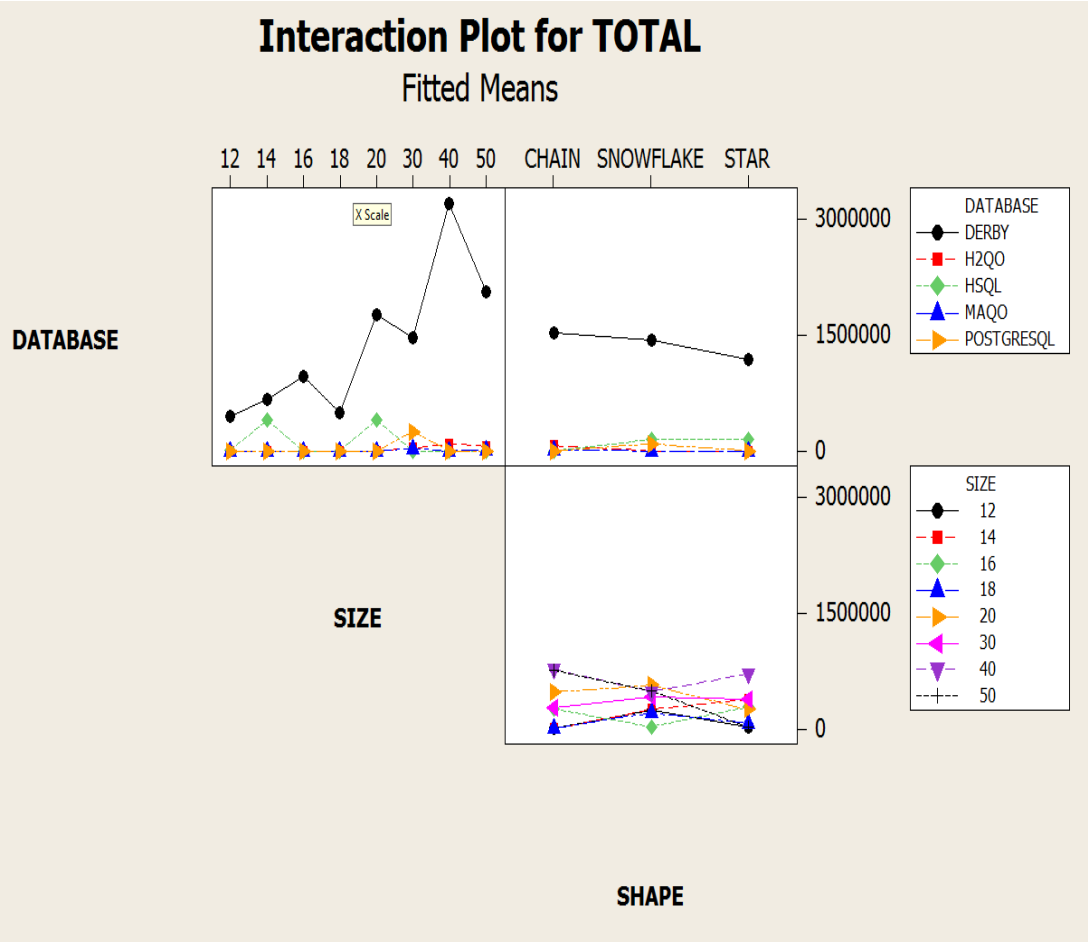


Figura 44 – Comportamento das Interações dos Efeitos Principais - Tempo Total.

Para fortalecer a avaliação realizada acerca da interação entre os fatores *DATABASE* e *SIZE*, apresenta-se na Figura 45 outro gráfico com o comportamento da interação em questão. Assim, além das observações relacionadas anteriormente, constata-se que problemas com 40 relações implicaram em mais tempo de processamento nos níveis *DERBY* e *H2QO*. Além disso, nota-se na média, que problemas com a dimensão 30 acarretaram em um ligeiro aumento no tempo de execução no SGBDR H2 utilizando o otimizador proposto (nível *MAQO*).

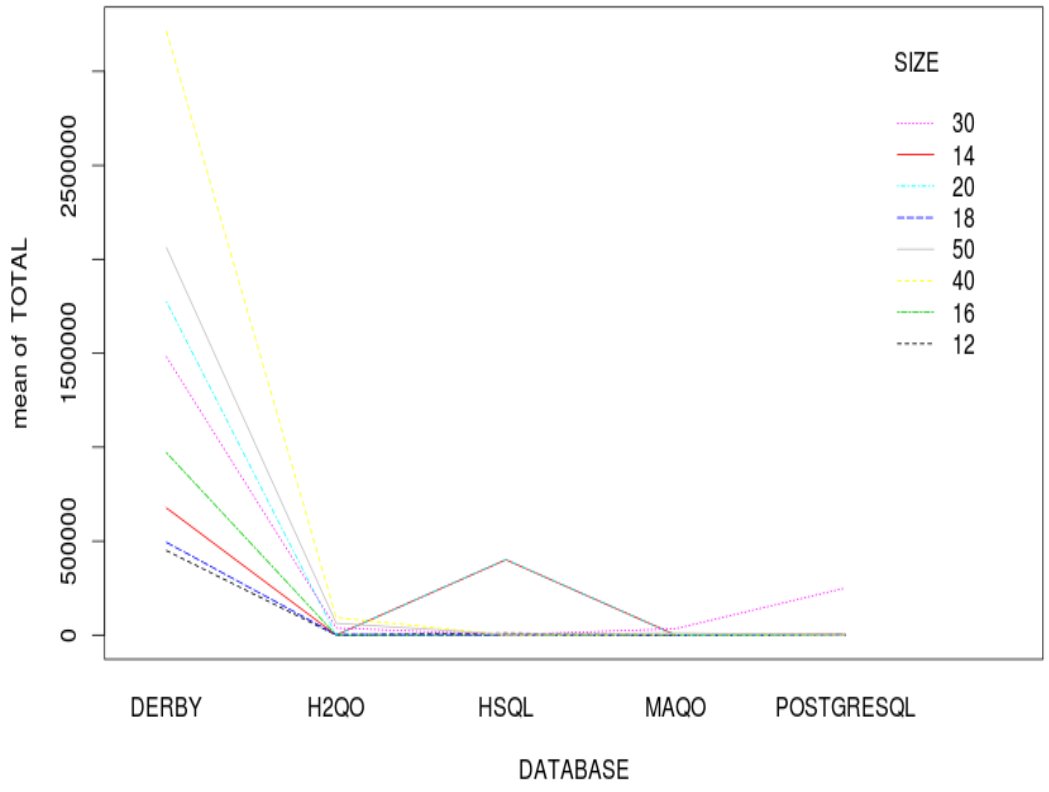


Figura 45 – Comportamento da Interação entre *DATABASE* e *SIZE* - Tempo Total.

A Figura 46 exibe a interação entre os fatores *SHAPE* e *SIZE* sob outra perspectiva. Tal figura também visa a trazer mais clareza à análise realizada anteriormente sobre a interação citada. Deste modo, com base no gráfico apresentado, além das observações previamente levantadas, conclui-se que as consultas no formato *snowflake* gastam, em média, mais tempo de execução em problemas com 20 relações. Em adição, consultas-teste nos formatos corrente, estrela e *snowflake* demandam menos tempo em problemas com as dimensões 14, 12 e 16, respectivamente.

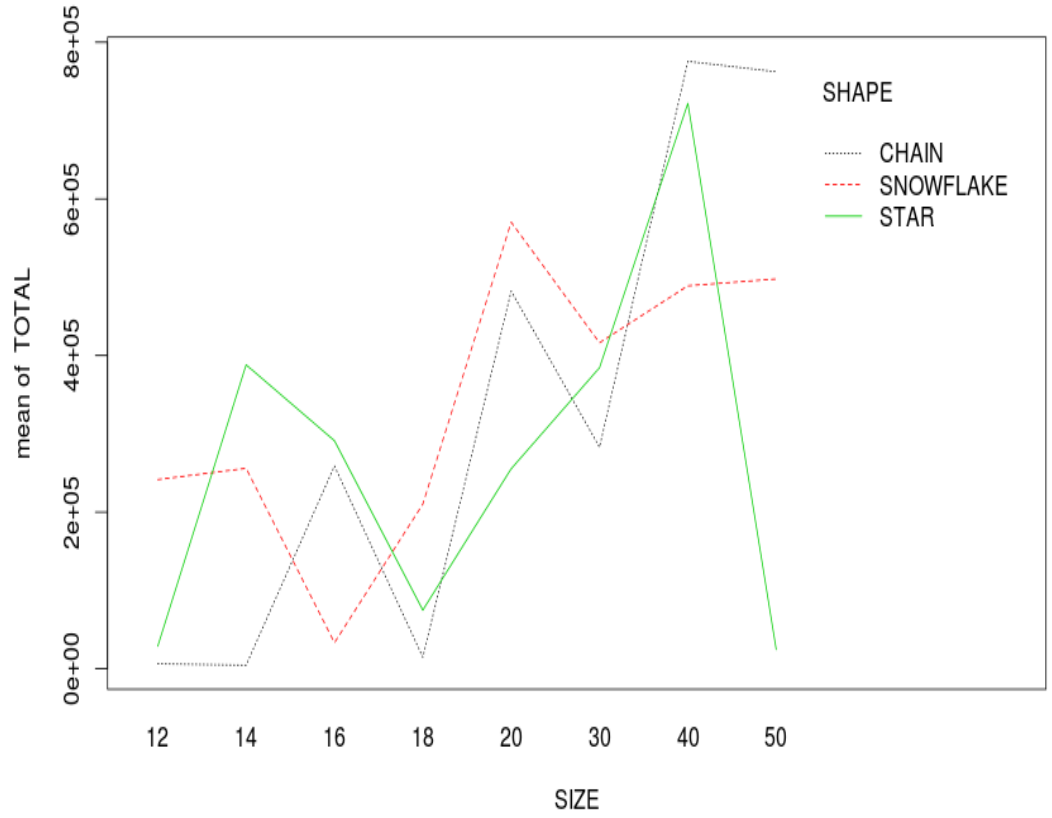


Figura 46 – Comportamento da Interação entre *SHAPE* e *SIZE* - Tempo Total.

A Tabela 14 mostrou que a interação entre os fatores *DATABASE* e *SHAPE* não tem significância estatística na variabilidade dos dados. A parte disso, a Figura 47 tem como objetivo apresentar uma disposição mais clara da interação entre os fatores relacionados. À vista disso, pode-se reparar um maior tempo de processamento dos SGBDRs *DERBY* e *H2QO* no formato corrente. Já o SGBDR *HSQL* demandou mais tempo em consultas nos formatos estrela e *snowflake*, ao passo que o *POSTRESQL* gastou mais tempo no formato *snowflake*. Não há uma distinção clara de esforço no nível *MAQO*. Por fim, excluindo o *HSQL*, nota-se que na média o formato estrela necessitou de menos tempo para sua execução.

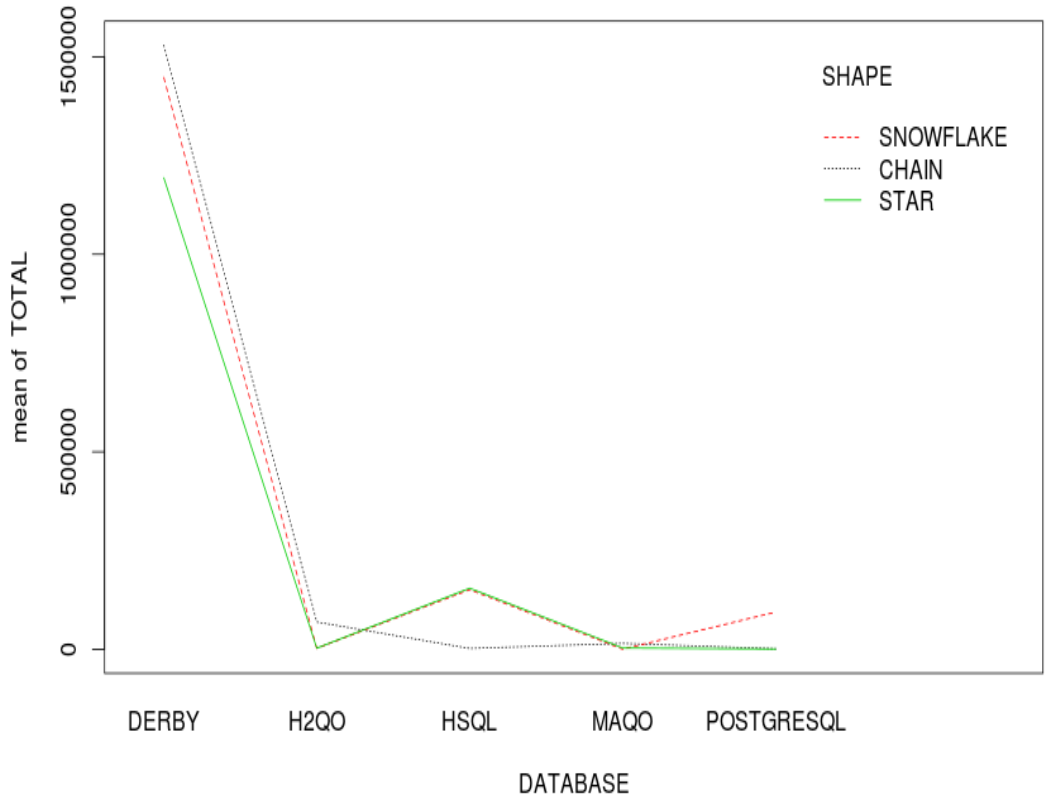


Figura 47 – Comportamento da Interação entre *DATABASE* e *SHAPE* - Tempo Total.

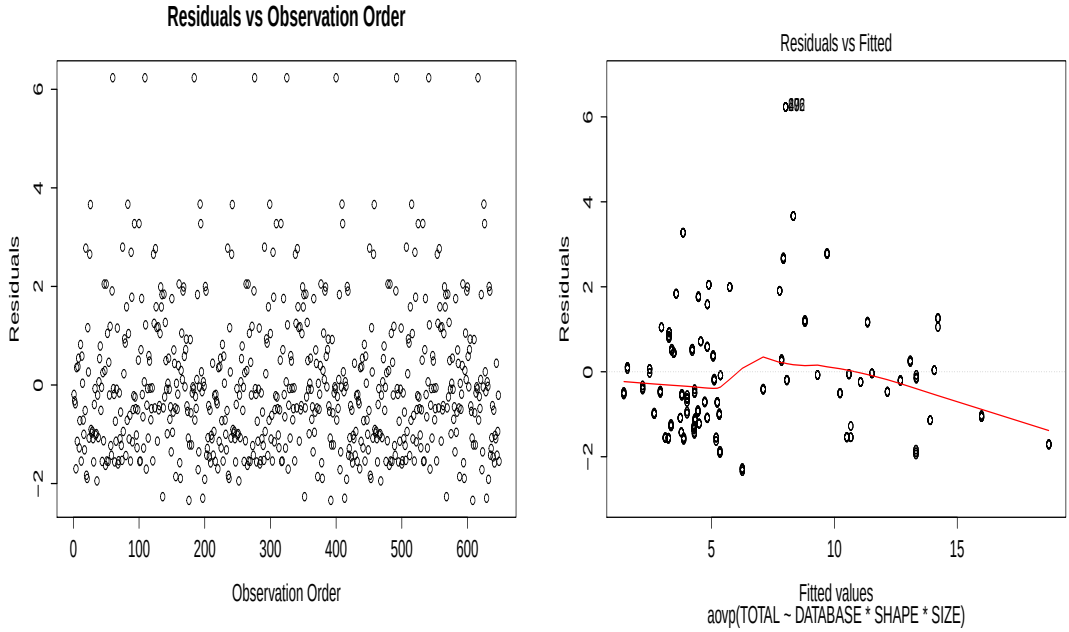
É importante observar que os SGBDRs que gastaram menos tempo de execução no experimento foram: *H2QO*, *MAQO* e *PostgreSQL*. Assim, buscando um estudo mais centrado na variabilidade dos tempos de execução desses sistemas, expõem-se na Tabela 15 os resultados do teste estatístico com o fator *DATABASE* restrito aos sistemas mais rápidos. Os resultados atestam que os fatores principais e suas interações são significantes estatisticamente. Dessarte, rejeita-se todas as hipóteses nulas levantas a 95% de confiança. Pela significância apresentada, é permitido dizer que parte da variabilidade dos dados pode ser explicada pelos fatores principais e suas interações, isto é, mesmo restrito aos 3 sistemas mais rápidos, a variação dos tempos de execução sofre influência de tais sistemas, formatos das consultas, dimensão dos problemas e as interações relacionadas.

Tabela 15 – ANOVA - Experimento Geral Reduzido

Source	**DF**	**SS**	**MS**	**Iter**	**Pr**
DATABASE	2	7.73e+10	3.86e+10	5000	<2e-16 ***
SHAPE	2	1.16e+11	5.79e+10	5000	<2e-16 ***
DATABASE:SHAPE	4	5.21e+11	1.30e+11	5000	<2e-16 ***
SIZE	1	1.61e+11	1.61e+11	5000	<2e-16 ***
DATABASE:SIZE	2	7.59e+10	3.79e+10	5000	<2e-16 ***
SHAPE:SIZE	2	1.13e+11	5.64e+10	5000	<2e-16 ***
DATABASE:SHAPE:SIZE	4	3.06e+11	7.65e+10	5000	<2e-16 ***
Residuals	630 4.50e+12	7.15e+09			

Signif. códigos: 0 '*' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1**

Como parte do processo estatístico aplicado, a validação da adequação do modelo em relação às premissas determinadas é feita com base nos gráficos 48a e 48b. Em relação à independência, não há a formação clara de uma estrutura padrão em relação ao espalhamento dos resíduos. Quanto à homoscedasticidade, não há um padrão claro da igualdade de variância em relação aos valores ajustados.



(a) *Independência* (b) *Homoscedasticidade*

Figura 48 – Premissas Modelo Experimento Geral Reduzido - Tempo Total.

A Figura 49 destaca o comportamento médio do fator *DATABASE* restrito aos níveis *H2QO*, *MAQO* e *POSTGRESQL* e os demais fatores. Vale ressaltar que o comportamento médio da Figura 49 é o mesmo da apresentado no gráfico 43. Entretanto, com a redução dos níveis do fator *DATABASE*, o gráfico a seguir apresenta com maior clareza a distinção do tempo médio total gasto pelos sistemas de banco de dados

relacionados. Assim, verifica-se com mais nitidez que o nível *MAQO* gastou menos tempo de processamento que os demais sistemas na média. Quanto ao tamanho dos problemas estudados, constata-se tempos médios parecidos para consultas com 12 a 20 relações e um maior gasto com o processamento em consultas com 30 relações. Por fim, nota-se que problemas no formato estrela demandam menos tempo médio de processamento e o formato *snowflake* exige mais tempo.

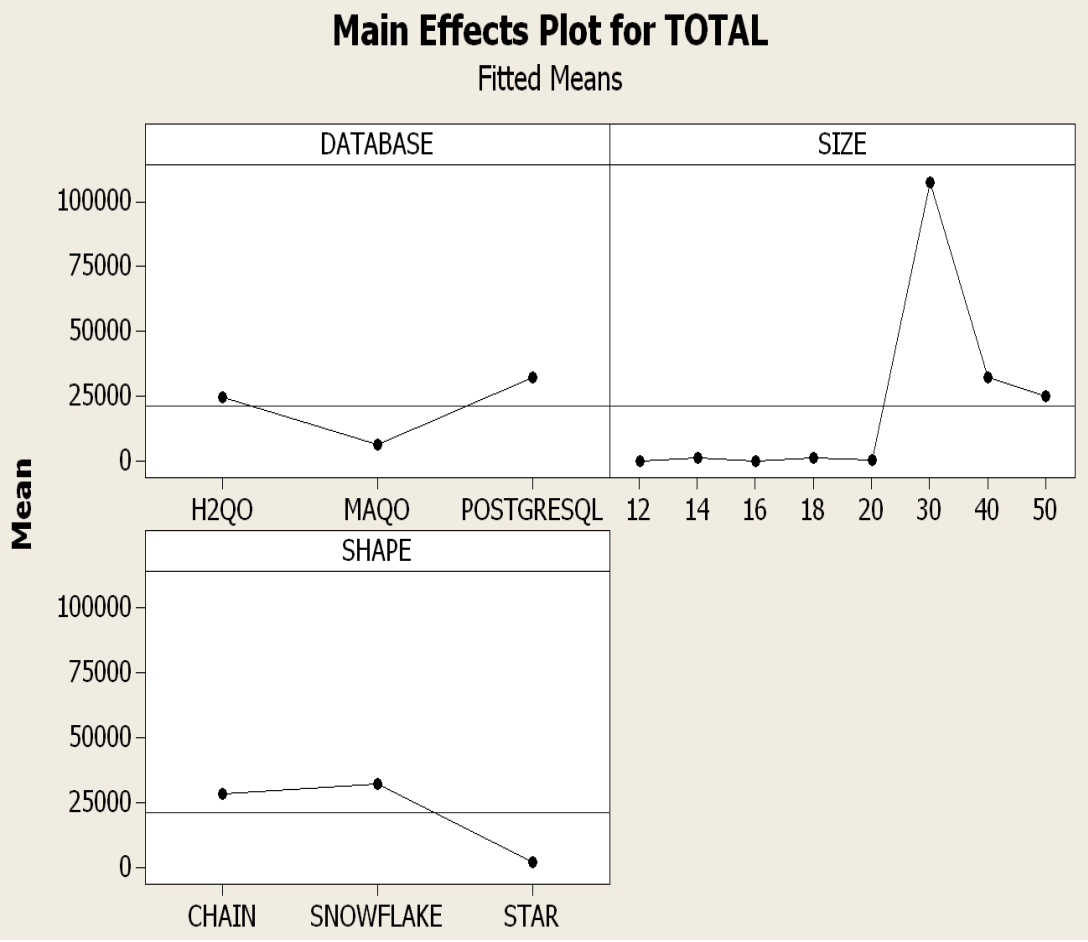


Figura 49 – Comportamento do Fator *DATABASE* Reduzido - Tempo Total.

Como destacado no capítulo 1, elege-se como maior contribuição deste trabalho, o desenvolvimento de uma nova abordagem para a otimização de muitas junções. Ante este fato, ressalta-se a grande importância de um estudo acerca das contribuições do algoritmo proposto em relação ao otimizador não-exaustivo oficial do H2 no tocante ao tempo de preparação e o custo dos planos gerados. Desta maneira, reduzindo-se o fator *DATABASE* aos níveis H2QO e MAQO, apresenta-se a seguir, estudos focados no tempo de preparação e do custo das soluções destes otimizadores.

Começando pela análise do custo dos planos gerados, a Tabela ANOVA 16 descreve os resultados do teste executado. Percebe-se a indicação de significância

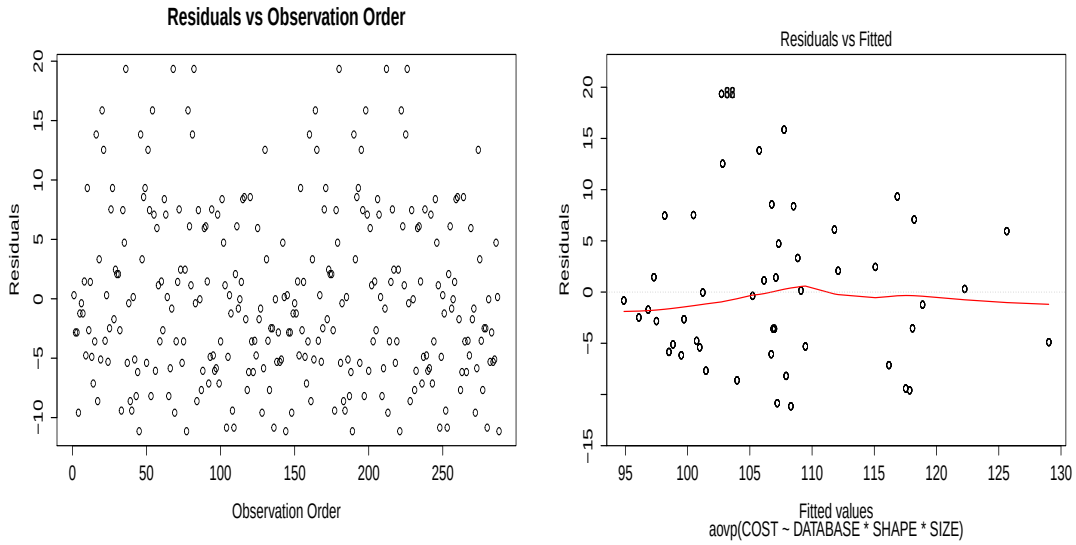
nos fatores principais, bem como em suas interações. Novamente, rejeitam-se as hipóteses relacionadas.

Tabela 16 – ANOVA - Experimento H2 - Custo

Source	**DF**	**SS**	**MS**	**Iter**	**Pr**
DATABASE	1	3.37e+182	3.37e+182	5000	0.0032 **
SHAPE	2	2.31e+183	1.15e+183	5000	<2e-16 ***
DATABASE:SHAPE	2	6.74e+182	3.37e+182	5000	<2e-16 ***
SIZE	1	4.37e+183	4.37e+183	5000	<2e-16 ***
DATABASE:SIZE	1	1.28e+183	1.28e+183	5000	<2e-16 ***
SHAPE:SIZE	2	8.74e+183	4.37e+183	5000	<2e-16 ***
DATABASE:SHAPE:SIZE	2	2.55e+183	1.28e+183	5000	<2e-16 ***
Residuals	276	1.44e+184	5.20e+181		

Signif. códigos: 0 ’*’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1**

A verificação da não-violação das premissas de independência e homoscedasticidade podem ser verificadas nas figuras 50a e 50b, respectivamente.



(a) *Independência* (b) *Homoscedasticidade*

Figura 50 – Premissas Modelo Experimento H2 - Custo.

A análise do comportamento do custo médio em relação aos fatores principais é mostrada na Figura 51.

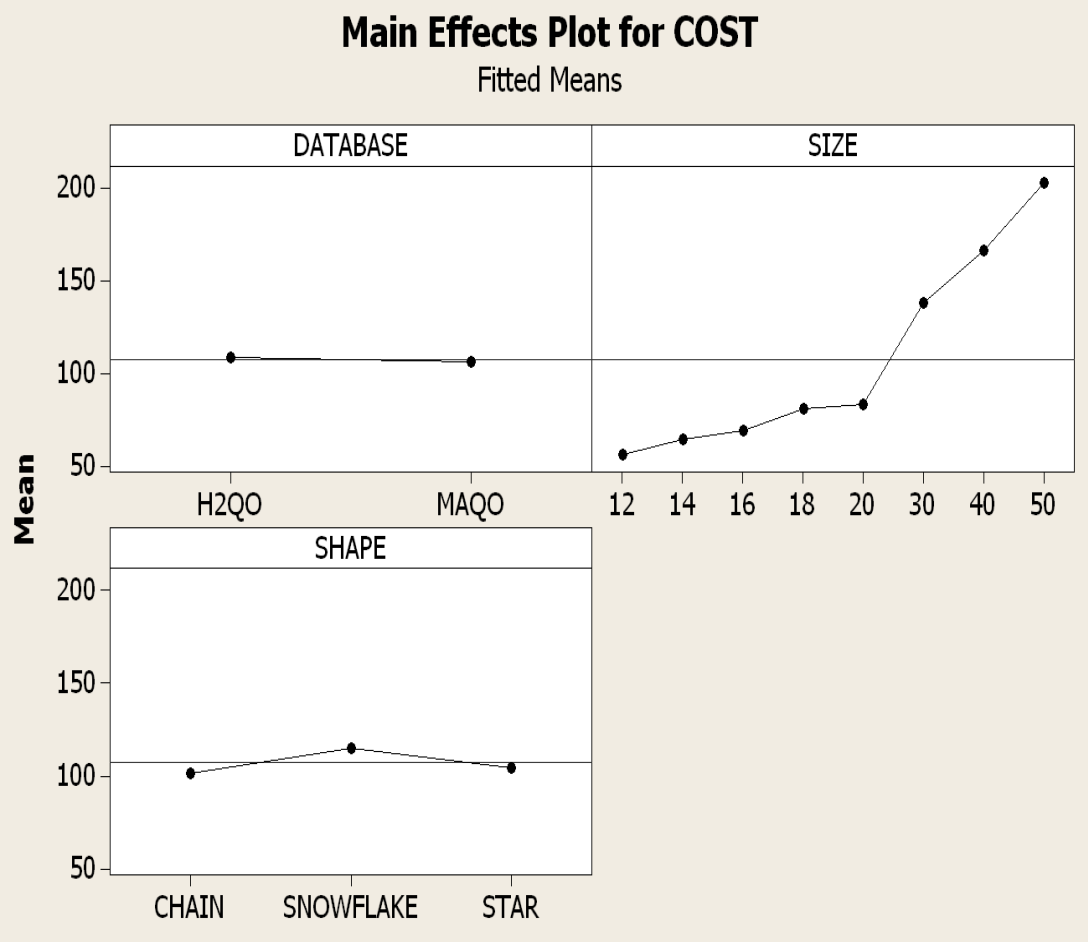


Figura 51 – Comportamento do H2 - Custo dos Planos - Escala Logarítmica.

Observa-se no gráfico de custos uma ligeira diferença do custo médio entre os otimizadores. Contudo, é válido citar que o custo médio está em escala logarítmica, o que reduz a percepção do tamanho da diferença entre os custos gerados por cada otimizador. Diante disto, é apresentado um outro gráfico (Figura 52), cujo custo médio do fator *DATABASE* não está em escala logarítmica. Quanto ao tamanho dos problemas, constata-se um custo crescente em relação à dimensão dos problemas. Finalmente, nota-se uma ligeira diferença entre os custos nos formatos de consulta estudados.

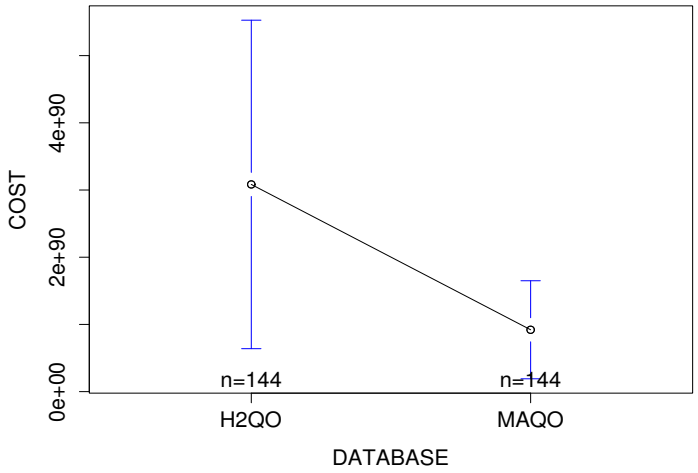


Figura 52 – Comportamento do Fator *DATABASE* no H2 - Custo dos Planos.

Ainda em relação ao custo, apresenta-se na Figura 53, a interação entre os fatores principais. Nesta figura, destaca-se a tendência do *MAQO* relativa a capacidade de gerar, na média, planos mais baratos a partir de problemas com 20 relações.

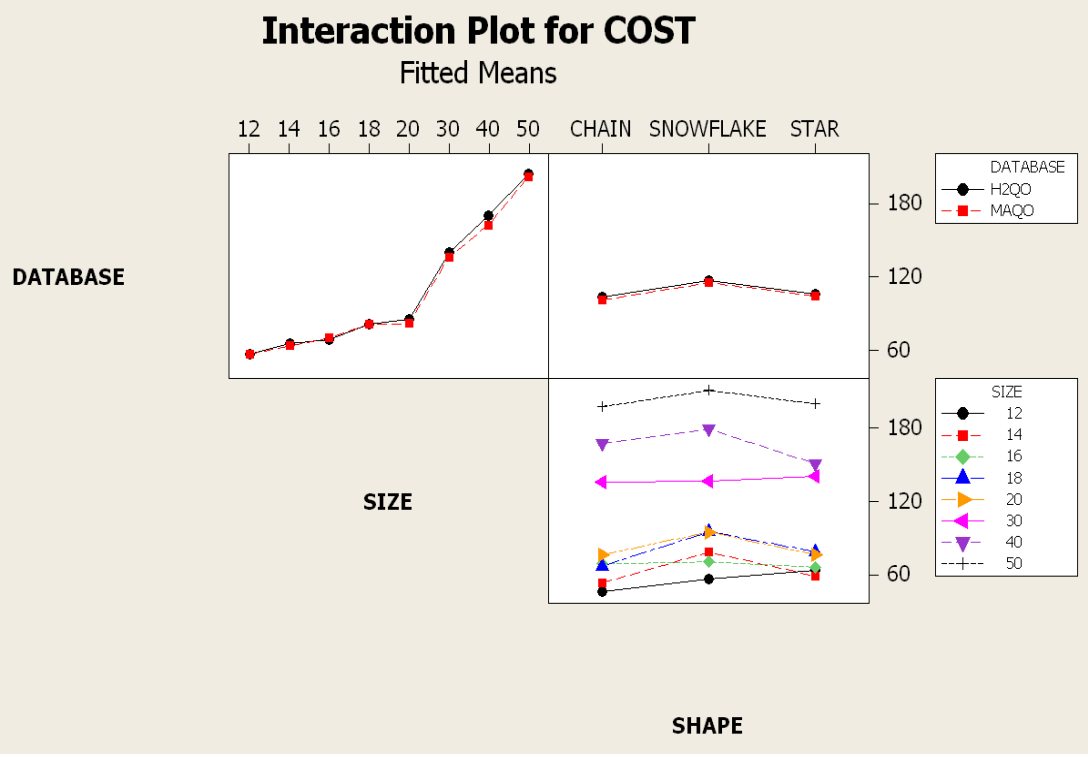


Figura 53 – Comportamento das Interações dos Efeitos Principais do H2 - Custo.

Prosseguindo a avaliação dos otimizadores testados no SGBDR H2, analisa-se agora o tempo gasto na preparação dos planos de consulta. Como pode ser visto na

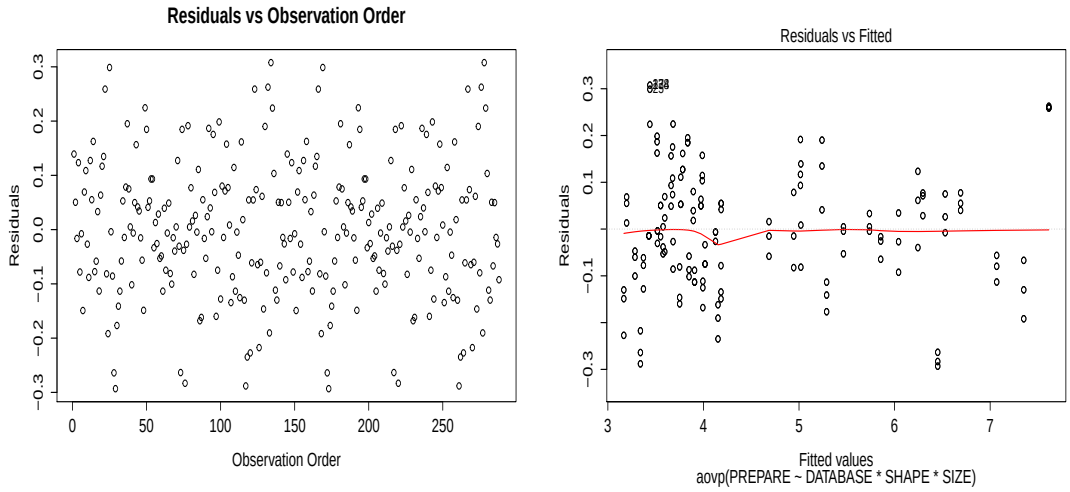
Tabela ANOVA 17, há significância na variabilidade dos dados atribuída aos fatores principais e suas interações. Portanto, rejeita-se com 95% de confiança as hipóteses relacionadas.

Tabela 17 – ANOVA - Experimento H2 - Tempo de Preparação

Source	**DF**	**SS**	**MS**	**Iter**	**Pr**
DATABASE	1	89324	89324	5000	<2e-16 ***
SHAPE	2	56673	28336	5000	<2e-16 ***
DATABASE:SHAPE	2	73611	36806	5000	<2e-16 ***
SIZE	1	650490	650490	5000	<2e-16 ***
DATABASE:SIZE	1	302344	302344	5000	<2e-16 ***
SHAPE:SIZE	2	130672	65336	5000	<2e-16 ***
DATABASE:SHAPE:SIZE	2	97161	48581	5000	<2e-16 ***
Residuals	276	266819	967		

Signif. códigos: 0 ’*’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1**

As premissas relacionadas à adequação do modelo gerado são validadas nas Figuras 54a e 54b.



(a) *Independência* (b) *Homoscedasticidade*

Figura 54 – Premissas Modelo Experimento H2 - Tempo de Preparação.

O tempo médio gasto na preparação dos planos de consulta associado a cada um dos fatores principais está disposto na Figura 55. Repara-se na figura em questão que o otimizador não-exaustivo do H2 consome, em média, menos tempo de planejamento que o algoritmo proposto neste trabalho. Mais, o tempo de preparação cresce na medida em que o número de relações da consulta aumenta. Quanto ao formato das consultas, o modelo *snowflake* é o que demanda um maior tempo médio de preparação.

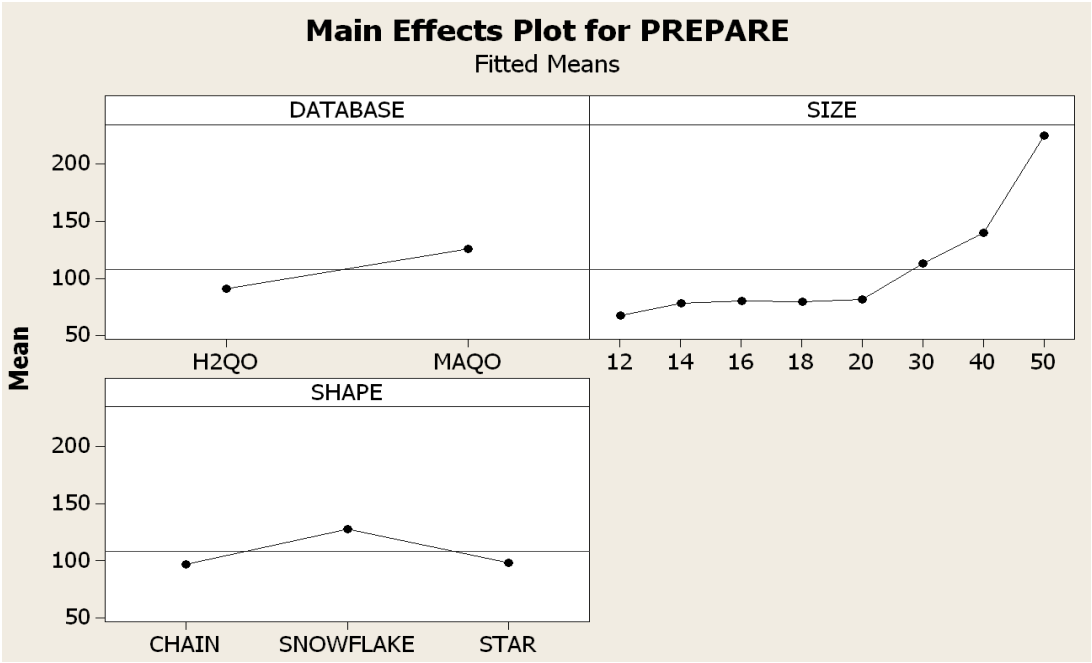


Figura 55 – Comportamento do H2 - Tempo de Preparação.

A interação entre os fatores principais está exposta na Figura 56. Verifica-se em média que o otimizador proposto foi capaz de preparar mais rapidamente as consultas com até 16 relações, ao passo que a partir de 18 relações, a metodologia empregada pelo otimizador oficial do H2 é mais rápida no planejamento dos planos.

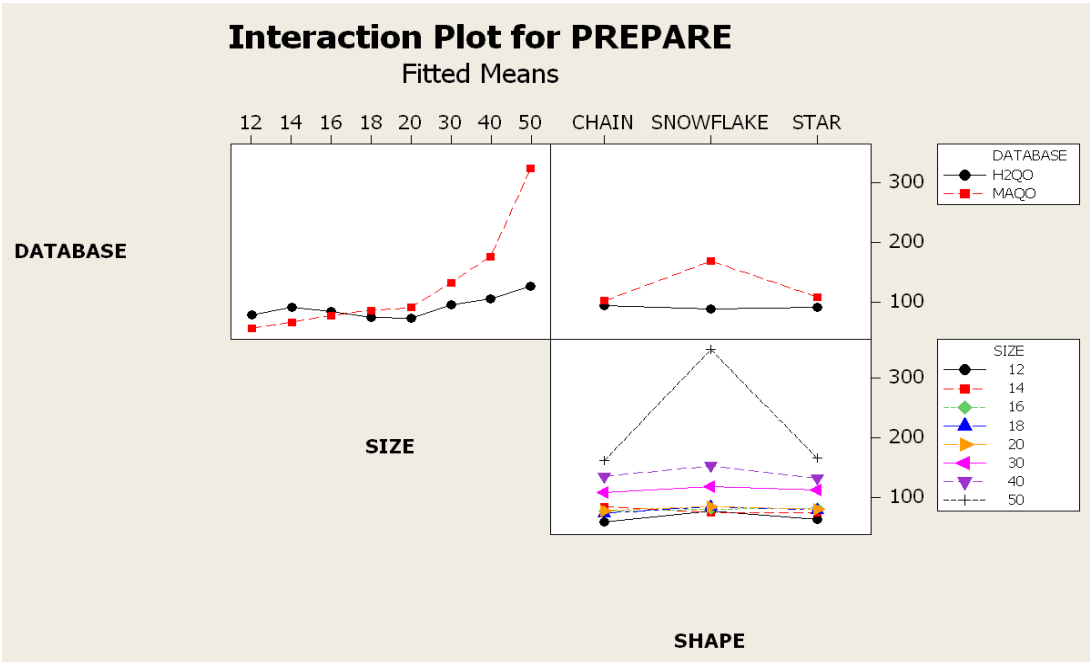


Figura 56 – Comportamento das Interações dos Efeitos Principais do H2 - Tempo de Preparação.

Embora relatado anteriormente que o otimizador padrão do H2 foi capaz de planejar mais rapidamente consultas com 18 ou mais relações, tal vantagem na preparação não resultou em consultas executadas em tempos totais menores, considerando-se o tempo total como sendo o tempo de planejamento somado ao tempo de execução. Como já discutido, o otimizador multiagente mostrou uma maior capacidade em encontrar planos mais baratos, o que refletiu em vários casos em tempos de execução menores. Portanto, mesmo com planejamentos mais demorados, o resultado médio do tempo total gasto na execução das consultas mostra que as execuções mais rápidas compensam o planejamento mais custoso no *MAQO*. A Figura 57 exibe o comportamento médio dos otimizadores e mostra que o *MAQO* possui um tempo total menor em várias dimensões de problemas.

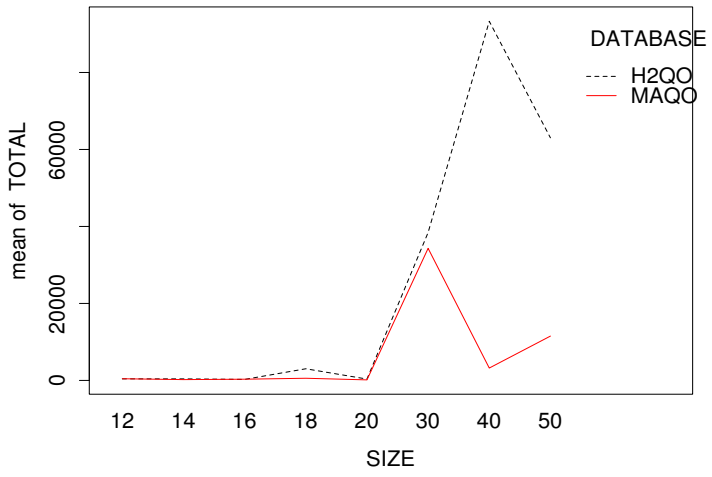


Figura 57 – Comportamento do Fator *DATABASE* no H2 - Tempo Total x Dimensão Problema.

6.4 Discussão dos resultados

Seguindo a ordem da avaliação experimental realizada, inicia-se a discussão dos resultados pelo experimento da Seção 6.3.2, onde foi empregado o *TPC-DS* na avaliação do otimizador exaustivo oficial do H2 e o algoritmo proposto. Dessarte, foi observado que o otimizador desenvolvido neste trabalho encontrou a solução ótima em todos os experimentos. Tal afirmação toma como base a comparação do custo dos planos gerados em relação aos custos retornados pela metodologia exaustiva. Mais ainda, o tempo de preparação do algoritmo multiagente foi menor, onde constatou-se uma redução do mesmo em 31% dos casos em comparação ao exaustivo. Vale observar que os parâmetros calibrados na Seção 6.3.1 foram aplicados tanto nos ex-

perimentos com o *TPC-DS* que possui problemas menos complexos, quanto no *benchmark* sintético (Seção 6.3.3) composto por consultas mais complexas. Um aspecto interessante da observação anterior é o fato de que o algoritmo multiagente apresenta tempos de preparação menores e custos equivalentes à metodologia exaustiva, ao passo que executa consultas mais rapidamente que o método não-exaustivo, e tudo isso com a utilização dos mesmos parâmetros. É notável a relação utilizada pelo otimizador multiagente na definição do esforço de otimização necessário nos diferentes tipos de problemas, pois o otimizador se mostrou eficiente em grade parte das consultas avaliadas. Não menos importante, é válido destacar que otimizador proposto se comportou bem em problemas envolvendo a junção de até 8 relações em um *benchmark* validado pela comunidade científica.

Prosseguindo a discussão dos resultados, abordou-se a seguir os efeitos do *benchmark* sintético construído, bem como das consultas-teste geradas com o foco em problemas com muitas junções. O experimento da Seção 6.3.3 tratou a comparação de vários SGBDRs: H2, HSQLDB, *Derby* e *PostgreSQL*. Objetivou-se a avaliação do tempo total gasto, o qual inclui a preparação dos planos das consultas e a execução dos mesmos. Assim, destaca-se a análise do fator *DATABASE*, que define o comportamento dos vários SGBDRs testados, onde é indiscutível a superioridade do SGBDR H2 utilizando o otimizador proposto, visto que apresentou o menor tempo médio de execução. O tempo total gasto por cada SGBDR para a execução completa do roteiro de testes é descrito na Tabela 18.

Tabela 18 – Resumo - Tempo Total de Execução do Roteiro de Testes

SGBDR	**Tempo**
MAQO	22m
H2QO	89m
HSQL	370m
Derby	5004m
PostgreSQL	116m

O otimizador proposto levou o SGBDR H2 a uma redução de 67 minutos no tempo total gasto para execução do roteiro de testes, resultando em uma melhora de mais de 300%. Mais ainda, a versão MAQO foi cerca de 5 vezes mais rápida que o popular SGBDR *PostgreSQL*. Tais resultados atestam que em uma comparação com o algoritmo genético do *PostgreSQL*, o uso da otimizador multiagente possibilitou uma melhoria de quase 400% nos tempos gastos para a execução das consultas. Analogamente, confrontando-se o algoritmo baseado na metodologia busca em profundidade do *Derby* ao sistema multiagente desenvolvido, verifica-se que H2 foi cerca de 171 vezes mais eficiente. Com o intuito de avaliar mais claramente o comportamento dos

sistemas nas diversas dimensões de problemas, apresenta-se na Figura 58 um gráfico sem os resultados do SGBDR *Derby*.

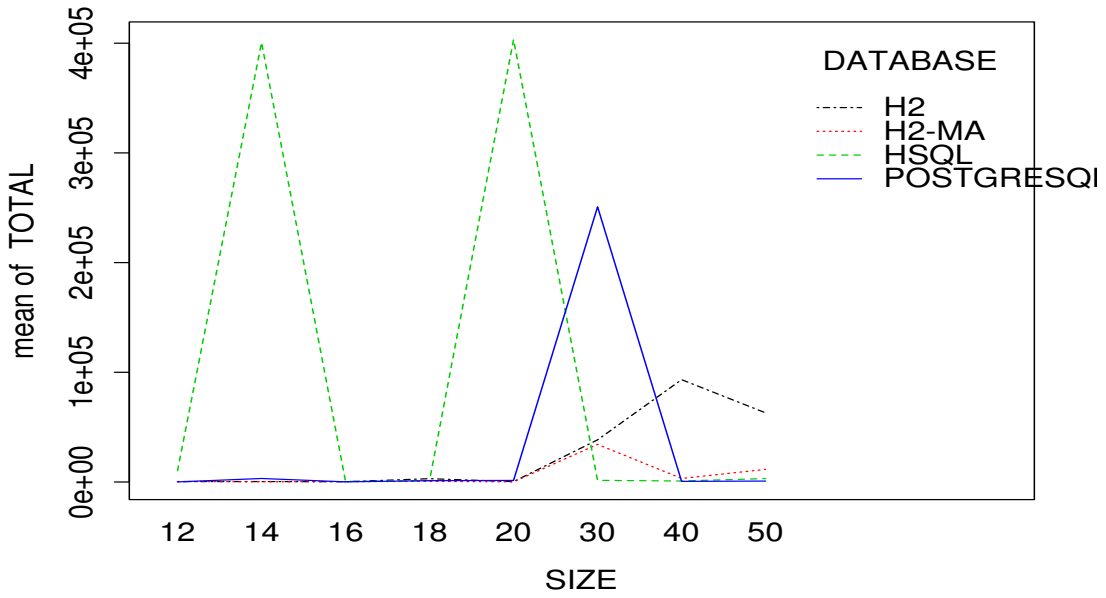


Figura 58 – Comportamento do Fator *DATABASE* sem o *Derby* - Tempo Total x Dimensão Problema.

Nota-se alguns picos na Figura 58, os quais foram determinantes para o aumento dos tempos de execução nos bancos relacionados. Analisando-se a versão MAQO, é possível localizar um aumento relevante do tempo nas dimensões 30 e 50. Uma posterior análise das consultas nessas dimensões, permitiu identificar uma anomalia do componente de custo do H2 relativa a incapacidade de inferir o custo real dos planos em algumas situações. Podem ser destacadas 3 consultas executadas pelo MAQO com tempos bem acima do H2QO e que tiveram o custo dos planos estimados como mais baratos. O tempo dessas consultas foi considerável, representando cerca de 28% do tempo total gasto no roteiro de testes. Tomando uma situação otimista, onde as consultas em questão fossem executadas com o mesmo tempo do H2QO, o tempo total gasto pelo MAQO no roteiro de testes cairia aproximadamente 16m, acentuando ainda mais a diferença de tempo entre as abordagens estudadas. Por fim, conforme o desempenho apresentado na Figura 58 e os tempos totais relacionados, em comparação ao otimizador oficial do H2, a versão MAQO apresentou tempos computacionais que a tornam especialmente interessante nos problemas envolvendo 18 ou mais relações.

Para quantificar o número de consultas executadas mais rapidamente por cada SGBDR, expõe-se na Tabela 19 um resumo com a porcentagem de consultas com

tempos de execução mais eficientes em comparações entre todos os pares de sistemas testados.

Tabela 19 – Resumo - Melhores Tempos

	MAQO	**H2QO**	**HSQL**	***Derby***	***PostgreSQL***
MAQO	-	58%	55%	96%	59%
H2QO	41%	-	48%	94%	45%
HSQL	45%	51%	-	91%	49%
Derby	4%	6%	9%	-	2%
PostgreSQL	41%	54%	50%	98%	-

Observa-se na Tabela 19 que a versão multiagente executou mais rapidamente que os outros sistemas em pelo menos 55% das consultas. Além disso, com base nos tempos computacionais apresentados, pode-se concluir que as consultas executadas mais eficientemente pelo MAQO representam problemas mais cruciais, implicando num dispêndio considerável de tempo pelos outros sistemas em tais consultas. Para encerrar a discussão, exibe-se por último, uma tabela quantificando as soluções com planos melhores e piores gerados pelo MAQO, quando relacionado ao algoritmo não-exaustivo oficial do H2.

Tabela 20 – Resumo - Custos dos Planos

Dimensão	**Custos Melhores**	**Custos Piores**
12	0%	10%
14	10%	0%
16	0%	22%
18	34%	0%
20	34%	0%
30	67%	0%
40	67%	0%
50	89%	0%

É válido notar que a distribuição dos planos com custos melhores segue a observação realizada anteriormente, a qual identificou um desempenho relevante para problemas com 18 ou mais relações no MAQO. Embora tenha apresentado planos um pouco piores nos problemas com 12 e 16 relações, a diferença no tempo de execução é ínfima, não passando de 1 segundo. Um outro ponto que merece destaque é o fato de o otimizador proposto gastar menos tempo de planejamento nos problemas com até 16 relações (situação verificada na Figura 56), o que levou a planos ligeiramente piores. Por outro lado, o planejamento mais rápido compensou o maior tempo na execução, uma vez que a diferença no tempo total gasto foi pequena.

6.5 Resumo

Este capítulo tratou da avaliação experimental realizada neste trabalho. Inicialmente, foi apresentada uma revisão sobre metodologias existentes para avaliação de algoritmos no problema de ordenação de junções. Também discorreu-se acerca da abordagem aplicada para a construção da base de testes e das consultas utilizadas para avaliar as metodologias testadas. Primeiro abordou-se o planejamento fatorial projetado para a seleção de alguns dos parâmetros do algoritmo. A seguir, empregou-se o *benchmark TPC-DS* para validar a capacidade do otimizador proposto em encontrar soluções ótimas. Por fim, comparou-se o algoritmo desenvolvido em relação a vários outros SGBDRs em vários formatos de consulta e em diferentes dimensões de problemas.

7 Conclusões e Trabalhos Futuros

A conclusão e a indicação de trabalhos futuros são tratados neste capítulo. Assim, as seções são organizadas como segue: a Seção 7.1 apresenta a conclusão final de todo o trabalho desenvolvido e a Seção 7.2 expõe algumas opções para trabalhos futuros.

7.1 Conclusão

Desenvolveu-se neste trabalho um algoritmo baseado em sistemas multiagente evolucionários aplicado ao problema de ordenação de junções. Foram definidas uma série de ações e perfis diferenciados buscando diferentes objetivos. O algoritmo proposto caracteriza-se por mesclar entre ações baseadas em operadores genéticos clássicos e outros métodos heurísticos. Implementou-se um conjunto de operadores de cruzamento e mutação, sendo os mesmos utilizados pelos agentes em ações relacionadas. Ressalta-se a proposição de um novo operador de cruzamento denominado PAGX. Como citado anteriormente, adicionalmente aos operadores genéticos, foram incluídas algumas ações baseadas em métodos de refinamento e construção semigulosa. Além disso, a infra-estrutura multiagente desenvolvida foi utilizada para a paralelização da heurística de refinamento descida completa.

O ambiente multiagente projetado pode ser caracterizado como acessível, pois os agentes podem obter informações precisas, atualizadas e completas sobre o ambiente (melhor solução corrente, aptidão da solução de outros agentes, etc). Adicionalmente, ele também pode ser considerado não-determinístico e dinâmico, pois não há certeza sobre o estado resultante de uma ação (ex.: um agente pode morrer antes de receber os pontos de vida solicitados a outro agente) e os agentes podem modificar o ambiente durante sua execução (ex.: podem atualizar a melhor solução corrente). Os agentes podem ser classificados como híbridos. As seguintes características dos agentes podem ser citadas: Reatividade, Proatividade, Habilidade Social, Veracidade e Benevolência.

A seção experimental foi subdividida em 3 experimentos distintos: o primeiro para calibrar os parâmetros do algoritmo proposto, o segundo para avaliar a eficiência do otimizador desenvolvido em encontrar soluções ótimas e o último para comparar o SGBDR H2 rodando o algoritmo multiagente com outros SGBDRs. É importante notar que os experimentos foram guiados por um planejamento fatorial de efeitos fixos, sendo o intervalo de confiança fixado em 95%, implicando em um nível de significância $\alpha = 0.05$. Os experimentos de calibração e de comparação com outros SGBDRs

empregaram na análise dos resultados uma metodologia estatística conhecida como teste de permutações. Já o experimento de validação aplicou o teste-t pareado na comparação das médias de amostras relativas às variáveis de saída coletadas. Por último, vale observar que todas as observações foram aleatorizadas e que nenhuma das premissas inerentes a adequação dos modelos gerados foi violada.

A calibração do método proposto avaliou a configuração dos seguintes parâmetros: o número máximo de agentes, tipo de movimento para a mutação, método de cruzamento, tipo de movimento para método de descida randômica, método da heurística semi-gulosa, coeficiente *LFC* da vida inicial do agente, coeficiente *DCL* para decremento de pontos de vida do agente e coeficiente *RDE* de esforço do método de descida randômica. Destaca-se que a escolha dos valores em cada parâmetro apoiou-se no comportamento do algoritmo relativo ao custo dos planos gerados e tempo de preparação do planos. De acordo com avaliação estatística realizada, apenas os fatores mutação e o coeficiente RDP não apresentaram significância na variabilidade dos dados e os seguintes parâmetros foram escolhidos: 8 agentes, cruzamento com 50% de chance para os métodos *PAGX* e *OX*, possibilidade de escolha igual a 50% para ambos os movimentos de troca e realocação na descida randômica e na mutação, versão *SCX* para a heurística semi-gulosa e os valores 0.3, 1 e 0.1 para os coeficientes *LFC*, *RDE* e *DCL*, respectivamente.

O experimento de validação utilizou o *benchmark TPC-DS* para a avaliar a capacidade do sistema multiagente proposto em encontrar ótimos globais em problemas envolvendo a junção de até 7 relações. Para tal, comparou-se o custo dos planos gerados pelo método exaustivo do H2 e o otimizador proposto (MAQO). Constatou-se que o MAQO foi capaz de encontrar todas as soluções ótimas e apresentou, em média, tempos menores que a abordagem exaustiva.

No último experimento, objetivou-se a avaliação do MAQO em problemas mais complexos, ou seja, na otimização de consultas com muitas junções (abrangem a junção de 10 ou mais relações). Assim, estudou-se problemas envolvendo a junção de 12, 14, 16, 18, 20, 30, 40 e 50 relações. O formato dos consultas-teste buscou representar problemas comuns em sistemas de apoio à tomada de decisões, extração de conhecimento e no campo de otimização RDF, portanto, optou-se pelos formatos: corrente, estrela e *snowflake*. Outrossim, cumpre salientar que na tentativa de se expressar problemas mais realistas, os valores adotados para os critérios de construção da base dados de teste foram baseados em distribuições reais de um banco de dados de produção do centro de TI de uma universidade brasileira. A avaliação realizada focou o tempo total gasto entre a preparação do plano e sua posterior execução por vários SGBDRs: H2 com o otimizador oficial (H2QO), H2 com o otimizador proposto (MAQO), HSQL, *Derby* e o popular *PostgreSQL*. Os resultados mostraram que o MAQO foi a

abordagem mais rápida. Ele foi o mais eficaz, sendo capaz de executar as consultas mais velozmente em pelo menos 55% dos casos. Em uma comparação com o algoritmo genético do *PostgreSQL*, verificou-se que a utilização do algoritmo multiagente proposto permitiu o SGBDR melhorar os tempos do H2 em aproximadamente 400%. Analogamente, observou-se uma melhoria geral dos tempos de execução do H2 quando utilizado o sistema multiagente evolucionário. Por fim, constatou-se que, em comparação ao H2QO, o MAQO apresentou tempos notavelmente menores para os problemas com 18 ou mais relações.

Face a superioridade da abordagem proposta sobre as outras metodologias testadas, nós mostramos o benefício do uso de sistemas multiagentes evolucionários na otimização de consultas envolvendo muitas junções. O conjunto de ações, os perfis definidos e o mecanismo de interação provaram ser uma ferramenta eficiente para explorar o espaço de soluções do problema em questão. Além disso, a utilização de um *benchmark* atestado pela comunidade científica (*TPC-DS*) permitiu a avaliação do algoritmo em problemas comprovadamente alinhados com a realidade, sendo que a abordagem proposta alcançou todos os ótimos globais. Por fim, consoante com o que foi dito, os resultados apresentados sugerem uma vantagem com o uso da metodologia multiagente evolucionária na melhoria dos tempos de execução do SGBDR H2 em ambientes menos carregados e com as consultas em grande parte caracterizadas por operações de leitura.

Finalmente, há que se ressaltar a validade do trabalho desenvolvido, visto que o mesmo produziu dois artigos, os quais foram aceitos em uma importante conferência e em um periódico com grande fator de impacto, respectivamente. Ao final do texto são disponibilizados os artigos citados. A Seção 7.2 seguinte sugere algumas áreas de pesquisa futuras.

7.2 Trabalhos Futuros

Como trabalhos futuros é válido citar uma extensão dos experimentos de forma a rodar com mais de um cliente. Este é um ponto interessante e poderia ser usado para explorar o otimizador proposto em um ambiente mais carregado, o qual pode representar muitas situações reais. Outro item importante é o estudo do otimizador em um ambiente distribuído. De acordo com a revisão apresentada no Capítulo 4, o campo de banco de dados distribuídos possui alguns trabalhos bem recentes relacionados ao problema foco deste trabalho. Consequentemente, o uso do otimizador para gerar planos em tais ambientes poderia ser outra contribuição. Mais ainda, sistemas multiagentes evolucionários são naturalmente indicados a problemas em ambientes distribuídos. Portanto, com a adição de novas ações e formas de interação, o otimi-

zador desenvolvido poderia gerar planos para tais ambientes. Por último, devido as similaridades do problema de ordenação de junções nos campos de SGBDR e RDF, uma área interessante de pesquisa diz respeito a integração do algoritmo proposto em algum motor de pesquisa RDF.

Referências

AHMED, Z. H. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics and Bioinformatics*, CSC Journals, Kuala Lumpur, Malaysia, Los Alamitos, CA, USA, v. 3, p. 96–105, 2010. ISSN 1985-2347. Citado na página 95.

AILAMAKI, A.; DEWITT, D. J.; HILL, M. D. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 11, n. 3, p. 198–215, nov. 2002. ISSN 1066-8888. Disponível em: <http://dx.doi.org/10.1007/s00778-002-0074-9>. Citado 2 vezes nas páginas 38 e 39.

AILAMAKI, A. et al. Dbmss on a modern processor: Where does time go? In: *Proceedings of the 25th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. (VLDB '99), p. 266–277. ISBN 1-55860-615-7. Disponível em: <http://dl.acm.org/citation.cfm?id=645925.671662>. Citado na página 31.

ANDERSON, M. J.; BRAAK, C. J. F. T. Permutation tests for multi-factorial analysis of variance. *Journal of Statistical Computation and Simulation*, v. 73, p. 85–113, 2003. Citado 2 vezes nas páginas 21 e 124.

ASTRAHAN, M. M. et al. System R: relational approach to database management. *ACM Transactions on Database Systems*, v. 1, p. 97–137, 1976. Citado na página 72.

BENNETT, K. P.; FERRIS, M. C.; IOANNIDIS, Y. E. A Genetic Algorithm for Database Query Optimization. In: *International Conference on Genetic Algorithms*. [S.l.: s.n.], 1991. p. 400–407. Citado 2 vezes nas páginas 19 e 76.

BRATBERGSENGEN, K. Hashing methods and relational algebra operations. In: *Proceedings of the 10th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1984. (VLDB '84), p. 323–333. ISBN 0-934613-16-8. Disponível em: <http://dl.acm.org/citation.cfm?id=645912.671296>. Citado na página 114.

CATTELL, R. Scalable sql and nosql data stores. *SIGMOD Record*, ACM, New York, NY, USA, v. 39, n. 4, p. 12–27, maio 2011. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/1978915.1978919>. Citado na página 16.

CHEN, Y. et al. Partial join order optimization in the paraccel analytic database. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. New York, NY, USA: ACM, 2009. (SIGMOD '09), p. 905–908. ISBN 978-1-60558-551-2. Disponível em: <http://doi.acm.org/10.1145/1559845.1559945>. Citado na página 118.

CODD, E. F. A relational model for large shared data banks. *Communications of The ACM*, v. 13, p. 377–387, 1970. Citado na página 16.

COPELAND, G. P.; KHOSHAFIAN, S. N. A decomposition storage model. *SIGMOD Record*, ACM, New York, NY, USA, v. 14, n. 4, p. 268–279, maio 1985. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/971699.318923>. Citado na página 38.

CORMEN, T. H. et al. *Introduction to algorithms*. [S.l.]: MIT press Cambridge, 2001. Citado 7 vezes nas páginas 18, 79, 81, 83, 88, 97 e 109.

DAVIS, L. Job shop scheduling with genetic algorithms. In: *Proceedings of the 1st International Conference on Genetic Algorithms*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1985. p. 136–140. ISBN 0-8058-0426-9. Disponível em: <http://dl.acm.org/citation.cfm?id=645511.657084>. Citado na página 95.

DONG, H.; LIANG, Y. Genetic algorithms for large join query optimization. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2007. (GECCO '07), p. 1211–1218. ISBN 978-1-59593-697-4. Disponível em: <http://doi.acm.org/10.1145/1276958.1277193>. Citado 4 vezes nas páginas 19, 82, 109 e 119.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, v. 26, n. 1, p. 29–41, Feb 1996. ISSN 1083-4419. Citado na página 86.

DREZEWSKI, R.; OBROCKI, K.; SIWIK, L. Agent-based co-operative co-evolutionary algorithms for multi-objective portfolio optimization. In: BRABAZON, A.; ONEILL, M.; MARINGER, D. (Ed.). *Natural Computing in Computational Finance*. Springer Berlin / Heidelberg, 2010, (Studies in Computational Intelligence, v. 293). p. 63–84. ISBN 978-3-642-13949-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-13950-5_5>. Citado 2 vezes nas páginas 19 e 92.

ELMASRI, R.; NAVATHE, S. *Fundamentals of Database Systems*. 6th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0136086209, 9780136086208. Citado 12 vezes nas páginas 16, 18, 22, 24, 28, 29, 30, 34, 35, 37, 111 e 122.

ELMASRI, R.; NAVATHE, S. Fundamentals of database systems. In: ____. 6th. ed. USA: Addison-Wesley Publishing Company, 2010. cap. 4 - Basic SQL. ISBN 0136086209, 9780136086208. Citado na página 26.

ELMASRI, R.; NAVATHE, S. Fundamentals of database systems. In: ____. 6th. ed. USA: Addison-Wesley Publishing Company, 2010. cap. 5 - More SQL: Complex Queries,Triggers, Views, and Schema Modification. ISBN 0136086209, 9780136086208. Citado na página 26.

ELMASRI, R.; NAVATHE, S. Fundamentals of database systems. In: ____. 6th. ed. USA: Addison-Wesley Publishing Company, 2010. cap. 3 - The Relational Data Model and SQL. ISBN 0136086209, 9780136086208. Citado na página 26.

ELMASRI, R.; NAVATHE, S. Fundamentals of database systems. In: ____. 6th. ed. USA: Addison-Wesley Publishing Company, 2010. cap. 6 - The Relational Algebra and Relational Calculus. ISBN 0136086209, 9780136086208. Citado na página 26.

FEIZI-DERAKHSHI, M.-R.; ASIL, H.; ASIL, A. Proposing a new method for query processing adaptation in database. *The Computing Research Repository*, abs/1001.3494, 2010. Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1001.html#abs-1001-3494>. Citado 2 vezes nas páginas 87 e 89.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. 7 - constraints and triggers. In: ____. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN 9780131873254. Citado na página 26.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. 8 - views and indexes. In: ____. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN 9780131873254. Citado na página 26.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. *Database Systems: The Complete Book*. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN 9780131873254. Citado 10 vezes nas páginas 15, 16, 25, 31, 33, 36, 37, 43, 44 e 47.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. Database systems: The complete book. In: ____. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. cap. 6 - The Database Language SQL. ISBN 9780131873254. Citado na página 26.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. Database systems: The complete book. In: ____. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. cap. 2 - The Relational Model of Data. ISBN 9780131873254. Citado na página 26.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. Database systems: The complete book. In: ____. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. cap. 5 - Algebraic and Logical Query Languages. ISBN 9780131873254. Citado na página 26.

GHAEMI, R. et al. Evolutionary query optimization for heterogeneous distributed database systems. *World Academy of Science*, v. 43, p. 43–49, 2008. Citado 2 vezes nas páginas 87 e 89.

GLOVER, F.; LAGUNA, M. *Tabu Search*. Boston: Kluwer Academic Publishers, 1997. Citado 2 vezes nas páginas 48 e 78.

GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675. Citado na página 49.

GOLSHANARA, L.; ROUHANI RANKOOHI, S.; SHAH-HOSSEINI, H. A multi-colony ant algorithm for optimizing join queries in distributed database systems. *Knowledge and Information Systems*, Springer London, v. 39, n. 1, p. 175–206, 2014. ISSN 0219-1377. Disponível em: <http://dx.doi.org/10.1007/s10115-012-0608-4>. Citado 2 vezes nas páginas 19 e 86.

GONÇALVES, F. A.; GUIMARÃES, F. G.; SOUZA, M. J. An evolutionary multi-agent system for database query optimization. In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2013. (GECCO '13), p. 535–542. ISBN 978-1-4503-1963-8. Disponível em: <http://doi.acm.org/10.1145/2463372.2465802>. Citado 2 vezes nas páginas 20 e 92.

GONÇALVES, F. A.; GUIMARÃES, F. G.; SOUZA, M. J. Query join ordering optimization with evolutionary multi-agent systems. *Expert Systems with Applications*, v. 41, n. 15, p. 6934 – 6944, 2014. ISSN 0957-4174. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0957417414002760>. Citado 2 vezes nas páginas 20 e 92.

GRAEFE, G. Efficient columnar storage in b-trees. *SIGMOD Record*, ACM, New York, NY, USA, v. 36, n. 1, p. 3–6, mar. 2007. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/1276301.1276302>. Citado na página 38.

GRAEFE, G.; DEWITT, D. J. The exodus optimizer generator. *SIGMOD Record*, ACM, New York, NY, USA, v. 16, n. 3, p. 160–172, dez. 1987. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/38714.38734>. Citado na página 78.

GRAEFE, G.; MCKENNA, W. J. The volcano optimizer generator: Extensibility and efficient search. In: *Proceedings of the Ninth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1993. p. 209–218. ISBN 0-8186-3570-3. Disponível em: <http://dl.acm.org/citation.cfm?id=645478.757691>. Citado na página 77.

GUTTOSKI, P. B.; SUNYE, M. S.; SILVA, F. Kruskal's algorithm for query tree optimization. In: *Proceedings of the 11th International Database Engineering and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2007. (IDEAS '07), p. 296–302. ISBN 0-7695-2947-X. Disponível em: <http://dx.doi.org/10.1109/IDEAS.2007.33>. Citado 4 vezes nas páginas 81, 90, 109 e 118.

HAN, W.-S. et al. Parallelizing query optimization. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 1, n. 1, p. 188–200, ago. 2008. ISSN 2150-8097. Disponível em: <http://dl.acm.org/citation.cfm?id=1453856.1453882>. Citado 3 vezes nas páginas 85, 89 e 109.

HEIMEL, M. Designing a database system for modern processing architectures. In: *Proceedings of the 2013 SIGMOD/PODS Ph.D. Symposium on PhD Symposium*. New York, NY, USA: ACM, 2013. (SIGMOD'13 PhD Symposium), p. 13–18. ISBN 978-1-4503-2155-6. Disponível em: <http://doi.acm.org/10.1145/2483574.2483577>. Citado na página 88.

HEIMEL, M.; MARKL, V. A first step towards gpu-assisted query optimization. In: CITESEER. *The Third International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures, Istanbul, Turkey*. [S.l.], 2012. p. 1–12. Citado na página 88.

HOGENBOOM, A.; FRASINCAR, F.; KAYMAK, U. Ant colony optimization for {RDF} chain queries for decision support. *Expert Systems with Applications*, v. 40, n. 5, p. 1555 – 1563, 2013. ISSN 0957-4174. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0957417412010500>. Citado 2 vezes nas páginas 87 e 122.

HOGENBOOM, A. et al. Rcq-ga: Rdf chain query optimization using genetic algorithms. In: NOIA, T.; BUCCAFURRI, F. (Ed.). *E-Commerce and Web Technologies*. Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5692). p. 181–192. ISBN 978-3-642-03963-8. Disponível em:

<http://dx.doi.org/10.1007/978-3-642-03964-5_18>. Citado 2 vezes nas páginas 87 e 122.

HOLLOWAY, A. L.; DEWITT, D. J. Read-optimized databases, in depth. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 1, n. 1, p. 502–513, ago. 2008. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1453856.1453912>. Citado 2 vezes nas páginas 16 e 121.

HUANG, M. D.; ROMEO, F.; SANGIOVANI-VINCENTELLI, A. An efficient general cooling schedule for simulated annealing. In: *International Conference on Computer Aided Design*. [S.l.: s.n.], 1986. Citado na página 74.

IBARAKI, T.; KAMEDA, T. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, ACM, v. 9, n. 3, p. 482–502, 1984. Citado 4 vezes nas páginas 18, 42, 72 e 73.

IOANNIDIS, Y. E. Query optimization. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 28, p. 121–123, March 1996. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/234313.234367>. Citado 3 vezes nas páginas 42, 43 e 44.

IOANNIDIS, Y. E.; KANG, Y. Randomized algorithms for optimizing large join queries. *SIGMOD Record*, ACM, New York, NY, USA, v. 19, n. 2, p. 312–321, maio 1990. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/93605.98740>. Citado 5 vezes nas páginas 75, 79, 85, 109 e 114.

IOANNIDIS, Y. E.; WONG, E. Query Optimization by Simulated Annealing. *SIGMOD Record*, v. 16, p. 9–22, 1987. Citado na página 73.

JARKE, M.; KOCH, J. Query optimization in database systems. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 16, n. 2, p. 111–152, jun. 1984. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/356924.356928>. Citado na página 109.

JOHNSON, D. S. et al. Optimization by simulated annealing: an experimental evaluation; part i. In: *Operations Research*. [S.l.: s.n.], 1989. Citado na página 74.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by Simulated Annealing. *Science*, v. 220, 4598, p. 671–680, 1983. Citado na página 48.

KITSUREGAWA, M.; TANAKA, H.; MOTO-OKA, T. Application of hash to data base machine and its architecture. *New Generation Computing*, Springer-Verlag, v. 1, n. 1, p. 63–74, 1983. ISSN 0288-3635. Disponível em: <http://dx.doi.org/10.1007/BF03037022>. Citado na página 37.

KLYNE, G.; CARROLL, J. J. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. 2004. Http://www.w3.org/TR/rdf-concepts/. Disponível em: <http://www.w3.org/TR/rdf-concepts/>. Citado 2 vezes nas páginas 16 e 45.

KNUTH, D. E. *The Art of Computer Programming, Volume III: Sorting and Searching*. [S.l.: s.n.], 1973. Citado na página 32.

KOSSMANN, D.; STOCKER, K. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 25, n. 1, p. 43–82, mar. 2000. ISSN 0362-5915. Disponível em: <http://doi.acm.org/10.1145/352958.352982>. Citado 3 vezes nas páginas 80, 117 e 120.

KRISHNAMURTHY, R.; BORAL, H.; ZANIOLO, C. Optimization of nonrecursive queries. In: *Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986. (VLDB '86), p. 128–137. ISBN 0-934613-18-4. Disponível em: <http://dl.acm.org/citation.cfm?id=645913.671481>. Citado 3 vezes nas páginas 72, 74 e 77.

LANGE, A. *Uma Avaliação de Algoritmos não Exaustivos para a Otimização de Junções*. Dissertação (Mestrado) — Universidade Federal do Paraná - UFPR, 2010. Citado 5 vezes nas páginas 85, 90, 109, 117 e 119.

LANZELOTTE, R. S. G.; VALDURIEZ, P.; ZAÏT, M. On the effectiveness of optimization search strategies for parallel execution spaces. In: *Proceedings of the 19th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. (VLDB '93), p. 493–504. ISBN 1-55860-152-X. Disponível em: <http://dl.acm.org/citation.cfm?id=645919.672650>. Citado na página 45.

LEE, C.; SHIH, C. sheng; CHEN, Y. huei. Optimizing Large Join Queries Using A Graph-Based Approach. *IEEE Transactions on Knowledge and Data Engineering*, v. 13, p. 298–315, 2001. Citado na página 109.

LIMITED, I. *Introduction to Database Systems*. Pearson Education, 2010. ISBN 9788131731925. Disponível em: <http://books.google.com.br/books?id=y7P9sa2MeGIC>. Citado na página 34.

MATYSIAK, M. Efficient optimization of large join queries using tabu search. *Information sciences*, Elsevier Science Inc., New York, NY, USA, v. 83, n. 1-2, p. 77–88, mar. 1995. ISSN 0020-0255. Disponível em: <http://dx.doi.org/10.1016/0020-0255(94)00094-R>. Citado na página 78.

MONTGOMERY, D. *Design and analysis of experiments*. Wiley, 2008. (Student solutions manual). ISBN 9780470128664. Disponível em: <http://books.google.com/books?id=kMMJAm5bD34C>. Citado 2 vezes nas páginas 74 e 113.

MULERO, V. M. *Genetic Optimization for large join queries*. Tese (Doutorado) — Universitat Politècnica de Catalunya, 2007. Citado 7 vezes nas páginas 19, 83, 90, 109, 117, 119 e 120.

MUNTES-MULERO, V.; ZUZARTE, C.; MARKL, V. An inside analysis of a genetic-programming based optimizer. In: *Proceedings of the 10th International Database Engineering and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2006. (IDEAS '06), p. 249–255. ISBN 0-7695-2577-6. Disponível em: <http://dx.doi.org/10.1109/IDEAS.2006.10>. Citado na página 85.

MUNTéS-MULERO, V. et al. Analyzing the genetic operations of an evolutionary query optimizer. In: BELL, D.; HONG, J. (Ed.). *Flexible and Efficient Information Handling*. Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4042). p. 240–244. ISBN 978-3-540-35969-2. Disponível em: <http://dx.doi.org/10.1007/11788911_21>. Citado na página 85.

MUNTéS-MULERO, V. et al. Cgo: A sound genetic optimizer for cyclic query graphs. In: ALEXANDROV, V. et al. (Ed.). *Computational Science ? ICCS 2006*. Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 3991). p. 156–163. ISBN 978-3-540-34379-0. Disponível em: <http://dx.doi.org/10.1007/11758501_25>. Citado na página 85.

MUNTéS-MULERO, V. et al. Intensive crossovers: Improving convergence and quality in a genetic query optimizer. In: SANTOS, J. C. R.; BOTELLA, P. (Ed.). *JISBD*. [s.n.], 2006. p. 131–140. ISBN 84-95999-99-4. Disponível em: <http://dblp.uni-trier.de/db/conf/jisbd/jisbd2006.html#Muntes-MuleroAZL06>. Citado na página 85.

MUNTéS-MULERO, V. et al. Improving quality and convergence of genetic query optimizers. In: KOTAGIRI, R. et al. (Ed.). *Advances in Databases: Concepts, Systems and Applications*. Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4443). p. 6–17. ISBN 978-3-540-71702-7. Disponível em: <http://dx.doi.org/10.1007/978-3-540-71703-4_3>. Citado na página 85.

MUNTéS-MULERO, V. et al. Parameterizing a genetic optimizer. In: BRESSAN, S.; KüNG, J.; WAGNER, R. (Ed.). *Database and Expert Systems Applications*. Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4080). p. 707–717. ISBN 978-3-540-37871-6. Disponível em: <http://dx.doi.org/10.1007/11827405_69>. Citado na página 85.

NAMBIAR, R. O.; POESS, M. The making of tpc-ds. In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB Endowment, 2006. (VLDB '06), p. 1049–1058. Disponível em: <http://dl.acm.org/citation.cfm?id=1182635.1164217>. Citado na página 112.

ÖZSU, T.; VALDURIEZ, P. *Principles of distributed database systems*. [S.l.]: Springer, 2011. ISBN 9781441988348. Citado 2 vezes nas páginas 21 e 44.

PIATETSKY-SHAPIRO, G.; CONNELL, C. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Record*, ACM, New York, NY, USA, v. 14, n. 2, p. 256–276, jun. 1984. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/971697.602294>. Citado na página 34.

POESS, M.; NAMBIAR, R. O.; WALRATH, D. Why you should run tpc-ds: A workload analysis. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 2007. (VLDB '07), p. 1138–1149. ISBN 978-1-59593-649-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1325851.1325979>. Citado 2 vezes nas páginas 111 e 112.

POOSALA, V. et al. Improved histograms for selectivity estimation of range predicates. *SIGMOD Record*, ACM, New York, NY, USA, v. 25, n. 2, p. 294–305, jun. 1996. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/235968.233342>. Citado 2 vezes nas páginas 33 e 34.

PRUD'HOMMEAUX, E.; SEABORNE, A. *SPARQL Query Language for RDF*. 2008. Latest version available as <http://www.w3.org/TR/rdf-sparql-query/>. Disponível em: <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>. Citado na página 45.

RHO, S.; MARCH, S. Optimizing distributed join queries: A genetic algorithm approach. *Annals of Operations Research*, Kluwer Academic Publishers, v. 71, n. 0, p. 199–228, 1997. ISSN 0254-5330. Disponível em: <http://dx.doi.org/10.1023/A%3A1018967414664>. Citado 3 vezes nas páginas 19, 86 e 87.

SADALAGE, P. J.; FOWLER, M. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. [S.l.]: Pearson Education, 2012. Citado na página 16.

SELINGER, P. G. et al. Access path selection in a relational database management system. In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1979. (SIGMOD '79), p. 23–34. ISBN 0-89791-001-X. Disponível em: <http://doi.acm.org/10.1145/582095.582099>. Citado 6 vezes nas páginas 34, 47, 72, 77, 80 e 88.

SEVINC, E.; COSAR, A. An evolutionary genetic algorithm for optimization of distributed database queries. *The Computer Journal*, Oxford University Press, Oxford, UK, v. 54, n. 5, p. 717–725, maio 2011. ISSN 0010-4620. Disponível em: <http://dx.doi.org/10.1093/comjnl/bxp130>. Citado 3 vezes nas páginas 19, 85 e 87.

SHAH, M. A. et al. Fast scans and joins using flash drives. In: *Proceedings of the 4th International Workshop on Data Management on New Hardware*. New York, NY, USA: ACM, 2008. (DaMoN '08), p. 17–24. ISBN 978-1-60558-184-2. Disponível em: <http://doi.acm.org/10.1145/1457150.1457154>. Citado na página 38.

SHAPIRO, L. D. et al. Exploiting upper and lower bounds in top-down query optimization. In: *Proceedings of the International Database Engineering & Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2001. (IDEAS '01), p. 20–33. ISBN 0-7695-1140-6. Disponível em: <http://dl.acm.org/citation.cfm?id=646290.686937>. Citado 3 vezes nas páginas 81, 116 e 117.

STEINBRUNN, M.; MOERKOTTE, G.; KEMPER, A. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 6, n. 3, p. 191–208, ago. 1997. ISSN 1066-8888. Disponível em: <http://dx.doi.org/10.1007/s007780050040>. Citado 4 vezes nas páginas 19, 79, 80 e 109.

STUCKENSCHMIDT, H. et al. Towards distributed processing of rdf path queries. *International Journal of Web Engineering and Technology*, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 2, n. 2/3, p. 207–230, dez. 2005. ISSN 1476-1289. Disponível em: <http://dx.doi.org/10.1504/IJWET.2005.008484>. Citado na página 87.

SWAMI, A. Optimization of large join queries: combining heuristics and combinatorial techniques. In: *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1989. (SIGMOD '89), p. 367–376. ISBN 0-89791-317-5. Disponível em: <http://doi.acm.org/10.1145/67544.66961>. Citado 5 vezes nas páginas 74, 79, 84, 109 e 113.

SWAMI, A.; GUPTA, A. Optimization of large join queries. *SIGMOD Record*, ACM, New York, NY, USA, v. 17, n. 3, p. 8–17, jun. 1988. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/971701.50203>. Citado 12 vezes nas páginas 42, 44, 46, 73, 74, 79, 83, 109, 112, 113, 114 e 115.

SWAMI, A.; IYER, B. A polynomial time algorithm for optimizing join queries. In: *Data Engineering, 1993. Proceedings. Ninth International Conference on*. [S.l.: s.n.], 1993. p. 345 –354. Citado na página 77.

SWAMI, A. N. A validated cost model for main memory databases. In: *SIGMETRICS*. [s.n.], 1989. p. 235. Disponível em: <http://dblp.uni-trier.de/db/conf/sigmetrics/sigmetrics89.html#Swami89>. Citado na página 113.

TSIROGIANNIS, D. et al. Query processing techniques for solid state drives. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2009. (SIGMOD '09), p. 59–72. ISBN 978-1-60558-551-2. Disponível em: <http://doi.acm.org/10.1145/1559845.1559854>. Citado 2 vezes nas páginas 38 e 39.

VANCE, B.; MAIER, D. Rapid bushy join-order optimization with cartesian products. *SIGMOD Record*, ACM, New York, NY, USA, v. 25, n. 2, p. 35–46, jun. 1996. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/235968.233317>. Citado 4 vezes nas páginas 79, 115, 116 e 117.

WHITLEY, D.; STARKWEATHER, T.; SHANER, D. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. 1990. Citado na página 97.

ZAFARANI, E. et al. Presenting a new method for optimizing join queries processing in heterogeneous distributed databases. In: *Knowledge Discovery and Data Mining, 2010. WKDD '10. Third International Conference on*. [S.l.: s.n.], 2010. p. 379–382. Citado 2 vezes nas páginas 87 e 89.

ZUKOWSKI, M.; NES, N.; BONCZ, P. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In: *Proceedings of the 4th International Workshop on Data Management on New Hardware*. New York, NY, USA: ACM, 2008. (DaMoN '08), p. 47–54. ISBN 978-1-60558-184-2. Disponível em: <http://doi.acm.org/10.1145/1457150.1457160>. Citado na página 38.

Anexos

ANEXO A – Periódico - *Experts Systems With Applications* - 2014

Expert Systems with Applications 41 (2014) 6934–6944

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Query join ordering optimization with evolutionary multi-agent systems

Frederico A.C.A. Gonçalves [a,c], Frederico G. Guimarães [a,*], Marcone J.F. Souza [b]

[a] Department of Electrical Engineering, Federal University of Minas Gerais, Belo Horizonte, Brazil
[b] Department of Computer Science, Federal University of Ouro Preto, Ouro Preto, Brazil
[c] IT Center, Federal University of Ouro Preto, Ouro Preto, Brazil

ARTICLE INFO

Keywords:
Join ordering problem
Query optimization
Multi-agent system
Evolutionary algorithm
Heuristics

ABSTRACT

This work presents an evolutionary multi-agent system applied to the query optimization phase of Relational Database Management Systems (RDBMS) in a non-distributed environment. The query optimization phase deals with a known problem called query join ordering, which has a direct impact on the performance of such systems. The proposed optimizer was programmed in the optimization core of the H2 Database Engine. The experimental section was designed according to a factorial design of fixed effects and the analysis based on the Permutations Test for an Analysis of Variance Design. The evaluation methodology is based on synthetic benchmarks and the tests are divided into three different experiments: calibration of the algorithm, validation with an exhaustive method and a general comparison with different database systems, namely Apache Derby, HSQLDB and PostgreSQL. The results show that the proposed evolutionary multi-agent system was able to generate solutions associated with lower cost plans and faster execution times in the majority of the cases.

1. Introduction

Database Management Systems (DBMS) are very complex software systems designed to define, manipulate, retrieve and manage data stored in a database. DBMS have an essential role in the information based society and represent critical components of business organization. Relational Database Management Systems (RDBMS) are those based on the relational model, which is the focus of this work. The information recovery depends on a database query and, as in Fig. 1, this query can be formed by many relations, which can be filtered and/or connected by different relational operators (Garcia-Molina, Ullman, & Widom, 2008) such as: selection ($\sigma_{\langle condition \rangle}$), projection ($\pi_{\langle attributes \rangle}$), intersection ($\bigcap$), union ($\bigcup$), set difference ($\setminus$), join ($\bowtie_{\langle condition \rangle}$) and others.

When processing a query, many steps can be executed by the RDBMS (Garcia-Molina et al., 2008; Elmasri & Navathe, 2010) until the delivery of a result: (i) scanning/parsing/validation, (ii) query optimization, (iii) query execution and (iv) result. The query optimization step, focus of this work, has a very important optimization task, which is ordering the relational operations of the query. Specifically in the case of the join operations, the most time-consuming operation in query processing (Elmasri & Navathe, 2010), the optimization task can be viewed as a combinatorial optimization problem commonly known as join ordering problem. The problem has similarities to the Traveling Salesman Problem (TSP) and according to Ibaraki and Kameda (1984), it belongs to the NP-Complete class. It is worth noting that after the optimization phase, the executor component receives a plan with all the instructions to its execution. Execution plans with relations ordered and accessed in a way that can cause high I/O and CPU cycles (high cost solutions) will impact directly in the query response time and affect the entire system. Besides, some costly plans can make the query execution impractical, because of their high execution times. Therefore, the use of techniques capable of finding good solutions in lower processing time is extremely important in a RDBMS. For instance, in one case reported in our experiments (Section 4) the proposed optimizer (Section 3.2) was capable to find a solution with a lower estimated cost, which allowed the plan to be executed almost 23% faster than the solution provided by the official optimizer used in H2 in the same experiment.

In this paper we propose an approach to query optimization that can be classified as a non-exhaustive one. We describe an evolutionary multi-agent system (EMAS) (see Section 3.1) for join ordering optimization running in the core of a real RDBMS named H2[1] in a non-distributed environment. The main feature of the

* Corresponding author. Tel.: +55 (31) 3409 3419.
E-mail addresses: fred@nti.ufop.br (F.A.C.A. Gonçalves), fredericoguimaraes@ufmg.br (F.G. Guimarães), marcone@iceb.ufop.br (M.J.F. Souza).

[1] http://www.h2database.com/.

http://dx.doi.org/10.1016/j.eswa.2014.05.005
0957-4174/

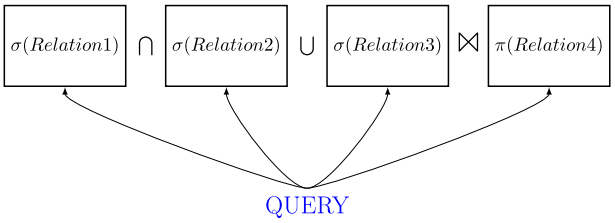


Fig. 1. Example query.

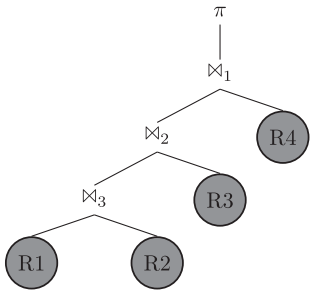


Fig. 2. Left-deep tree.

algorithm is having a team of intelligent agents working together in a cooperative or competitive way to achieve the solution of the problem. The agents of the system are able to interact and evolve in parallel. We extend initial ideas presented in Gonçalves, Guimarães, and Souza (2013) by including the following contributions: improvements in the algorithm operators; parallel implementation of the local search heuristics; evaluation of the parameters of the algorithm in the calibration phase; extended experiments with more realistic data; and comparison of the proposed algorithm with the official query planner in H2 and other DBMS, namely HSQLDB, Derby and PostgreSQL.

We highlight as the main contribution of this paper, the development of a technique not yet explored in the join ordering optimization field. Such method solves the join ordering problem in a parallel way, and as will be reviewed in Section 2, most of the proposed algorithms in the literature process the related problem sequentially. Still regarding the proposed algorithm, a new crossover method can be highlighted. The experiment design is validated with a real Database Management System, and consequently, a real cost model. Finally, we emphasize a more realistic evaluation methodology of the algorithms.

The paper is organized as follows. Section 2 discusses in more detail the query optimization problem and some methodologies applied to solve it. The proposed optimizer is detailed in Section 3. The evaluation methodology is introduced in Section 4. The computational experiments and the conclusions/future work are presented in Sections 5 and 6, respectively.

2. Query optimization problem

The optimization problem of this work consists in defining the best order of execution of the join operations among the relations specified in the query to be processed. According to Ioannidis (1996), the solution space can be divided into two modules: the algebraic space and the space of structures and methods. The algebraic space is the focus of the discussion in this section, because refers to the execution order of the relational operations considered. The space of structures and methods, on the other hand, is related to the available methods for relational operators in the RDBMS (more information about these methods can be found in Garcia-Molina et al. (2008) and Elmasri & Navathe (2010)).

The final result of the optimizer is a plan with all the necessary instructions to the query execution. Besides the operations order, a query can be represented by many different shapes. Assume that, for example, a join operator for 4 relations (*multi-way join*) is available and then a query with 4 relations could be expressed by only one join operation. However, in practice, the join operation is binary (*two-way join*) because the combinations for multi-way joins grow very rapidly (Elmasri & Navathe, 2010) and there are mature and proven efficient implementations for binary join in the literature. A representation commonly used by RDBMS is called left-deep-tree (see Fig. 2). This representation has only relations at the leaves and the internal nodes are relational operations. Due to this representation, the join operation is treated as binary (join two tables only). Even with these restrictions, the number of possible solutions remains high – for a query with $N+1$ relations the number of solutions is given by $\binom{2N}{N}N!$. Further information about the join ordering problem can be found in Ibaraki and Kameda (1984), Swami and Gupta (1988), Ioannidis (1996) and Steinbrunn, Moerkotte, and Kemper (1997).

It is worth noting that the cost of a solution is not given necessarily by the actual cost of executing the query, but instead, can be given by a cost function F that estimates the real cost of the solution. The cost estimation can use many metrics, for instance: I/O, CPU and Network. The optimizer relies on the RDBMS for the task of estimating the cost of the solution. Therefore, in the optimization process, this function is abstracted and treated as a black box, just receiving a candidate solution and returning its (estimated) cost. Information about the cost model of relational operations are provided by Garcia-Molina et al. (2008) and Elmasri and Navathe (2010).

Listing 1 presents a practical example with a query that returns all marks from computer science students.

Two join operations can be extracted from the previous query: $J_1 = \{student \bowtie marks\}$ and $J_2 = \{student \bowtie dept\}$. Besides, two valid solutions can be identified in this simple example: $S_1 = \{J_1, J_2\}$ and $S_2 = \{J_2, J_1\}$, one with a lower cost than the other. In practice, the problem can have a huge combination of possible solutions, preventing the use of exact methods or exhaustive approaches. Nonetheless, non-exhaustive algorithms fit well in these situations.

Given the complexity of the problem, several studies were presented for non-distributed environments along the years since the early days of relational databases in the 1970s (Codd, 1970). The seminal work is presented by Selinger, Astrahan, Chamberlin, Lorie, and Price (1979), advancing an exhaustive method based on dynamic programming (*DP*) with a time complexity $O(N!)$, where N stands for the number of relations in the query. Ibaraki and Kameda (1984) presented two algorithms, named *A* and *B*, with a time complexity $O(N^3)$ and $O(N^2 \log N)$, respectively. An extension of Algorithm *B* named *KBZ* algorithm with time complexity $O(N^2)$ is presented by Krishnamurthy, Boral, and Zaniolo (1986). A simulated annealing (*SA*) version for the current problem is presented in Ioannidis and Wong (1987). Many methods are compared in Swami and Gupta (1988): Perturbation Walk, Quasi-random Sampling, Iterated Improvement (*II*), Sequence Heuristic and SA. Additionally, in Swami and Gupta (1988), the authors created a new evaluation methodology to check the heuristics,

```
SELECT name, disc, mark FROM student,
    marks, dept WHERE id_student=id_marks
    AND dpt_student=id_dept AND id_dept='
    DECOM'
```

Listing 1. Example – query SQL.

considering cardinality, selectivity of the join predicates, distinct values and indexes. Based on the methods *II* and *SA*, an algorithm called *2PO* has been proposed in Ioannidis and Kang (1990), which combines the two techniques mentioned. A genetic algorithm (GA) applied to the join ordering problem is presented in Bennett, Ferris, and Ioannidis (1991) and compared against the *DP* (Selinger et al., 1979). The authors in Swami and Iyer (1993) proposed an extension of the algorithm (*AB*) with some improvements and a time complexity $O(N^4)$. The meta-heuristic Tabu Search (*TS*) (Glover & Laguna, 1997) is explored for this problem in Matysiak (1995). An algorithm named *blitzsplit* is presented by Vance and Maier (1996). Another comparison between many algorithms is presented in Steinbrunn et al. (1997). SA and GA algorithms were applied again in Lee, sheng Shih, and huei Chen (2001) and Dong and Liang (2007), respectively. In the work by Guttoski, Sunye, and Silva (2007), an implementation of the *kruskal* algorithm is described. A parallel based *DP* algorithm is discussed in Han, Kwak, Lee, Lohman, and Markl (2008). Many methods in the literature are reviewed by Lange (2010) in his Master thesis.

More recently, the GPU technology in database systems was addressed by Heimel and Markl (2012) and Heimel (2013). They discuss about some initial ideas for the design of a GPU-assisted query optimizer, with plans to implement a GPU-accelerated version of the DP algorithm (Selinger et al., 1979). The generation and use of hybrid query plans, i.e., the optimization and execution of queries with plans that mix the use of CPU and GPU processors is discussed in Breß, Schallehn, and Geist (2013). Regarding distributed databases, the authors in Sevinç and Coşar (2011) have proposed a new GA version (NGA) and compared it against a previous GA method (Rho & March, 1997), exhaustive and random algorithms. According to the authors, the NGA was capable to find optimal results in 80% of cases and improved the results over previous GA in 50%. A review about some algorithms applied to query optimization in distributed database systems can be found in Tewari (2013). Lastly, in Golshanara, Rouhani Rankoohi, and Shah-Hosseini (2014), an ant-colony optimizer (ACO) (Dorigo, Maniezzo, & Colorni, 1996) is proposed to order the join operations of an environment with the data replicated across multiple database sites. The ACO is compared with other algorithms, among them, a GA method (Sevinç & Coşar, 2011). According to the results, the ACO reduced the optimization time in about 80% without lost quality of the solutions.

The first use of GA and multi-agent systems (MAS) for query optimization in distributed DBMS is proposed in Ghaemi, Fard, Tabatabaee, and Sadeghizadeh (2008). They define the following agents: Query Distributor Agent (QDA) to divide the query into sub-queries, Local Optimizer Agents (LOAs) that applies a local genetic algorithm and Global Optimizer Agent (GOA) responsible for finding the best join order between the sites. In a comparison with a dynamic programming method, the authors verified that their approach takes less time for processing the queries. An extension of Ghaemi et al. (2008) with focus on building an adaptive system is given by Zafarani, Derakhshi, Asil, and Asil (2010) and Feizi-Derakhshi, Asil, and Asil (2010). The results have shown a reduction of up to 29% in time response. It is worth noting the distinctions between the present work and the work from Ghaemi et al. (2008), Zafarani et al. (2010) and Feizi-Derakhshi et al. (2010). First, their methods are supposed to run in a distributed environment; secondly, they are running outside of the DBMS; and lastly, the agents have a limited and different way of interaction. Basically, one agent (QDA) breaks the query into pieces and distributes part of the analysis of relations of some data source with a registered agent (LOA) that will execute a standalone version of genetic algorithm to find a possible order to join the associated relations. Finally, the last agent (GOA) will try to minimize the network traffic, by executing the partial plan defined by the related LOA, sending the partial result across the network to another data source, and joining it with other partial results of plans defined by other LOA. The process terminates when all partial results are joined.

The join ordering problem is not restricted only to traditional database systems, in the Resource Description Framework (RDF) field (Klyne & Carroll, 2014), a typical scenario is composed by many possible interconnected heterogeneous sources of data distributed over network. The join ordering problem arises when a RDF query (Prud'hommeaux & Seaborne, 2008) needs to join many sources of data. The join operation in this context is similar to the relational databases and has impact on the response time. Hogenboom, Frasincar, and Kaymak (2013) have compared an ACO approach against an 2PO (Stuckenschmidt, Vdovjak, Broekstra, & Houben, 2005) and GA (Hogenboom, Milea, Frasincar, & Kaymak, 2009) methods in the join ordering of the sources. The results of ACO have shown lower execution times and better solutions quality for queries consisting of up to 15 joins. For larger problems the GA performed better.

In relation to some RDBMS in the market, *H2*[2] uses a brute force method for queries with up to 7 relations and a mixed algorithm composed by an exhaustive, greedy and random search methods is applied to queries with more than 7 relations. The PostgreSQL[3] has an optimizer based on DP (Selinger et al., 1979) for ordination of queries with up to 11 relations and a GA for queries with more than 11 relations. The RDBMS MySQL[4] and Apache Derby[5] employ a depth-first search based algorithm.

3. Query optimizer

This section discusses the proposed optimizer applied to the join ordering problem presented previously. Section 3.1 introduces concepts about evolutionary multi-agent systems (EMAS) and some of their applications. The proposed evolutionary multi-agent optimizer is explained in Section 3.2.

3.1. Evolutionary multi-agent systems – EMAS

In this subsection we provide an overview of the use of multi-agent systems and evolutionary algorithms in optimization. Multi-agent systems can be defined as systems involving teams of autonomous agents working together in a cooperative or competitive way to achieve the solution of a given problem. Such systems differ from purely parallel systems, because of the distinctive interaction between the agents. The related agents have specific characteristics such as reactivity, proactivity, sociability and so on (Wooldridge, 2009). Evolutionary algorithms, on the other hand, work with a population of candidate solutions and this population evolves iteratively by means of heuristic operators inspired or motivated by concepts of natural systems and Darwinian principles. In a typical evolutionary algorithm, the fitness of the individuals depends only on the quality of that individual in solving the problem.

More recently, some researchers have explored the integration between evolutionary algorithms and multi-agent systems, trying to take the best from both worlds. The integration between multi-agent systems and evolutionary algorithms lead to the so-called evolutionary multi-agent systems – EMAS ('t Hoen et al., 2004; Hanna & Cagan, 2009; Barbati, Bruno, & Genovese, 2012).

[2] H2 – Version 1.3.174 – 2013 www.h2database.com/html/performance.html.
[3] PostgreSQL – Version 9.2.6 – 2013 www.postgresql.org/docs/9.2/interactive/geqo.html.
[4] MySQL – Version 5.1.72 – 2013 http://dev.mysql.com/doc/internals/en/optimizer.html.
[5] Derby – Version 10.9.1.0 – 2012 http://db.apache.org/derby/papers/optimizer.html.

In such systems, the agents have also the ability to evolve, reproduce and adapt to the environment, competing for resources, communicating with other agents, and making autonomous decisions (Drezewski, Obrocki, & Siwik, 2010).

Evolutionary multi-agent systems have been applied in different contexts, among them: decision making (Dahal, Almejalli, & Hossain, 2013; Khosravifar et al., 2013), multi-agent learning (Enembreck & Barthès, 2013; Van Moffaert et al., 2014; Li, Ding, & Liu, 2014), multi-objective optimization (Drezewski et al., 2010; Tao, Laili, Zhang, Zhang, & Nee, 2014).

3.2. Proposed optimizer

In this work we present a method based on evolutionary algorithms and multi-agent systems. It employs techniques inspired by evolutionary models and is composed by a set of agents that work together to achieve the best possible solution for the problem, i.e., the best join order for the relations in the query. Such methodology extends the default behavior of genetic algorithms and multi-agent systems, since the agents can act pro-actively and reactively. They can adopt specific interaction mechanisms and explore the solution space in a smart way by applying genetic operators and constructive heuristics. Drezewski et al. (2010) cites two different evolution mechanisms: mutual selection and host-parasite interaction. The mutual selection was the technique adopted in this work, in which each agent chooses individually another agent and executes the available and related actions.

The proposed evolutionary multi-agent system can be described as $EMAS = \langle E, \Gamma, \Omega \rangle$, where E is the environment, Γ is the system resources set and Ω is the information available to the agents in the system.

The environment is non-deterministic and dynamic, there is no certainty about the results of an agent action and agents can change the environment. The environment definition is given by $E = \langle T^E, \Gamma^E, \Omega^E \rangle$, wherein T^E represents the environment topography, Γ^E the resources of the environment and Ω^E the information available to the agents. We use only one kind of resource and information. The resource is expressed in terms of life points of each agent. The information about the other agents and the best current solution corresponds to Ω^E.

The topography is represented by the notation $T^E = \langle A, l \rangle$, where A is the set of agents (population) in the environment and l is a function that allows to locate a specific agent. Each agent $a \in A$ can be denoted by the expression $a = \langle sol^a, Z^a, \Gamma^a, PR^a \rangle$, sol^a stands for the solution represented by the agent, which is an integer array and each element of this array represents a relation id. This array also defines the execution order to join the relations. The actions set of the agent is given by Z^a and its life points by Γ^a. The current goal of the agent is defined by its current profile PR^a. The initial life of all agents is expressed in terms of $LIFE = IL \times NR$, where IL is the initial life coefficient and NR is the number of relations in the query. All actions available in the environment are listed as follows:

- ***getLife*** (***gl***) – Used to obtain life points from another agent;
- ***giveLife*** (***vl***) – Give life points to another agent;
- ***lookWorse*** (***lw***) – Search for an agent with worse fitness value;
- ***lookPartner*** (***lp***) – Search for a partner in the set of agents in the environment;
- ***crossover*** (***cr***) – Crossover operator used to generate offspring;
- ***mutation*** (***mt***) – Mutation operator;
- ***becomeBestSol*** (***bb***) – Update its solution with the best current solution;
- ***updateBestSol*** (***ub***) – Update the best solution;
- ***randomDescent*** (***rd***) – Random descent local search method;
- ***parallelCompleteDescent*** (***pd***) – Parallel Best Improvement local search method;
- ***semiGreedyBuild*** (***gb***) – Construct a solution by using a semi-greedy heuristic;
- ***processRequests*** (***pr***) – Evaluate and process pending requests;
- ***tryChangeProfile*** (***cp***) – Tries to change the profile of the agent;
- ***stop*** (***st***) – Stops momentarily its execution;
- ***die*** (***id***) – Action of dying, when the life points of the agent ends.

Each agent is associated to an execution thread. These agents can be classified as hybrid ones, since they are reactive with internal state and deliberative agents seeking to update the best solution and not to die. The mutation and local search methods employ swap and reallocation movements (Fig. 3).

By default, the mutation can apply in each execution at most 5 movements (swap or reallocation) and the random descent method has a maximum number of iterations without improvement equal to $RD_{mov} = RDE \times IL$, where RDE is a coefficient of effort and IL the initial life of an agent.

Three crossover operators were implemented: *Ordered Crossover* – OX (Davis, 1985), *Sequential Constructive Crossover* – SCX (Ahmed, 2010) and a new operator proposed in this work named *Pandora-Aquiles Greedy Crossover* – PAGX. The examples described next use as input the information of the Fig. 4.

In the crossover OX, two descendants can be generated. First a crossover point is selected at random, which defines the part of one of the agents will go to the descendant. The rest of the sequence is taken from the other agent in an ordered way avoiding repetition of elements. Considering the agents in Fig. 4 and a crossover point comprising the 1st e 2nd relations of Agent 1, one of the resulting descendants is presented in Fig. 5.

The strategy SCX starts by adding the first relation R_{CUR} of the Agent 1 in the resulting descendant. After that, the next relation R_{NEXT} subsequent to R_{CUR} from one of the parents, not present in the descendant and with lower join cost $R_{CUR} \bowtie R_{NEXT}$ is chosen. Then R_{NEXT} becomes R_{CUR} and the process continues until all relations are added in the descendant as in Fig. 6. The cost of the join operations between each pair of relations is stored in a cost matrix Fig. 4(b) and ties are solved randomly.

The PAGX method mixes randomness and determinism. Initially a crossover point is chosen as in OX. However, the resulting descendant inherits the partial solution with lower cost from one of the parents or from the first agent in the case of a tie. After that, a list L with the remaining relations is created and in each iteration, the relation with lower join cost is taken from L or the first element in L if a tie occurs. The process ends when L becomes empty. An example result considering a cut point with the 2nd e 3rd relations and the agents from Fig. 4(a) is presented in Fig. 7.

The action semiGreedyBuild is based on a semi-greedy (greedy-random) heuristic. At each iteration, a list CL containing all relations not present in the current solution is sorted according

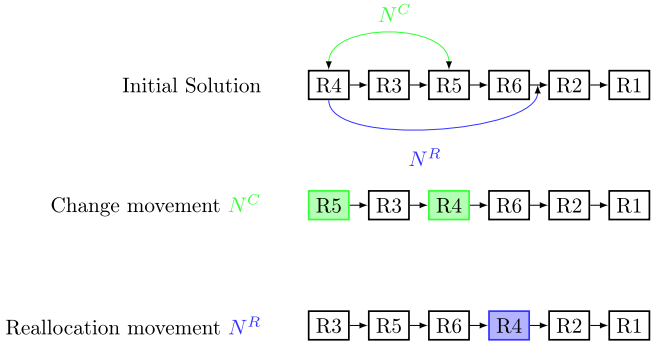
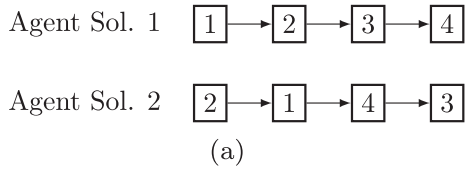


Fig. 3. Movement types.



(a)

Join Cost Matrix				
	R1	R2	R3	R4
R1	99	10	20	5
R2	10	99	15	35
R3	20	15	99	45
R4	5	35	45	99

(b)

Fig. 4. Agents informations.

to a greedy function or classification criterion. The best ranked elements from *CL* form the restricted candidates list *RCL*. A new relation is taken randomly from the *RCL* and inserted in the current solution. The method stops when all relations have been inserted. In this work, we adopt two different classification criteria to *CL*: the *ERX* criterion that takes into consideration the number of links between each relation and relations with less links are better classified and the *SCX* criterion that is based on the cost of links or on the join cost of the relations, where relations with lower join costs are better classified.

Lastly, the best current solution is refined by the parallel best improvement method (*parallelCompleteDescent*) and a local optimum according to the neighborhood N^C or N^R (selected at random) will be returned. The method has a task list *TL* shared by the agents, which distributes all the neighborhood analysis into distinct tasks. Thus, each task evaluated by the agents will be taken from *TL*. The process ends when all tasks have been evaluated. It is worth noting that each task will be analyzed by only one agent and no more than one time. The method returns the best solution of one of the agents.

Each agent can execute different goals according to specific profiles, they can generate a solution together with another agent, refine its own solution and so on. The designed profiles are:

- **RESOURCE** – The agent tries to increase his life points by searching agents with worse fitness values and requesting part of their life points;
- **REPRODUCTION** – The agent looks for partners to produce new solutions by crossover;
- **MUTANT** – The agent suffers mutation and tries to explore other parts of the solution space;
- **RANDOM_DESCENT** – The agent refines its own solution by using random descent local search;
- **SEMI-GREEDY** – The agent builds new solutions by using the semi-greedy heuristic.

Regarding the defined profiles, it is worth mentioning that to avoid high randomness and high effort of refinement, the probabilities of choosing the MUTANT and RANDOM_DESCENT profiles were set as 10% for each one. On the other hand, to a smart and effective exploration of the solution space, the REPRODUCTION and SEMI-GREEDY profile have received a probability equal to 60% AND 15%, respectively. Finally, in order to impose some selective pressure to the algorithm, the probability of RESOURCE profile was set as 5%. The set of actions of each profile is distributed as follows:

- **RESOURCE** – lookWorse, processRequests, getLife, tryChangeProfile, stop and die;
- **REPRODUCTION** – lookPartner, processRequests, crossover, updateBestSol, tryChangeProfile, stop and die;
- **MUTANT** – mutation, processRequests, updateBestSol, tryChangeProfile, stop and die;
- **RANDOM_DESCENT** – becomeBestSol, randomDescent, processRequests, updateBestSol, tryChangeProfile, stop and die;
- **SEMI-GREEDY** – semiGreedyBuild, processRequests, updateBestSol, tryChangeProfile, stop and die.

The pseudo-code of the Evolutionary Multi-Agent Query Optimizer (MAQO) is presented in Algorithm 1. This algorithm first initializes the system (line 2). Then, all agents are created with the profile *REPRODUCER*. One of them (line 4) is started with an initial solution in the order in which the relations are read during the initial parsing of the query and the other ones are filled with a solution generated by the semi-greedy heuristic. The initialization and waiting of the agents and final parallel refinement of the best current solution are done in lines 9, 11 and 20.

Algorithm 1. MAQO

```
   Data: QueryRelations
   Result: sol*
 1 begin
 2     Environment E ← initializeEnvironment(QueryRelations);
 3     //Create the agents with the profile REPRODUCTION;
 4     createAgentWithInitialSolution(E, REPRODUCTION);
 5     for i=2 to E →MAXAGENTS do
 6         createAgentWithSemiGreedySolution(E, REPRODUCTION);
 7     end
 8     //Initialize threads;
 9     initializeAgents(E);
10     //Wait threads finish;
11     waitAgents(E);
12     //All agents update their solutions;
13     for i=1 to E →MAXAGENTS do
14         E →findAgent(i)→becomeBestSol();
15     end
16     //Generate the tasks that will be executed by the agents;
17     List TASKS ← generateTasks();
18     //Parallel Best Improvement in the best current solution;
19     for i=1 to E →MAXAGENTS do
20         E →findAgent(i)→parallelCompleteDescent(TASKS);
21     end
22     return E →bestSolution;
23 end
```

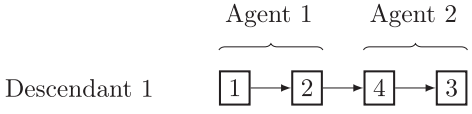


Fig. 5. Descendant OX.

Once the agents have been initialized, the evolutionary process will also start, since agents may evolve through the execution of the associated actions, which can be individual or related to

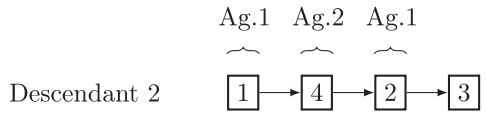


Fig. 6. Descendant SCX.

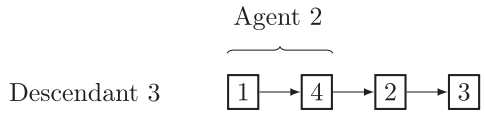


Fig. 7. Descendant PAGX.

another agent. At every iteration of the agent in a given profile, its life points decrease. The agents can change their profiles only after a minimal number of iterations in the current profile. They will always change to *RESOURCE* profile when the life level reaches 10% of the initial life. Those agents with no more life points execute the action *stop*. When all agents stop, the optimizer executes the parallel best improvement action and then finally all agents die (action *die*) and the algorithm terminates its execution.

4. Evaluation methodology

The evaluation methodology of the problem (Section 2) in this work involves the optimization and execution of a group of queries in SQL. Such process can be based on synthetic benchmarks built according to some criteria as done in Swami and Gupta (1988), Swami (1989), Vance and Maier (1996), Steinbrunn et al. (1997), Brunie and Kosch (1997), Lee et al. (2001), Shapiro et al. (2001), Dong and Liang (2007), Lange (2010) or on those benchmarks standardized by the industry, such as the TPC-DS (decision support benchmark) from TPC (*Transaction Processing Performance Council*).[6] The TPC-DS as well as other benchmarks from TPC were discarded in our experiments, because our minimal number of relations in the queries starts with 12, and for example the TPC-DS has only 7 out of 99 queries with 8 or more relations.

The methodology proposed in this work is based mainly on the ideas presented by Swami and Gupta (1988) and Swami (1989). We randomly create the test database and the test queries according to some criteria. In an attempt to better approximate real problems, some of the database construction criteria were based on observed distributions in a production database from the IT Center of a Brazilian University. The evaluation methodology is divided in two stages, one builds the database and another builds the queries. To create and load the database, the following criteria were used:

- **Cardinality distribution of tuples** – $[10, 100)$ – 26%, $[100, 1000)$ – 27%, $[1000, 1000000]$ – 47%;
- **Distribution of distinct values of the tuples** – $[0, 0.2)$ – 30%, $[0.2, 1)$ – 16%, 1 – 54%;
- **Relations Columns** – three per relation, being one reserved for primary key;
- **Probability of a column to have indexes and foreign keys** – 90% and 20%, respectively.

The total number of relations created and loaded according to the previous distributions was set as 60. The cardinality and the number of distinct values of the columns were selected at random from the given intervals. The evaluation of complex queries with many relations fits well in decision support problems and data mining (Brunie & Kosch, 1997; Dong & Liang, 2007). There are problems in the literature with up 100 relations. The number of relations presented in the queries was fixed in: 12, 16, 20, 30, 40 and 50 relations. Besides, the generation of the test queries was guided by three different graph shapes: chain, star and snowflake.

The generation of a query in a specific shape, first selects at random the given number of relations from the 60 available. After that, the join predicates are formed by randomly selecting the columns from both relations. It is important to note that all projected columns are the same of the join clauses and the predicates always will be generated according to the graph shape. Besides, multiple seeds can be used to generate different queries and an example of a query built with shape chain and with the relations R_1, R_7, R_2, R_{10}, R_{60} and R_{50} is presented in Listing 2.

[6] TPC: http://www.tpc.org.

$$\begin{aligned}&\textbf{SELECT } * \textbf{ FROM } R_1, R_7, R_2, R_{10}, R_{60}, R_{50} \textbf{ WHERE}\\ &\quad R_1.col_1 = R_7.col_2 \textbf{ AND } R_7.col_3 = R_2.col_2 \textbf{ AND}\\ &\quad R_2.col_1 = R_{10}.col_3 \textbf{ AND } R_{10}.col_1 = R_{60}.col_3 \textbf{ AND}\\ &\quad R_{60}.col_2 = R_{50}.col_1\end{aligned}$$

Listing 2. Example – SQL chain scheme.

Table 1
Factors and levels – calibration experiment.

Factor	Level
NA	8, 16
IL	1, 1.5
DL	0.03, 0.05, 0.1
RDE	0.3, 0.5
MM	Change, Reallocation
RDM	Change, Reallocation
CM	PAGX, SCX, OX
SM	ERX, SCX

Table 2
Significance of the factors and their interactions.

Factor	Significance		Interaction Signif.	
	Cost	Setup	Cost	Setup
NA	Ye	Yes	MM	MM, SM, RDM,IL and DL
IL	Yes	Yes	–	DL
DL	Yes	Yes	–	–
RDE	No	No	–	–
MM	No	No	NA	NA
RDM	Yes	Yes	DL	IL, DL
CM	Yes	Yes	SM, RDM, RDE, IL	RDM, IL
SM	Yes	Yes	RDM, IL	DL

Table 3
ANOVA – efficiency comparison.

Source	DF	SS	MS	Iter	Pr
DB	1	7e+28	7e+28	51	0.961
SHAPE	1	1e+33	1e+33	1009	0.090·
DB:SHAPE	1	2e+26	2e+26	51	0.961
SIZE	1	2e+34	2e+34	5000	<2e−16***
DB:SIZE	1	1e+29	1e+29	51	0.725
SHP:SIZE	1	2e+33	2e+33	2906	0.033*
DB:SHAPE:SIZE	1	3e+26	3e+26	51	0.863
Residuals	16	8e+33	5e+32		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '·' 0.1 ' ' 1.

5. Experiments

The experimental section is divided in three subsections: the first (5.1) for the calibration of the algorithm parameters, the

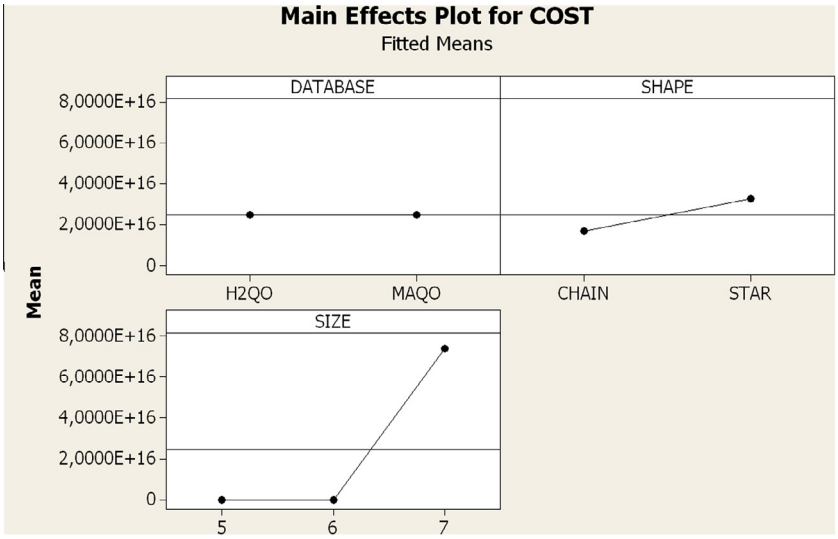


Fig. 8. Main effects – cost efficiency.

Table 4
Factors and levels – general experiment.

Factor	Level
DATABASE	H2QO, MAQO, HSQLDB, DERBY, POST
SHAPE	Chain, Star, Snowflake
SIZE	12, 16, 20, 30, 40 and 50

Table 5
ANOVA – general experiment.

Source	DF	SS	MS	Iter	Pr
DB	4	6e+12	1e+12	5000	<2e−16***
SHAPE	2	5e+11	2e+11	5000	0.0008***
DB:SHAPE	8	3e+12	4e+11	5000	<2e−16***
SIZE	1	6e+11	6e+11	5000	<2e−16***
DB:SIZE	4	3e+12	8e+11	5000	<2e−16***
SHAPE:SIZE	2	3e+11	1e+11	3606	0.0541·
DB:SHAPE: SIZE	8	2e+12	3e+11	5000	<2e−16***
Residuals	150	9.04e+12	6.03e+10		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1.

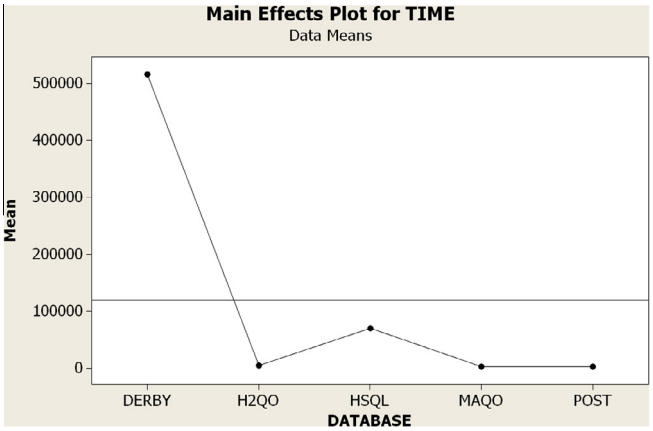


Fig. 9. Main factor – DATABASE.

Table 6
Best execution times overview.

	H2QO	HSQL	DERBY	POSTGRESQL
MAQO	74.78%	72.17%	100%	63.3%

second (5.2) to test the efficiency of the MAQO in finding global optima and the last (5.3) to compare the RDBMS H2 running the MAQO against other RDBMS. It is noteworthy that all experiments have used a factorial design of fixed effects, the statistical analysis used the R/Minitab softwares and was done with Permutational ANOVA (Permutations Test for an Analysis of Variance Design by Anderson & Braak (2003)) and all the related assumptions were not violated by the generated models. The confidence interval for the experiments was set as 95%, implying a significance level $\alpha = 0.05$ and all observations were randomized.

For each ANOVA table hereafter, DF represents the *Degrees of Freedom*, SS represents the *Sum of Squares*, MS means *Mean Square*, Iter is the number of iterations until the criterion is met and Pr refers to the *p*-value of the related statistical test.

The queries were executed in a machine with *Core i7-2600 CPU 3.40* GHz, with 16 GB *RAM*, operating system *Ubuntu* 10.10-x86_64. The timeout for the execution of the queries was set as 2 h and the maximum number of resulting tuples was limited in 10 million. According to the H2 database, a query with 50 relations and 50 projected columns could generate a result with about 9 GB for 10 million tuples. We adopted the strategy of cold cache in all executions. Thus, before the execution of each query, the OS pages in the memory are synchronized and then freed. Each one of the experiments have used different seeds to generate the distinct queries.

Finally, the RDBMS chosen to implement the optimizer was the H2 Database Engine and consequently, the cost model adopted is the same defined by the H2. For this reason, we considered only the method Nested-Loop-Join. We remark that the RDBMS H2 considers only the I/O operations in its cost model. Moreover, the possible representations for the solutions in the search space is restricted to left-depth tree.

5.1. Calibration of the algorithm

In order to calibrate the parameters of the optimizer, we have selected 1 test query with 40 relations in the chain and star graph shapes. The factors analyzed are: the maximum number of agents

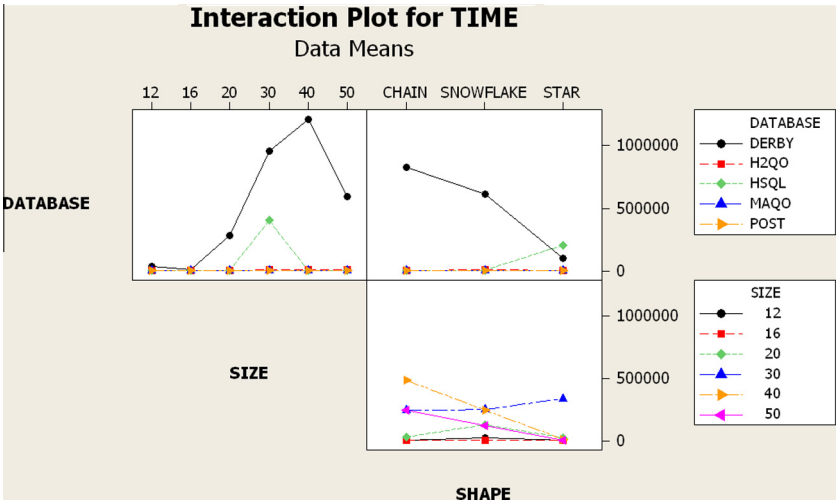


Fig. 10. Main effects and interactions – general experiment.

(NA), mutation type movement (MM), crossover method (CM), random descent type movement (RDM), semi-greedy method (SM), initial life coefficient (IL), decrement life coefficient (DL) and random descent effort (RDE). The initial life of the agents is given by the related coefficient times the number of relations in the query, the life decremented in each agent iteration is expressed by the initial life times the related coefficient and the number of iterations of the random descent without improvement is given by the effort times the number of relations in query. Table 1 describes the levels for each factor.

We have considered two complete replications, totaling 2304 observations. The response variables are the cost of the generated plans and the setup time in milliseconds. The test hypotheses are no effect for the main factors and their interactions for all levels and significance for the main factors and/or their interactions for at least one of the levels. The parameters were selected by combining the effect of both response variables, trying to get lower costs but not with necessarily higher setup times. According to the ANOVA results, the factors with significance are distributed as follows and the related null hypotheses were rejected.

According to Table 2 and the related mean values of the factors and their interactions, the maximum number of agents was set as 8, probability of 50% to choose one of the movement types for both mutation and random descent methods, crossover method with 50% of chance to use PAGX or OX, semi-greedy method defined with SCX strategy and the values 1.5, 0.1 and 0.3 for initial life coefficient, decrement life coefficient and random descent effort, respectively.

5.2. Comparison with the exhaustive method

The goal of this Subsection is to present the potential of MAQO to find global optima. To accomplish that, the proposed optimizer was compared against the official brute force method H2QO of H2 database. The comparison have used 30 different queries for problems with 5, 6 and 7 relations and chain and star graph shapes, totaling 360 queries for each optimizer distributed into two complete replications. The main factors studied are: the database algorithm (MAQO and H2QO), the problem size (5, 6 and 7) and the query shape (chain and star). The test hypotheses are no effect for the factors/interactions for all levels and significance for at least one of the levels in the factors/interactions. The results related to the response variable cost are summarized in Table 3.

According to the ANOVA Table, only the factor SIZE and its interaction with SHAPE have significance, which allows us to reject the related null hypotheses and say that part of the variability of the data can be explained by them. Thus, no statistical significance was detected between the results of the algorithms. The MAQO was capable to find the optima in 90% of the cases. Fig. 8 shows graphically the mean costs of the main factors and its interactions.

5.3. Comparison with other database systems

This Section is intended to compute the total time spent in optimization and execution of a set of queries by 4 different RDBMS. We choose 3 databases implemented in Java language and one native system implemented in C language, they are: H2,[7] HSQLDB,[8] Derby[9] and PostgreSQL,[10] respectively. A total of 2 complete experimental replications have been considered and the test-set is composed by 3 different queries for problems with 12, 16, 20, 30, 40 and 50 relations and the shapes distributed in chain, star and snowflake, totaling 540 queries. The H2 database was tested with two different optimizers, one with the official algorithm described in Section 2 and another with the method proposed in this work, namely H2QO and MAQO, respectively. The mains factors and its levels are presented in Table 4.

Considering the total time as response variable in milliseconds, the ANOVA Table 5 shows that the main factors (and some of their interactions) have significance on data variability, allowing us to reject the related null hypotheses with 95% degree of confidence. The difference in the execution times is caused by the database systems, shape and the size of the queries.

Based on the main objective of this work, that is to compare the proposed algorithm running in a real database against different real systems, the factor DATABASE is the focus of the analysis in this subsection. Fig. 9 shows that the average execution time of the DBMS Derby is the higher and the MAQO and PostgreSQL are the lowest. These results show the superiority of the developed algorithm in relation to the official optimizer of the H2 database. Table 6 gives an overview of the queries executed with lower times against the other databases.

[7] H2 – Version 1.3.174, www.h2database.com.
[8] HSQLDB – Version 2.0.0, http://hsqldb.org/.
[9] Derby – Version 10.9.1.0, http://db.apache.org/derby.
[10] PostgreSQL – Version 9.2.6 – www.postgresql.org.

Table 7
Best plans overview.

Query shape	MAQO(%) – H2QO(%)					
	12	16	20	30	40	50
Chain	50-50	0-100	50-50	100-0	50-50	100-0
Star	0-100	100-0	100-0	100-0	100-0	100-0
Snowflake	50-50	100-0	100-0	100-0	100-0	100-0

Fig. 10 presents graphic information about the main factors and their interactions. According to the average time results of all database systems, the shape snowflake and size 30 consumed more time in the optimization and execution of the queries. On the other hand, the MAQO spent on average more time on queries in the shape Star and Snowflake and size equal to 50.

Finally, Table 7 summarizes the results obtained by the H2QO and MAQO algorithms in relation to the cost of the plans. It is clear that the evolutionary multi-agent algorithm performed better in the majority of cases. The H2QO overcomes the MAQO only in the problems Chain-16 and Star-12. Additionally, out of 108 executed queries by each optimizer, only 14% of plans generated by the default planner in H2 were better than the ones generated by MAQO.

6. Conclusions and future work

This paper presented a novel database query optimizer based on evolutionary algorithms and multi-agent systems and an evolution mechanism supported by mutual selection. Many actions and profiles have been designed to provide different goals to the agents and to explore the solution space of the query join ordering problem in a smart way. The algorithm employs classical genetic operators, constructive heuristics and refinement methods according to the associated agent profiles. We proposed a new crossover method called *Pandora-Aquiles Greedy Crossover*.

The execution environment can be described as accessible, because the agents can obtain precise, updated and complete informations about the environment. Besides, it can be characterized as deterministic and dynamic. The agents are hybrids: reactive with internal state. The following features of the agents can be cited: reactivity, proactivity, sociability, veracity and benevolence.

The evaluation methodology of this work follows mainly the ideas from Swami and Gupta (1988) and Swami (1989). We developed a benchmark according to distributions based on a real production database of an IT Center of a Brazilian University. The experiments were subdivided into 3 separated evaluations: calibration of the algorithm, validation of the algorithm with a brute force method and a comparison with 3 different database systems. A factorial design was applied in all experiments with a confidence interval set as 95%. Moreover, all the related assumptions were not violated by the generated models.

For the calibration phase, it was used one test query with 40 relations in the chain and star graph shapes. According to the results, the maximum number of agents was set as 8, probability of 50% to choose one of the movement types for mutation and random descent methods, crossover method with 50% of chance to use PAGX or OX, semi-greedy method defined with SCX strategy and the values 1.5, 0.1 and 0.3 for initial life coefficient, decrement life coefficient and random descent effort, respectively.

In the comparison of the proposed optimizer MAQO against the official H2 brute force method, we have used 30 different queries for problems with 5, 6 and 7 relations and chain and star graph shapes. The results showed that MAQO was capable to find the optima in 90% of the cases.

The general test has adopted 3 different queries for problems with 12, 16, 20, 30, 40 and 50 relations and the shapes: chain, star and snowflake. The proposed optimizer was compared against the official H2 official optimizer and the database systems HSQL, Derby and PostgreSQL. According to the results, the developed algorithm showed execution times better than the official planner H2QO in 74.78% of the cases, better than PostgreSQL in 63.3% of the cases and in 100% of cases than Derby. Besides, in the same experiment, compared with H2QO, the MAQO returned plans with lower costs in 86% of the cases.

Sections 1 and 2 show the importance of the join ordering problem. It is important to say that, good execution plans, in fact, lead to faster response times in the system. Considering that the synthetic database is based on real distributions and the queries are based on common problem representations, one can expect that in an environment with few users (low concurrency), as the results suggest, the use of the proposed algorithm over the official H2 optimizer would improve the response times in 74.78% in possible real world cases. Besides, the new optimizer allowed H2 to be better than the very popular RDBMS PostgreSQL in 63.3% of the problems. Finally, according to the superiority of H2 with the new planner in relation to Derby, in a real situation similar to the proposed synthetic environment, the best decision is choosing H2.

Therefore, in view of the superiority of this novel proposed approach over others in the literature, we show the benefits of using an evolutionary multi-agent system for query join ordering optimization. Moreover, the set of actions, the available profiles and interaction mechanism have proved to be an efficient tool to explore the solution space of the related problem and the parallel feature of the algorithm has allowed it to prepare the plans with lower setup times. As a possible weakness of the proposed optimizer, it is worthy to note that all experiments were executed with only a single caller, i.e., only one query was optimized per execution. Thus, in the calibration phase, the number of agents was configured as equal to the total number of available CPU cores. However, in a environment with more then one caller, such configuration could overload the CPU.

As future work, it is worthy to cite an extension of the experiments in order to run with more than one caller. This is a interesting point and could be used to explore the proposed optimizer in a environment with high load, which can represent many real world situations. Another very important future study is the execution of the optimizer in a distributed environment. As pointed out in the review presented in Section 2, distributed database systems is a field recently explored. Consequently, the application of the optimizer to generate plans in such environment would be another contribution. Moreover, EMAS are naturally indicated for and compliant with distributed environments and problems. Thus, with new actions and ways of interaction between the agents, the optimizer could generate plans in a distributed environment. Another area recently studied is related to the use of GPU processors to support some database components and its operations. Since the proposed algorithm has agents running in parallel, the use of GPU to support its execution could be another avenue to explore. At last, the similarities between the join ordering problem in the RDBMS and RDF fields seems to be another possible area to investigate. The integration of the proposed algorithm with some RDF query engine is indicated too.

Acknowledgments

The authors would like to thank the support given by the following Brazilian agencies: State of Minas Gerais Research Foundation – FAPEMIG (Pronem 04611/10, PPM CEX 497/13); Coordination for the Improvement of Higher Level Personnel – CAPES; National Council of Scientific and Technological Development – CNPq (Grants

30506/2010-2, 312276/2013-3, 306694/2013-1); and by the IT Center (NTI) of UFOP.

We also thank the anonymous reviewers for their valuable criticism, which have greatly contributed to the final version of the manuscript.

References

Ahmed, Z. H. (2010). Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics and Bioinformatics, 3*, 96–105.

Anderson, M. J., & Braak, C. J. F. T. (2003). Permutation tests for multi-factorial analysis of variance. *Journal of Statistical Computation and Simulation, 73*, 85–113. http://dx.doi.org/10.1080/00949650215733.

Barbati, M., Bruno, G., & Genovese, A. (2012). Applications of agent-based models for optimization problems: A literature review. *Expert Systems with Applications, 39*, 6020–6028. http://dx.doi.org/10.1016/j.eswa.2011.12.015<http://www.sciencedirect.com/science/article/pii/S0957417411016861>.

Bennett, K. P., Ferris, M. C., & Ioannidis, Y. E. (1991). A genetic algorithm for database query optimization. In *International conference on genetic algorithms* (pp. 400–407).

Breß, S., Schallehn, E., & Geist, I. (2013). Towards optimization of hybrid CPU/GPU query plans in database systems. In *New trends in databases and information systems* (pp. 27–35). Springer.

Brunie, L., & Kosch, H. (1997). Optimizing complex decision support queries for parallel execution. In *Parallel and distributed processing techniques and applications* (pp. 858–867).

Codd, E. F. (1970). A relational model for large shared data banks. *Communications of the ACM, 13*, 377–387. http://dx.doi.org/10.1145/362384.362685.

Dahal, K., Almejalli, K., & Hossain, M. A. (2013). Decision support for coordinated road traffic control actions. *Decision Support Systems, 54*, 962–975. http://dx.doi.org/10.1016/j.dss.2012.10.022<http://www.sciencedirect.com/science/article/pii/S0167923612002771>.

Davis, L. (1985). Job shop scheduling with genetic algorithms. In *Proceedings of the first international conference on genetic algorithms* (pp. 136–140). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.<http://dl.acm.org/citation.cfm?id=645511.657084>.

Dong, H., & Liang, Y. (2007). Genetic algorithms for large join query optimization. In *Proceedings of the ninth annual conference on genetic and evolutionary computation GECCO '07* (pp. 1211–1218). New York, NY, USA: ACM. http://dx.doi.org/10.1145/1276958.1277193<http://doi.acm.org/10.1145/1276958.1277193>.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 26*, 29–41. http://dx.doi.org/10.1109/3477.484436.

Drezewski, R., Obrocki, K., & Siwik, L. (2010). Agent-based co-operative co-evolutionary algorithms for multi-objective portfolio optimization. In A. Brabazon, M. ONeill, & D. Maringer (Eds.), *Natural computing in computational finance. Studies in computational intelligence* (Vol. 293, pp. 63–84). Berlin/Heidelberg: Springer<http://dx.doi.org/10.1007/978-3-642-13950-5_5>.

Elmasri, R., & Navathe, S. (2010). *Fundamentals of database systems* (6th ed.). USA: Addison-Wesley Publishing Company.

Enembreck, F., & Barthès, J.-P. A. (2013). A social approach for learning agents. *Expert Systems with Applications, 40*, 1902–1916. http://dx.doi.org/10.1016/j.eswa.2012.10.008<http://www.sciencedirect.com/science/article/pii/S0957417412011220>.

Feizi-Derakhshi, M.-R., Asil, H., & Asil, A. (2010). Proposing a new method for query processing adaption in database. *CoRR, abs/1001.3494*<http://dblp.uni-trier.de/db/journals/corr/corr1001.html#abs-1001-3494>.

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Upper Saddle River, NJ, USA: Prentice Hall Press.

Ghaemi, R., Fard, A., Tabatabaee, H., & Sadeghizadeh, M. (2008). Evolutionary query optimization for heterogeneous distributed database systems. *World Academy of Science, 43*, 43–49.

Glover, F., & Laguna, M. (1997). *Tabu search*. Boston: Kluwer Academic Publishers.

Golshanara, L., Rouhani Rankoohi, S., & Shah-Hosseini, H. (2014). A multi-colony ant algorithm for optimizing join queries in distributed database systems. *Knowledge and Information Systems, 39*, 175–206. http://dx.doi.org/10.1007/s10115-012-0608-4<http://dx.doi.org/10.1007/s10115-012-0608-4>.

Gonçalves, F. A. C. A., Guimarães, F. G., & Souza, M. J. F. (2013). An evolutionary multi-agent system for database query optimization. In *Proceeding of the 15th annual conference on genetic and evolutionary computation conference GECCO '13* (pp. 535–542). New York, NY, USA: ACM. http://dx.doi.org/10.1145/2463372.2465802<http://doi.acm.org/10.1145/2463372.2465802>.

Guttoski, P. B., Sunye, M. S., & Silva, F. (2007). Kruskal's algorithm for query tree optimization. In *Proceedings of the 11th international database engineering and applications symposium IDEAS '07* (pp. 296–302). Washington, DC, USA: IEEE Computer Society. http://dx.doi.org/10.1109/IDEAS.2007.33<http://dx.doi.org/10.1109/IDEAS.2007.33>.

Han, W.-S., Kwak, W., Lee, J., Lohman, G. M., & Markl, V. (2008). Parallelizing query optimization. *Proc. VLDB Endow., 1*, 188–200<http://dl.acm.org/citation.cfm?id=1453856.1453882>.

Hanna, L., & Cagan, J. (2009). Evolutionary multi-agent systems: An adaptive and dynamic approach to optimization. *Journal of Mechanical Design, 131*, 011010.

Heimel, M. (2013). Designing a database system for modern processing architectures. In *Proceedings of the 2013 Sigmod/PODS Ph.D. symposium on PhD symposium SIGMOD'13 PhD Symposium* (pp. 13–18). New York, NY, USA: ACM. http://dx.doi.org/10.1145/2483574.2483577<http://doi.acm.org/10.1145/2483574.2483577>.

Heimel, M., & Markl, V. (2012). A first step towards GPU-assisted query optimization. In *The third international workshop on accelerating data management systems using modern processor and storage architectures, Istanbul, Turkey* (pp. 1–12). Citeseer.

Hogenboom, A., Frasincar, F., & Kaymak, U. (2013). Ant colony optimization for {RDF} chain queries for decision support. *Expert Systems with Applications, 40*, 1555–1563. http://dx.doi.org/10.1016/j.eswa.2012.08.074<http://www.sciencedirect.com/science/article/pii/S0957417412010500>.

Hogenboom, A., Milea, V., Frasincar, F., & Kaymak, U. (2009). Rcq-ga: Rdf chain query optimization using genetic algorithms. In T. Noia & F. Buccafurri (Eds.), *E-commerce and web technologies. Lecture notes in computer science* (Vol. 5692, pp. 181–192). Berlin Heidelberg: Springer. http://dx.doi.org/10.1007/978-3-642-03964-5_18<http://dx.doi.org/10.1007/978-3-642-03964-5_18>.

Ibaraki, T., & Kameda, T. (1984). On the optimal nesting order for computing N relational joins. *ACM Transactions on Database Systems, 9*, 482–502. http://dx.doi.org/10.1145/1270.1498.

Ioannidis, Y. E. (1996). Query optimization. *ACM Computing Surveys, 28*, 121–123. http://dx.doi.org/10.1145/234313.234367<http://doi.acm.org/10.1145/234313.234367>.

Ioannidis, Y. E., & Kang, Y. (1990). Randomized algorithms for optimizing large join queries. *SIGMOD Record, 19*, 312–321. http://dx.doi.org/10.1145/93605.98740<http://doi.acm.org/10.1145/93605.98740>.

Ioannidis, Y. E., & Wong, E. (1987). Query optimization by simulated annealing. *SIGMOD Record, 16*, 9–22. http://dx.doi.org/10.1145/38713.38722.

Khosravifar, B., Bentahar, J., Mizouni, R., Otrok, H., Alishahi, M., & Thiran, P. (2013). Agent-based game-theoretic model for collaborative web services: Decision making analysis. *Expert Systems with Applications, 40*, 3207–3219. http://dx.doi.org/10.1016/j.eswa.2012.12.034<http://www.sciencedirect.com/science/article/pii/S0957417412012754>.

Klyne, G., & Carroll, J. J. (2014). *Resource description framework (RDF): Concepts and abstract syntax*. W3C Recommendations<http://www.w3.org/TR/rdf-concepts/>.

Krishnamurthy, R., Boral, H., & Zaniolo, C. (1986). Optimization of nonrecursive queries. In *Proceedings of the 12th international conference on very large data bases VLDB '86* (pp. 128–137). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.<http://dl.acm.org/citation.cfm?id=645913.671481>.

Lange, A. (2010). *Uma Avaliação de Algoritmos não Exaustivos para a Otimização de Junções* (Master's thesis). Universidade Federal do Paraná, UFPR.

Lee, C., sheng Shih, C., & huei Chen, Y. (2001). Optimizing large join queries using a graph-based approach. *IEEE Transactions on Knowledge and Data Engineering, 13*, 298–315. http://dx.doi.org/10.1109/69.917567.

Li, J., Ding, C., & Liu, W. (2014). Adaptive learning algorithm of self-organizing teams. *Expert Systems with Applications, 41*, 2630–2637. http://dx.doi.org/10.1016/j.eswa.2013.11.008<http://www.sciencedirect.com/science/article/pii/S0957417413009196>.

Matysiak, M. (1995). Efficient optimization of large join queries using tabu search. *Information Sciences – Informatics and Computer Science, 83*, 77–88. http://dx.doi.org/10.1016/0020-0255(94)00094-R<http://dx.doi.org/10.1016/0020-0255(94)00094-R>.

Prud'hommeaux, E., & Seaborne, A. (2008). Sparql query language for rdf. Latest version available as <http://www.w3.org/TR/rdf-sparql-query/>. URL: <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.

Rho, S., & March, S. (1997). Optimizing distributed join queries: A genetic algorithm approach. *Annals of Operations Research, 71*, 199–228. http://dx.doi.org/10.1023/A:1018967414664<http://dx.doi.org/10.1023/A3A1018967414664>.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on management of data SIGMOD '79* (pp. 23–34). New York, NY, USA: ACM. http://dx.doi.org/10.1145/582095.582099<http://doi.acm.org/10.1145/582095.582099>.

Sevinç, E., & Coşar, A. (2011). An evolutionary genetic algorithm for optimization of distributed database queries. *Computer Journal, 54*, 717–725. http://dx.doi.org/10.1093/comjnl/bxp130<http://dx.doi.org/10.1093/comjnl/bxp130>.

Shapiro, L. D., Maier, D., Benninghoff, P., Billings, K., Fan, Y., Hatwal, K., et al. (2001). Exploiting upper and lower bounds in top-down query optimization. In *Proceedings of the international database engineering & applications symposium IDEAS '01* (pp. 20–33). Washington, DC, USA: IEEE Computer Society<http://dl.acm.org/citation.cfm?id=646290.686937>.

Steinbrunn, M., Moerkotte, G., & Kemper, A. (1997). Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal, 6*, 191–208. http://dx.doi.org/10.1007/s007780050040<http://dx.doi.org/10.1007/s007780050040>.

Stuckenschmidt, H., Vdovjak, R., Broekstra, J., & Houben, G. (2005). Towards distributed processing of rdf path queries. *International Journal of Web Engineering and Technology, 2*, 207–230. http://dx.doi.org/10.1504/IJWET.2005.008484<http://dx.doi.org/10.1504/IJWET.2005.008484>.

Swami, A. (1989). Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proceedings of the 1989 ACM SIGMOD international conference on management of data SIGMOD '89* (pp. 367–376). New York,

NY, USA: . http://dx.doi.org/10.1145/67544.66961<http://doi.acm.org/10.1145/67544.66961>.

Swami, A., & Iyer, B. (1993). A polynomial time algorithm for optimizing join queries. In *Proceedings. Ninth international conference on data engineering, 1993* (pp. 345–354). http://dx.doi.org/10.1109/ICDE.1993.344047.

Swami, A., & Gupta, A. (1988). Optimization of large join queries. *SIGMOD Record, 17*, 8–17. http://dx.doi.org/10.1145/971701.50203<http://doi.acm.org/10.1145/971701.50203>.

Tao, F., Laili, Y. J., Zhang, L., Zhang, Z. H., & Nee, A. C. (2014). Qmaea: A quantum multi-agent evolutionary algorithm for multi-objective combinatorial optimization. *Simulation, 90*, 182–204. http://dx.doi.org/10.1177/0037549713-485894<http://sim.sagepub.com/content/90/2/182.abstract>. arXiv: <http://sim.sagepub.com/content/90/2/182.full.pdf+html>.

Tewari, P. (2013). Query optimization strategies in distributed databases. *International Journal of Advances in Engineering Sciences, 3*, 23–29.

't Hoen, P. J., & Jong, E. D. (2004). Evolutionary multi-agent systems. In X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tino, A. Kabán, & H.-P. Schwefel (Eds.), *Parallel problem solving from nature – PPSN VIII. Lecture notes in computer science* (Vol. 3242, pp. 872–881). Berlin, Heidelberg: Springer<http://dx.doi.org/10.1007/978-3-540-30217-9_88>.

Vance, B., & Maier, D. (1996). Rapid bushy join-order optimization with cartesian products. *SIGMOD Record, 25*, 35–46. http://dx.doi.org/10.1145/235968.-233317<http://doi.acm.org/10.1145/235968.233317>.

Van Moffaert, K., Brys, T., Chandra, A., Esterle, L., Lewis, P. R., & Nowé, A. (2014). A novel adaptive weight selection algorithm for multi-objective multi-agent reinforcement learning. In *Proceedings of the 2014 IEEE world congress on computational intelligence*.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. John Wiley & Sons <http://books.google.com.br/books?id=X3ZQ7yeDn2IC>.

Zafarani, E., Derakhshi, M. F., Asil, H., & Asil, A. (2010). Presenting a new method for optimizing join queries processing in heterogeneous distributed databases. In *International workshop on knowledge discovery and data mining* (pp. 379–382). doi: <http://doi.ieeecomputersociety.org/10.1109/WKDD.2010.122>.

ANEXO B – Conferência - *Genetic and Evolutionary Computation Conference* - 2013

An Evolutionary Multi-Agent System for Database Query Optimization

Frederico A. C. A. Gonçalves
Graduate Program in Electrical Engineering - Federal University of Minas Gerais
Belo Horizonte, Brazil
IT Center - Federal University of Ouro Preto
Ouro Preto, Brazil
fred@nti.ufop.br

Frederico G. Guimarães
Department of Electrical Engineering - Federal University of Minas Gerais
Belo Horizonte, Brazil
fredericoguimaraes@ufmg.br

Marcone J. F. Souza
Department of Computer Science - Federal University of Ouro Preto
Ouro Preto, Brazil
marcone@iceb.ufop.br

ABSTRACT

Join query optimization has a direct impact on the performance of a database system. This work presents an evolutionary multi-agent system applied to the join ordering problem related to database query planning. The proposed algorithm was implemented and embedded in the core of a database management system (DBMS). Parameters of the algorithm were calibrated by means of a factorial design and an analysis based on the variance. The algorithm was compared with the official query planner of the H2 DBMS, using a methodology based on benchmark tests. The results show that the proposed evolutionary multi-agent system was able to generate solutions associated with low execution costs in the majority of the cases.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems; H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software; I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search

Keywords

Join Ordering, Multi-agent System, Query Optimization

1. INTRODUCTION

Database Management Systems (DBMS) are computer systems that have an essential role in the creation, storage and maintenance of information. These systems are among the most complex software systems, integrating a number of components that must work together to guarantee the correct operation of the whole environment.

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

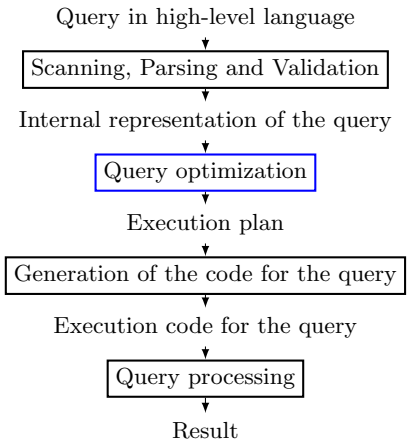


Figure 1: Typical steps for the processing of a query.

According to [9], the processing of one query can be divided into the steps shown in Figure 1. In this paper, we focus on the query optimization step. The task of this component is to define the execution plan for the query. The definition of the execution plan can involve an ordination of many types of database operations: UNION, INTERSECTION, DIFFERENCE, JOIN and so on. Specifically in the case of the join operations, the most time-consuming operation in query processing, the task can be viewed as a combinatorial optimization problem commonly known as join ordering problem. It consists in defining the best order of execution of the join operations among the relations specified in the query to be processed. The joining between two relations is an operation in relational algebra allowing the merging of information from these relations. The join operation will be treated as a binary operation (*two-way join*). However, this operation can have more than two relations as input too (*multiway-join*), and in accordance with [9], the combinations for multiway joins grow very rapidly.

Regarding query optimization, Ioannidis [16] proposes two classes of spaces: the algebric space and the space of structures and methods. The algebric space refers to the exe-

cution order of the actions considered, using relational algebra as representation (e.g. σ for data selection). The space of structures and methods, on the other hand, relates to the implementation choices available for the DBMS in hand. Possible choices include: sequential access method or index-based, nested-loop join methods, hash-join methods, or merge-join methods, among others. The different ways of representing the solutions combined with the access and join methods define different possibilities for join ordering, which implies in general an NP-complete problem [15]. A more detailed description of the join ordering problem can be found in [15, 25, 16, 23].

According to Garcia-Molina et al. [11], the I/O operations dominate the time spent in query processing. Given this observation, we can estimate the cost of a given query in terms of the number of accesses to disk and I/O operations. In addition to the accesses to disk, the CPU operations can also be considered in the cost model of the queries.

Query optimization and planning have a direct impact in the response time of the system, since a good execution plan would lead to a faster query result. In this article we present an algorithm for query optimization based on a multi-agent system. The main feature of the algorithm is to have a team of intelligent agents working together in a cooperative or competitive way to achieve the solution of the problem. Such systems differ from other parallel systems by presenting interaction between agents. These agents have characteristics such as reactivity, proactiveness, sociability, and so on [29]. An application of cooperation between agents for decision support can be found in [5].

More recently, some researchers have explored the integration between evolutionary algorithms and multi-agent systems, trying to take the best from both worlds. Evolutionary algorithms work with a population of structures that encode possible solutions for the problem [4, 7]. This population evolves iteratively by means of heuristic operators inspired or motivated by concepts of natural systems and Darwinian principles. In a typical evolutionary algorithm, the fitness of the individuals depends only on the quality of that individual in solving the problem. The integration between multi-agent systems and evolutionary algorithms lead to the so-called Evolutionary Multi-Agent Systems (EMAS). In such systems, the agents have also the ability to evolve, reproduce and adapt to the environment, competing for resources, communicating with other agents, and making autonomous decisions [8]. An example of EMAS is available in [2, 1], where the authors employ the use of Memetic Algorithm and Multi-Agent system.

There is some work in the literature regarding the join ordering problem, see Section 2. Many of these papers propose algorithms to solve the problem in an exhaustive or non-exhaustive way. Exhaustive approaches guarantee the optimal planning, while non-exhaustive methods do not. Some well-known DBMS such as PostgreSQL[1] and H2[2], employ exhaustive methods in query processing when the query involves a small number of relations. For queries that extrapolate a specified limit, due to the complexity of the problem, a non-exhaustive method is used.

The present article describes an approach that can be classified as a non-exhaustive one. As far as we know, EMAS have not been applied to such a context. Therefore, the main contribution of this article is the development of an evolutionary multi-agent system included directly in the query planner of the H2 DBMS. The agents of the system are able to interact and evolve in parallel. Additionally, the algorithm extends the behavior of purely evolutionary methods, with the inclusion of refinement and constructive heuristics in the set of actions of the agents. The results show that the proposed evolutionary multi-agent system was able to generate solutions associated with low execution costs in the majority of the cases.

The article is organized as follows. Section 2 presents and discusses the related work. The proposed algorithm is detailed in Section 3. Section 4 reports the calibration of the parameters of the algorithm and the results obtained with some benchmark tests. Section 5 presents our conclusions.

2. RELATED WORK

The join ordering problem has been studied in the literature for more than three decades. The seminal work is presented by [21], advancing an exhaustive method based on dynamic programming (DP) of a relational database system called System-R. The algorithm exhibited a time complexity $O(N!)$, and became a classic approach for solving the join ordering problem, where N stands for the number of relations in the query.

In [15], the authors demonstrate that this problem belongs in general to the class of NP-Complete. The authors used the Nested-Loop-Join as the join method. They presented two algorithms A and B, with time complexity $O(N^3)$ and $O(N^2 \log N)$, respectively. The algorithm B is applicable only to acyclic queries and guarantees the optimal solution, while the algorithm A is not restricted only to acyclic queries and does not guarantee the best solution.

An extension of Algorithm B [15] is presented by [19]. In this work, the authors developed an algorithm with time complexity $O(N^2)$ known as KBZ algorithm.

An algorithm based on the simulated annealing (SA) meta-heuristic is presented in [18] for query optimization involving many relations. The algorithm found some optimal solutions in some problems and got close to the best solutions in other test problems.

Many methods are compared in [25]: Perturbation Walk, Quasi-random Sampling, Iterated Improvement(II), Sequence Heuristic and Simulated Annealing (SA). The authors create a test database to evaluate the heuristics, considering cardinality, selectivity of the join predicates, distinct values and indices. Moreover, test queries were generated with directed graphs. According to the results, the heuristic II performed best. In a continuation of the work, another methods were tested in [24], they are: II, SA, KBZ, Greedy Algorithm and so on. According to the results, the II and the *Greedy Algorithm* performed better.

Based on the algorithms II and SA, an algorithm called *2PO* has been proposed in [17], it makes use of the two techniques mentioned. The *2PO* showed be more efficient than II and SA in the solutions quality and optimization time.

A genetic algorithm applied to the join order problem is presented in [4]. Two different strategies for crossover and mutation methods were used. The genetic algorithm was compared against the DP [21], and was capable to find better solutions to more complex problems.

[1]PostgreSQL: www.postgresql.org
[2]H2 Database: www.h2database.com

To solve some limitations of the KBZ, the authors in [26] proposed a extension of the algorithm (AB) with some improvements and a time complexity $O(N^4)$. The computational experiments compared: AB, Greedy Algorithm, DP, KBZ, II and $2PO$. The AB returned, on average, better solutions than the non-exaustive techniques, and was able to find solutions as good as those obtained by DP.

The meta-heuristic Tabu Search (TS) [13] is explored in [20]. Three methods were tested: TS, SA and II. The TS method found better solutions in almost all test problems, only for less complex problem the algorithm was worst than others.

Another exhaustive algorithm named *blitzsplit* was developed by [28]. The algorithm is based on DP and show a time complexity $O(3^N)$. According to the authors, the proposed algorithm found solutions in short time and little effort was expended to eliminate very costly solutions.

A comparative between many algorithms (DP, KBZ, SA, II, $2PO$, Genetic algorithms, among other methods) is presented in [23]. The results show that exhaustive methods fit better in less complex problems. The $2PO$ presented in general better results over solutions quality and optimization time, while SA was capable to find better solutions spending more optimization time. For low optimization times, the II, Genetic and 2PO were more indicated.

Genetic algorithms have been applied again in [7]. The authors conclude that the efficiency of the method is directly related to the cost model for the queries and also the neighborhood structures and operators used in the algorithm.

In [12], the authors present a multi-agent system for query optimization in distributed DBMS. The authors compare their method with dynamic programming, showing that their approach takes less time for processing the queries. They define the following agents: Query Distributor Agent (QDA), Local Optimizer Agents (LOAs) and Global Optimizer Agent (GOA). QDA divides the query into sub-queries and distributing them to the LOAs, that apply a local genetic algorithm. GOA is responsible for finding the best join order between the sites.

An extension of the algorithm proposed by [12] is given in [30, 10]. The extension focuses on building an adaptive system. This adaptive system presented a reduction of up to 29% in time response. A new agent called Adaptive Query Agent (AQA) was added to the system. This agent recognizes the similarity between queries. When similarities are identified, previously stored execution plans are re-used, otherwise, the query is executed and the agent's knowledge is updated. The adaptive agent also manages stored plans.

A parallel based DP algorithm is discussed in [14]. The algorithm was prototyped in the DBMS *Postgresql*, version 8.3. The algorithm improved the optimization time over classic DP [21] up to 547 times. Beyond that, for some problems, it obtained a linear speedup.

In relation to some DBMS in the market, the following strategies can be cited:

- **H2** - The version 1.3.167 from 2012[1], uses a brute force method for queries with up to 7 relations. For queries with more than 7 relations, a mixed algorithm composed by an exhaustive, greedy and random search methods is applied;
- ***Postgresql*** - The version 9.2 has an optimizer based on DP [21] for ordination of queries with up to 11 relations and a genetic algorithm for queries with more than 12 relations;
- ***Mysql***[2] - This DBMS has a hybrid algorithm based on greedy and exhaustive techniques. The exhaustive search is controlled and the greedy methodology is applied to evaluate promising solutions.

[1]www.h2database.com/html/performance.html

3. QUERY OPTIMIZER BASED ON EMAS

In this paper we developed an evolutionary multi-agent system for the join ordering problem. EMAS can extend the behavior of conventional genetic and multi-agent algorithms. The agents can act in proactive and reactive way, they can make decisions by applying differentiated interaction mechanisms with the other agents and explore the solutions space in a smart way by using the traditional operators of the genetic algorithms.

The environment is composed by a set of agents, where each agent works to find the best solution, i.e. the best join order for the relations in the query. The system can be described as $EMA = \langle E, \Gamma, \Omega \rangle$, where E is the environment, Γ is the set of resources, Ω contains the types of information available for the agents in the system. The environment is accessible, which means that the agents can obtain precise and complete information about the state of the environment. It is also non deterministic, for there is no certainty about the state resulting from a given action, and dynamic, since the agents can modify the environment during execution.

The environment E can be described as $E = \langle T^E, \Gamma^E, \Omega^E \rangle$, where T^E represents the topography, Γ^E are the resources and Ω^E is the information. We use only one kind of resource and information. The resource is expressed in terms of life points of each agent. The information about the other agents and the best current solution is represented in Ω^E.

The topography is given by $T^E = \langle A, l \rangle$, A represents the set of agents in the environment and l is a function returning the localization of a specific agent. Each agent a can be described by the tuple $a = \langle sol^a, Z^a, \Gamma^a, PR^a \rangle$, sol^a is the solution represented by the agent, actually encoded as an integer vector. Each element in the vector represents a relation in the query. The vector defines the join order for processing the query. Z^a is the set of actions of the agents. Γ^a and PR^a define the life points and profile of the agent. The set of system actions is detailed below:

- **getLife** - Used to obtain life points from another agent;
- **giveLife** - Give life points to another agent;
- **findWorse** - Search for an agent with worse fitness value;
- **findPartner** - Search for a partner in the set of agents in the environment;
- **crossover** - Crossover operator used to generate offspring;
- **mutation** - Mutation operator;
- **updateBestSolution** - Update the best solution;

[2]MySQL: www.mysql.com

- **randomDescent** - Random descent local search;
- **bestImprovement** - Best Improvement Method;
- **semiGreedyBuild** - Construct a solution by using a semi-greedy heuristic;
- **processRequests** - Evaluate and process pending requests;
- **changeProfile** - Change Profile of the agent;
- **die** - action of dying, when the life points of the agent ends.

Each agent is associated with an execution thread. These agents can be classified as hybrid ones, since they are reactive with internal state and deliberative agents seeking to update the best solution and not to die. The following characteristics can be highlighted: Reactivity, Pro-activity, Social ability and Veracity. The mutation and local search operators employ swap movements as illustrated in Figure 2.

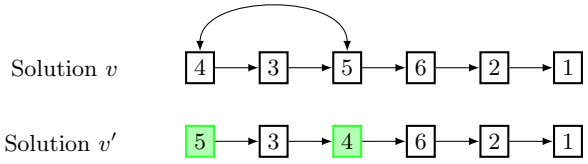


Figure 2: Swap movement - N^S

Two crossover operators are available: *Ordered Crossover* - OX [6] and *Sequential Constructive Crossover* - SCX [3]. When the crossover action is executed, one of these operators is selected at random. The examples described next use as input the solutions in Figure 3.

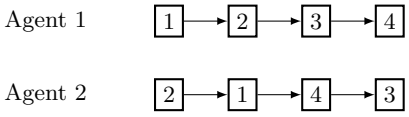


Figure 3: Initial agents

In the crossover OX, a crossover point is selected at random, which defines the part of Agent 1 that goes to the descendent. The rest of the sequence is taken from Agent 2 in an ordered way avoiding repetition of elements. Considering the agents in Figure 3 and a crossover point after the second relation, the resulting descendent is presented in Figure 4.

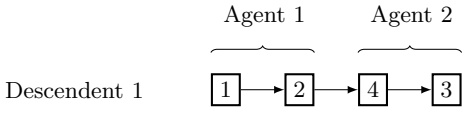


Figure 4: Descendent OX

In the strategy SCX, the cost of the join operations between each pair of relations is stored in a cost matrix. The values in the cost matrix are based on the cost model currently used in H2. The method starts by adding the first relation from Agent 1 to the descendent. After that, the descendent inherits the next relation from the parent with the smallest cost in the given cost matrix. Ties are solved randomly. For the agents in Figure 3, the descendent is given in Figure 5.

Cost Matrix				
	R1	R2	R3	R4
R1	99	10	20	5
R2	10	99	15	35
R3	20	15	99	45
R4	5	35	45	99

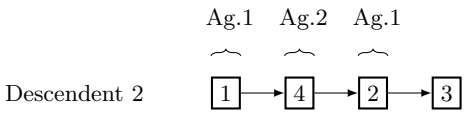


Figure 5: Descendent SCX

The action semiGreedyBuild is based on a semi-greedy (greedy-random) heuristic, which introduces randomness into a greedy constructive heuristic. A new solution is constructed by inserting relations step by step. For each insertion, the candidate elements are sorted according to a greedy function. In this work, we adopt two different greedy functions, based on two classification criteria (selected at random). The first one is based on the number of links between each relation to the other in the query. The relations with less links are better classified by the greedy function. The other approach is based on the cost of links. The relations with the smaller costs are better classified by the greedy function. The best elements form the restricted candidate list (RCL). The new element to be added to the solution is randomly taken from the RCL. When all relations have been inserted, a new solution is returned.

In a multi-agent system, the agents can execute different goals, which can be: survive, generate a solution together with another agent, refine his own solution and so on. To accomplish that, different profiles were designed to different goals, they are:

- **RESOURCE** - The agent tries to increase his life points by searching agents with worse fitness values and requesting part of their life points;
- **REPRODUCTION** - The agent looks for partners to produce new solutions by crossover;
- **MUTANT** - The agent suffers mutation;
- **RANDOM_DESCENT** - The agent refines its own solution by using random descent local search;
- **SEMI-GREEDY** - The agent builds new solutions by using the semi-greedy heuristic;

The Multi-Agent Query Optimizer (MAQO) pseudo-code is presented in the Algorithm 1. This algorithm first initializes the system (line 2). Then, three agents are created with the profiles RANDOM_DESCENT, SEMI-GREEDY

and MUTANT respectively (lines 4, 6 and 8). The initial solution used by the agent RANDOM_DESCENT is the same order given by the query, that is, the order in which the relations are read during the initial parsing of the query. The other solutions are constructed using the number of links as greedy function. The initialization of the agents and final refinement of the solution are done in lines 14, 16 and 18. The Best Improvement Method guarantees that the returned solution will be a local optimum in relation to the neighborhood N^S.

Algorithm 1: MAQO

```
   Data: QueryRelations
   Result: sol*
 1 begin
 2 |  Environment E ← initializeEnvironment();
 3 |  //Create at least one agent with the profile
   |  RANDOM_DESCENT;
 4 |  createAgent(E, RANDOM_DESCENT);
 5 |  //Create at least one agent with the profile
   |  SEMI-GREEDY;
 6 |  createAgent(E, SEMI-GREEDY);
 7 |  //Create at least one agent with the profile
   |  MUTANT;
 8 |  createAgent(E, MUTANT);
 9 |  //Create the other agents with the profile
   |  REPRODUCTION;
10 |  for i=4 to E →MAXAGENTS do
11 |  |  createAgent(E, REPRODUCTION);
12 |  end
13 |  //Initialize threads;
14 |  initializeAgents(E);
15 |  //Wait threads to finish;
16 |  waitAgents(E);
17 |  //Best Improvement on Best Agent;
18 |  E →bestAgent→bestImprovement();
19 |  return E →bestSolution;
20 end
```

Once the agents have been initialized, the evolutionary process will also start, since agents may evolve through the execution of the related actions, which can be individual or joint actions. Each agent executes the goal of each profile by performing actions associated to that profile. We present next how the set of actions is distributed in each profile:

- **RESOURCE** - findWorse, processRequest, getLife, changeProfile and die;
- **REPRODUCTION** - findPartner, processRequest, crossover, updateBestSolution, changeProfile and die;
- **MUTANT** - mutation, processRequest, updateBestSolution, changeProfile and die;
- **RANDOM_DESCENT** - randomDescent, processRequest, updateBestSolution, changeProfile and die;
- **SEMI-GREEDY**- semiGreedyBuild,processRequest, updateBestSolution, changeProfile and die;

At every iteration of the agent in a given profile, its life points decrease. When an agent has no more life points, it executes the action die. When all agents have died, the algorithm stops. The action processRequests depends on the kind of request made. For instance, if an agent requests life points from another agent, the action processRequest will execute the action giveLife in response to the request. Changing profile is allowed only after a minimum number of iterations in the current profile or when the life level gets to a predetermined critical limit. In this case, the new profile becomes RESOURCE.

4. EXPERIMENTS

The computational experiments were performed in a *Core i7-2600 CPU 3.40*GHz, with 16 GB *RAM*, operational system *Ubuntu* 10.10-x86_64. The algorithm was developed in Java and IDE Netbeans. The DBMS H2 version 1.3.167 was used in the implementation. In the literature, deterministic evaluation methodologies are found in [28, 22] and random methodologies in [25, 24]. Benchmarks for evaluation of database systems were made available by the corporation Transaction Processing Performance Council[1] (TPC). The design of experiments proposed here is based on [25, 24]. The database was created and loaded according to the following information:

- **Cardinality distribution of tuples**- [10, 100) - 20%, [100, 1000) - 64%, [1000, 10000] - 16%;
- **Distribution of distinct values of the tuples** - [0, 0.2) - 75%, [0.2, 1) - 5%, 1 - 20%;
- **Relations Columns** - three per relation, being one reserved for primary key;
- **Percentage of indices and foreign keys** - 25% and 25%, respectively.

The number of relations presented in the queries was fixed in: 30, 50, 80 and 100 relations. Besides, the generation of the test queries was guided by four different graph shapes: chain, grid, star and multi-star. Actually, 150 relations were built using different values by selecting them randomly in the intervals described previously, but all test queries involve 100 relations maximum. The test queries were generated at random, varying the selection of relations and the selection of columns to form the join predicate. In all cases, a set of 10 queries in graph shape was created. It is worth noting that a series of sanity checks are executed to allow the creation of the foreign keys correctly. An example of a query in chain format is presented as follow:

Listing 1: Chain - SQL query

```
SELECT * FROM R1, R7, R2, R10, R150, R50 WHERE
    R1.col1 = R7.col2 AND R7.col3 = R2.col2 AND
    R2.col1 = R10.col3 AND R10.col1 = R150.col3 AND
    R150.col2 = R50.col1
```

The cost model adopted is the same defined by the DBMS H2. For this reason, we considered only the method Nested-Loop-Join. We remark that the DBMS H2 considers only the I/O operations in its cost model. Moreover, the possible representations for the solutions in the search space is restricted to left-depth tree. H2 also does not include in its model values related to intermediate results of the join operations, as presented in [27]. Therefore, the cost of a given join depends on the cardinality of the relations in the

[1]TPC: http://www.tpc.org

join operation, neglecting the intermediate results. More information about common restrictions adopted to limit the exploration of the solutions space are available in [17, 23, 11].

4.1 Calibration of the algorithm

In order to calibrate the parameters of the algorithm, we have selected 5 test queries with 30 relations in the form multi-star graph. We have used a factorial design of fixed effects. The factors analyzed are: the maximum number of agents and initial life percentage. The life of the agents is given by this percentage times the number of relations in the query. Table 1 describes the levels for each factor.

Table 1: Factors and levels

Factor	Level
Maximum number of agents	8, 16, 32, 64
Initial life	1, 2, 3, 4

We have considered two complete replications, totaling 160 observations. The number of observations is determined as follows: $N_t = N_c \times N_q \times N_r = 4^2 \times 5 \times 2 = 160$, where N_c is the number of possible combinations, N_q is the number of queries, and N_r is the number of replications. The response variables are the cost of the generated plan and the setup time. The confidence interval was set as 95%, implying a significance level $\alpha = 0.05$. All observations were randomized. The test hypotheses are:

1. $\mathbf{H_0}$: $\tau_1 = \tau_2 = \ldots = \tau_a = 0$ (no effects for the factor Maximum number of agents);
   $\mathbf{H_1}$: at least one $\tau_i \neq 0$.

2. $\mathbf{H_0}$: $\beta_1 = \beta_2 = \ldots = \beta_b = 0$ (no effects for the factor Initial life percentage);
   $\mathbf{H_1}$: at least one $\beta_j \neq 0$.

3. $\mathbf{H_0}$: $(\tau\beta)_{11} = (\tau\beta)_{12} = \ldots = (\tau\beta)_{ab} = 0$ (no interaction effect for the factors);
   $\mathbf{H_1}$: at least one $(\tau\beta)_{ij} \neq 0$.

The statistical analysis was done with (ANOVA - Analysis of Variance). According to the results obtained in this experiment, there was no significant factors with respect to the setup computational time and the cost of the generated plans. Regarding the cost, all generated plans achieved the same cost. The parameters chosen were the ones associated with the minimum setup time. A summary of the results generated by the statistical software *Minitab* with respect to the setup times is given in Table 2. In this table, DF represents the *Degrees of Freedom*, SS represents the *Sum of Squares*, MS means *Mean Square*, F and P refer to the type and p-value of the statistical test.

The selection of values for the parameters was based on Figure 6. Based on this experiment we have chosen 32 agents and life equal to 1. The generated model did not violate any of the established assumptions.

From the experiments, we note that some plans considered better with respect to the cost, actually took more time to process the query than other plans with higher costs. This behavior shows that the cost model of the DBMS H2 does not fit well with the real costs.

Table 2: ANOVA - Setup time of the generated plan

Source	DF	SS	MS	F_0	P
agents	3	22239	7413	0,62	0,613
life	3	21532	7177	0,60	0,625
agents×life	9	114160	12684	1,06	0,441
Error	16	191915	11995		
Total	31	349847			

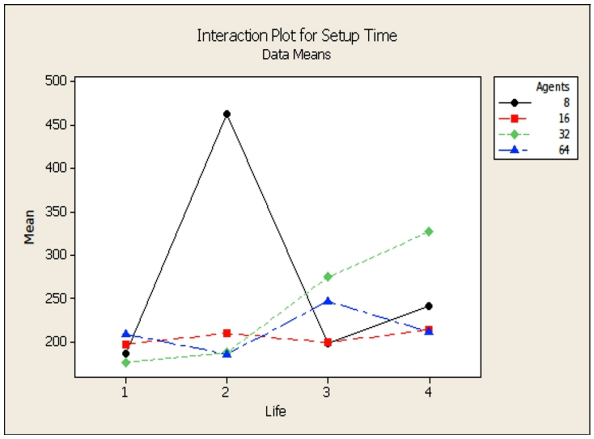


Figure 6: Setup time - Interaction of factors

4.2 Results

For the analysis of the results, we have used again a factorial design of fixed effects. Two different algorithms were tested, the default algorithm in H2 (H2 Query Optimizer - H2QO) and the developed evolutionary multi-agent algorithm (MAQO). Another factor studied was the format of the generated queries. Moreover, 3 complete replications were executed, giving a total of 480 executions to each planner. The output variable for data analysis was the cost of the generated plans. The confidence interval was set as 95%.

The total of executions of each algorithm is given by the following equation: $N_t = N_c \times N_q \times N_r = 16 \times 10 \times 3 = 480$, where N_c is the number of possible combinations, N_q is the number of grouped queries and N_r is the total of replications. The hypotheses are:

1. $\mathbf{H_0}$: no effect for the main factors and his interactions for all levels;
   $\mathbf{H_1}$: effect for the main factors and/or his interactions for at least one of the levels.

For the test queries, the cost increases as the complexity of the problem increases. For this reason we study the behavior of each algorithm in each dimension separately. The average costs were calculated in logarithmic scale. This was done to avoid that the results recorded in the DBMS get greater than the maximum limit in H2. The query timeout was set to 30 minutes. It is noteworthy that normality, independence and homoscedasticity assumptions were not violated by the generated models. The results were similar for 30, 50, 80 and 100 relations. Due to limited space, only the results

for 100 relations will be presented. All the results will be available from the authors [1].

4.3 Queries with 100 relations

Table 3 shows that the factors (and their interaction) have significance on data variability. The difference in the costs is caused by the algorithms and the shape of the queries. Consequently, the null hypothesis can be reject with 95% degree of confidence. Besides, the determination coefficient $R-Sq(adj)$ means that the generated model totally explains of the variability.

Table 3: ANOVA - 100 relations

Source	DF	SS	MS	F_0	P
Blocks	2	0,1	0,0	1,27	0,3
Alg.	1	3430,4	3430,4	96798,87	0,0
Shape	3	33301,8	11100,6	313239,46	0,0
Alg.Shap.	3	1697,3	565,8	15965,38	0,0
Error	14	0,5	0,0		
Total	23	38430,1			

S = 0,188250 R-Sq = 100,00% R-Sq(adj) = 100,00%

Based on Figure 7, we can verify that the average cost of the factor Algorithm in the level MAQO is lower than in the level H2QO. This results shows the superiority of the developed algorithm. Regarding the format of the queries, the GRID shape returned lower costs, while the MULTI-STAR shape was the most difficult to solve. It is also observed in Figure 8, that the evolutionary algorithm reached lower costs in all shapes.

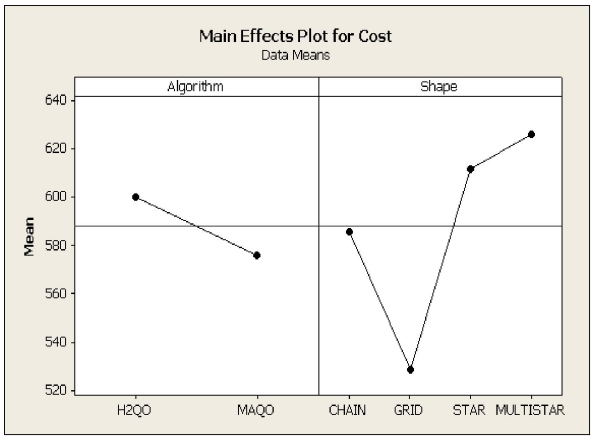


Figure 7: Main Effects - 100 relations

4.4 Discussion

It is possible to observe that the developed algorithm is able to obtain better results in terms of the execution cost of the plans in all cases. Additionally, out of 480 executed queries, only 1% of plans generated by the default planner in H2 were better than the ones generated by MAQO.

Table 4 presents an overview of the results obtained by both algorithms. It is clear that the multi-agent algorithm performed better in all test problems.

[1]http://cpdee.ufmg.br/∼zepfred/gecco2013/

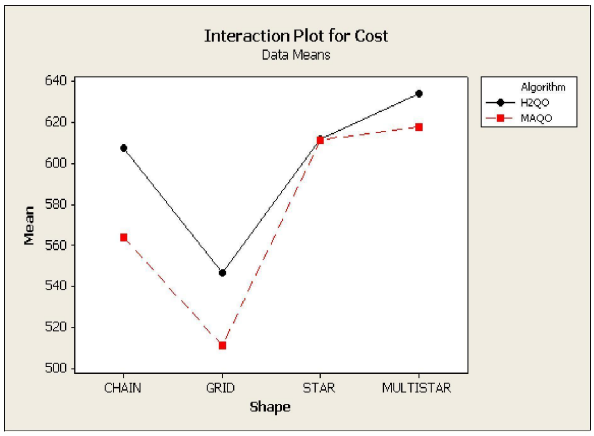


Figure 8: Interaction Effects - 100 relations

Table 4: Best Plans Overview

Query Shape	MAQO(%) - H2QO(%) 30	50	80	100
Chain	93 - 7	100 - 0	100 - 0	100 - 0
Grid	100 - 0	100 - 0	100 - 0	90 - 10
Star	100 - 0	100 - 0	100 - 0	100 - 0
Multi-star	97 - 3	100 - 0	100 - 0	100- 0

5. CONCLUSIONS

This article presented an algorithm based on evolutionary multi-agent systems. A number of actions and profiles for the agents were defined. We developed a benchmark of test queries following ideas from [25, 24] to evaluate and compare the developed algorithm and the default planner available in the DBMS H2. The test problems involve queries with 30, 50, 80 and 100 relations. The shapes of the representations adopted for the queries were graphs of the following types: chain, grid, star and multi-star.

The proposed algorithm was implemented in the optimization core of the DBMS H2. Another factorial design was used to compare MAQO and the official planner H2QO. Based on the results of the experiments, it is possible to say that MAQO was able to find low cost plans in 99% of the cases. The results suggest the superiority of the proposed approach and the benefit of using an evolutionary multi-agent system for query optimization and planning. However, it was observed that the cost model in H2 does not fit well with real costs of the plan. For this reason, the algorithm proposed has achieved longer execution times in some problems, even though the estimated cost was lower. Because of this fact, only the costs of the plans were considered.

We believe that the inclusion of the intermediate results in the join operations and the use of statistical information might improve the coherence of the cost model with the real cost of the queries.

Given the results reported, one can conclude that the developed algorithm is better than the official planner in H2. By using a more accurate cost model, the efficiency of the proposed query optimizer would be self-evident, representing real reduction in execution times.

6. ACKNOWLEDGMENTS

The authors would like to thank the support given by NTI UFOP, CAPES, CNPq (grants 305506/2010-2 and 306458/2-010-1), FAPEMIG (grants PPM-469-11, APQ-04611-10 and PPM-00366-12).

7. REFERENCES

[1] G. Acampora, J. Cadenas, V. Loia, and E. Ballester. Achieving memetic adaptability by means of agent-based machine learning. *Industrial Informatics, IEEE Transactions on*, 7(4):557–569, 2011.

[2] G. Acampora, J. Cadenas, V. Loia, and E. Ballester. A multi-agent memetic system for human-based knowledge selection. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 41(5):946–960, 2011.

[3] Z. H. Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics and Bioinformatics*, 3:96–105, 2010.

[4] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis. A Genetic Algorithm for Database Query Optimization. In *International Conference on Genetic Algorithms*, pages 400–407, 1991.

[5] K. Dahal, K. Almejalli, and M. A. Hossain. Decision support for coordinated road traffic control actions. *Decision Support Systems*, 54(2):962 – 975, 2013.

[6] L. Davis. Job shop scheduling with genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 136–140, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[7] H. Dong and Y. Liang. Genetic algorithms for large join query optimization. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1211–1218, New York, NY, USA, 2007. ACM.

[8] R. Drezewski, K. Obrocki, and L. Siwik. Agent-based co-operative co-evolutionary algorithms for multi-objective portfolio optimization. In A. Brabazon, M. ONeill, and D. Maringer, editors, *Natural Computing in Computational Finance*, volume 293 of *Studies in Computational Intelligence*, pages 63–84. Springer Berlin / Heidelberg, 2010.

[9] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[10] M.-R. Feizi-Derakhshi, H. Asil, and A. Asil. Proposing a new method for query processing adaption in database. *CoRR*, abs/1001.3494, 2010.

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[12] R. Ghaemi, A. Fard, H. Tabatabaee, and M. Sadeghizadeh. Evolutionary query optimization for heterogeneous distributed database systems. *World Academy of Science*, 43:43–49, 2008.

[13] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[14] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1(1):188–200, Aug. 2008.

[15] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N relational joins. *ACM Transactions on Database Systems*, 9:482–502, 1984.

[16] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28:121–123, March 1996.

[17] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. *SIGMOD Rec.*, 19(2):312–321, May 1990.

[18] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. *Sigmod Record*, 16:9–22, 1987.

[19] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 128–137, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[20] M. Matysiak. Efficient optimization of large join queries using tabu search. *Inf. Sci. Inf. Comput. Sci.*, 83(1-2):77–88, Mar. 1995.

[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[22] L. D. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H.-m. Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *Proceedings of the International Database Engineering & Applications Symposium*, IDEAS '01, pages 20–33, Washington, DC, USA, 2001. IEEE Computer Society.

[23] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, Aug. 1997.

[24] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 367–376, New York, NY, USA, 1989. ACM.

[25] A. Swami and A. Gupta. Optimization of large join queries. *SIGMOD Rec.*, 17(3):8–17, June 1988.

[26] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 345 –354, apr 1993.

[27] A. Swami and K. B. Schiefer. On the estimation of join result sizes. In *Proceedings of the 4th international conference on extending database technology: Advances in database technology*, EDBT '94, pages 287–300, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[28] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. *SIGMOD Rec.*, 25(2):35–46, June 1996.

[29] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009.

[30] E. Zafarani, M. F. Derakhshi, H. Asil, and A. Asil. Presenting a new method for optimizing join queries processing in heterogeneous distributed databases. *International Workshop on Knowledge Discovery and Data Mining*, 0:379–382, 2010.