Otimização da Junção por Laço Aninhado Usando Expressões da Cláusula *Where*

Guilherme Batista Mendes Pereira (aluno), Manoel Palhares Moreira (orientador)

Instituto de Informática da Pontifícia Universidade Católica de Minas Gerais (PUC Minas) Belo Horizonte, MG – Brasil

gbmpereira@gmail.com, palhares@pucminas.br

Abstract*: Database management systems (DBMS) have been widely used in computer systems, and nowadays, with multiple transactions per second, there have been several studies to improve the query execution performance. The aim of this study is to improve the performance of Nested Loop Join, the simpler join method, although present in most DBMS. Statistics from tables related to the join condition were gathered in execution time and used to limit the number of iterations of the loops from this method, so the query response time could be improved. The proposed algorithm was implemented in SQLite, but as shown in the tests this optimization can be beneficial to other databases systems.*

Resumo*: Sistemas gerenciadores de banco de dados (SGBD) têm sido amplamente utilizados em sistemas computacionais e, atualmente, com várias transações por segundo, há um foco em melhorar a execução de suas consultas. Este trabalho se dedica ao melhoramento da Junção por Laço Aninhado, por ser o método mais simples de execução de junções, mas presente em quase todos SGBDs. Estatísticas das tabelas participantes de uma junção, coletadas em tempo de execução, foram usadas para limitar o número de iterações dos laços deste método, melhorando o tempo de resposta da consulta. O algoritmo proposto foi implementado no SQLite, mas como indicaram os testes, a otimização pode beneficiar outros sistemas de banco de dados.*

1. Introdução

Um dos principais motivos para o sucesso dos sistemas gerenciadores de bancos de dados (SGBD) relacionais é o otimizador de consultas. Esse componente é capaz de determinar o modo mais eficiente para executar as consultas feitas a um banco, eliminando do usuário a responsabilidade dessa tarefa e melhorando o desempenho de todo o processo. Assim, a otimização de consultas é uma área que vem sendo muito estudada para que as decisões tomadas pelo otimizador sejam aprimoradas.

Uma consulta a um SGBD é um código desenvolvido por um usuário na linguagem *Structured Query Language* (SQL), e enviado ao gerenciador, para que este retorne dados armazenados de acordo com o que foi solicitado. Para executar esse código, o banco de dados pode optar por possíveis planos de execução. O papel do otimizador de consultas é então escolher dentre estes planos, aquele que possivelmente gaste menos recursos computacionais e tenha o melhor tempo de resposta.

Uma das decisões feitas pelo otimizador é qual algoritmo o sistema usará para executar cada operador da consulta, principalmente o de junção, o operador responsável

pela união dos dados de duas tabelas. Há vários métodos para a execução de junções, sendo a Junção por Laço Aninhado (JLA) o mais simples deles. Esse método apresenta a maior complexidade de tempo de execução, mas é o mais indicado quando as tabelas participantes possuem poucos registros. O problema é que o otimizador pode tomar decisões ruins caso as estatísticas dos registros estejam desatualizadas e pode, por exemplo, acabar escolhendo a JLA mesmo que as cardinalidades das tabelas participantes sejam grandes. Desse modo, caso haja poucas tuplas resultantes, a maioria das iterações do laço se tornam desnecessárias, desperdiçando assim recursos que poderiam estar sendo usados por outras consultas. Portanto é interessante estudar um meio de melhorar esse algoritmo.

Para contornar esse problema alguns SGBDs usam alternativas à JLA, como por exemplo, o MySQL e seu *Block Nested Loop Join* (Oracle Corporation and/or its affiliates, 1997), uma solução otimizada mas que ainda possui o mesmo problema das iterações desnecessárias. Já alguns estudos citados na Seção 2, alteram a lógica do otimizador de consultas para fazer uma otimização em tempo de execução, diminuindo a possibilidade de uma consulta ser executada por um plano de execução não otimizado. Apesar dessas melhorias, um estudo para o aperfeiçoamento do algoritmo da JLA mais simples ainda é válido, pois muitos SGBDs ainda o utilizam.

Este trabalho tem como objetivo, portanto, a otimização do método de junção JLA, para reduzir as iterações desnecessárias dos laços do algoritmo empregado, obtendo ganhos em seu desempenho. Para tal, interferiu-se no algoritmo de execução deste método no SQLite (Owens, 2006), SGBD que faz uso de apenas um método de junção, a JLA. Desse modo, qualquer otimização nesse método será amplamente perceptível nesse gerenciador.

Este artigo apresenta na Seção 2, trabalhos relacionados que estudaram diferentes formas de aperfeiçoar o processo de otimização de consultas. A Seção 3 apresenta uma explicação detalhada sobre o operador de junção e como funciona cada um dos principais métodos de execução deste operador, principalmente a JLA. A Seção 4 descreve a otimização proposta. A Seção 5 apresenta os testes e a análise dos resultados. Por último são apresentadas as conclusões do trabalho.

2. Trabalhos Relacionados

A demanda de armazenamento e consultas de dados em praticamente todos os sistemas computacionais vem crescendo ao longo dos anos. Logo, a otimização da consulta aos dados se tornou essencial para melhorar o desempenho não só em relação à manipulação dos dados, como dos sistemas de maneira geral.

No System-R (Selinger, 1979), SGBD construído pela IBM na década de 1970 foi utilizado um processamento de consultas que se tornou padrão em praticamente todos sistemas de banco de dados, o de otimizar a consulta para depois executar. Assim, há dois grandes ramos de estudo para o aperfeiçoamento do desempenho em banco de dados: como as consultas são otimizadas antes da execução, e como as consultas são executadas.

Um exemplo de trabalho que estuda possíveis melhoramentos na otimização de consultas antes da execução, é o de Chaudhuri e Shim (1994) que propõe uma alteração para que o operador *group-by* seja incluído na árvore do plano de execução entre as junções, e não adiado até que todas junções tenham sido executadas, como nos

processadores de consultas tradicionais. É uma alteração importante, pois o custo de processamento de uma consulta pode ser reduzido uma vez que o *group-by* diminui a cardinalidade de uma relação, e se for corretamente priorizado na árvore de execução, pode salvar parte dos custos de junções subseqüentes.

Já no contexto de estudo de melhorias na execução de consultas, pode-se citar uma técnica chamada de Processamento Adaptativo de Consultas (PAC) (Deshpande, 2007). O objetivo do PAC é montar um plano de execução que é adaptável às condições de execução em tempo real. Isto é possível intercalando a execução da consulta com a exploração ou modificação do plano de execução. A diferença das diferentes técnicas de PAC podem ser explicadas por como cada uma faz esta intercalação.

Um exemplo de PAC é o *Mid-Query Reoptimization* (Kabra, 1998). Esse método funciona adicionando um ponto de verificação em lugares chaves da árvore de execução, normalmente antes das junções. O ponto de verificação monitora a cardinalidade das tuplas que passam por ele e se o valor for próximo do valor estimado pelo otimizador, ele não faz nada. Caso contrário, a execução é terminada, o que já foi processado é guardado em uma tabela temporária, e o restante da consulta com esta nova tabela temporária é enviada novamente ao otimizador para montar um novo plano de execução, como demonstrado na Figura 1.

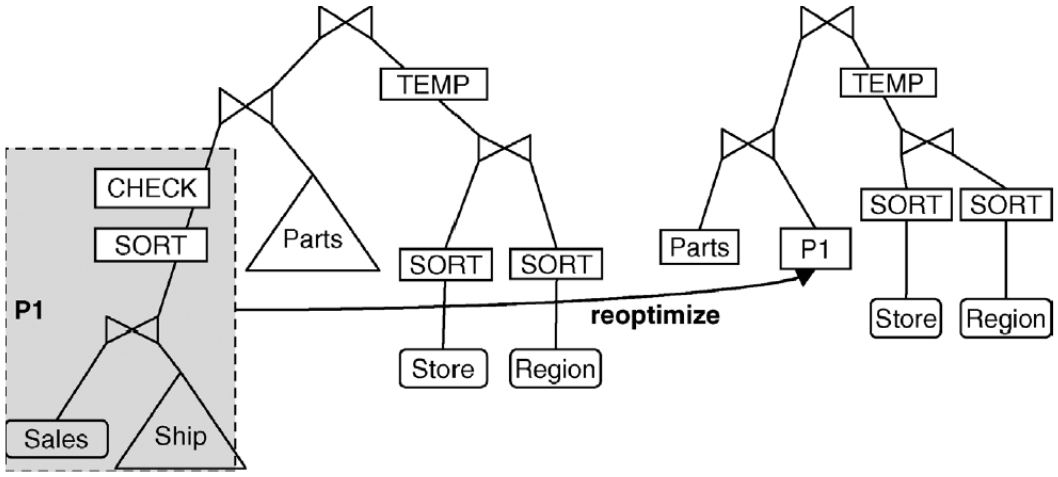


Figura 1: Reotimizando com Mid-Query Reoptimization

Fonte: Deshpande (2007)

A modificação apresentada neste trabalho, assim como as técnicas de PAC, objetiva um melhoramento no modo como as consultas são executadas, mais especificamente, no modo como as junções são executadas. A seguir apresentam-se os principais métodos de junções existentes na maioria dos SGBDs e em quais situações elas são escolhidas preferencialmente aos outros.

3. Métodos de Junção

Para um melhor entendimento da execução de consultas com junção, do qual este trabalho se dedica, esta seção aborda os principais métodos de junção, seu funcionamento e em quais circunstâncias os otimizadores de consultas as escolhem.

Junções são operadores que permitem buscar dados de mais de uma tabela do banco de dados. Uma junção é caracterizada por duas tabelas na cláusula *from* e a relação entre essas tabelas é definida através da existência de uma ou mais condições na cláusula *where*, ou cláusula *on*, relacionando as duas tabelas.

Para executar uma consulta com junção eficientemente, o método de junção que o otimizador escolhe é essencial. Junção por Laço Aninhado, Junção por Hash e Junção por Ordenação-Intercalação são os principais, e presentes na maioria dos SGBDs.

3.1. Junção por Ordenação-Intercalação

Como o nome implica o método Junção por Ordenação-Intercalação (*Sort-Merge Join*) começa ordenando ambas as tabelas pelas colunas indicadas nos predicados da junção para depois fundi-las com essas mesmas colunas. Ele é a melhor opção quando as linhas participantes da junção já estão ordenadas, ou quando alguma operação já feita pela consulta ordenou os dados. O Oracle por exemplo, segue os seguintes passos (Chen, 2007):

- Ordena todas as linhas que serão usadas na junção se estas já não tiverem sido ordenadas por uma operação que já passou. As linhas são ordenadas pelas colunas usadas na condição da junção.
- Mescla (*merge*) as duas tabelas, ou fontes, de modo que as linhas de ambas as fontes são combinadas pelo valor igual da coluna usada na condição da junção, para assim retornar a linha resultante como mostrado na Figura 2.

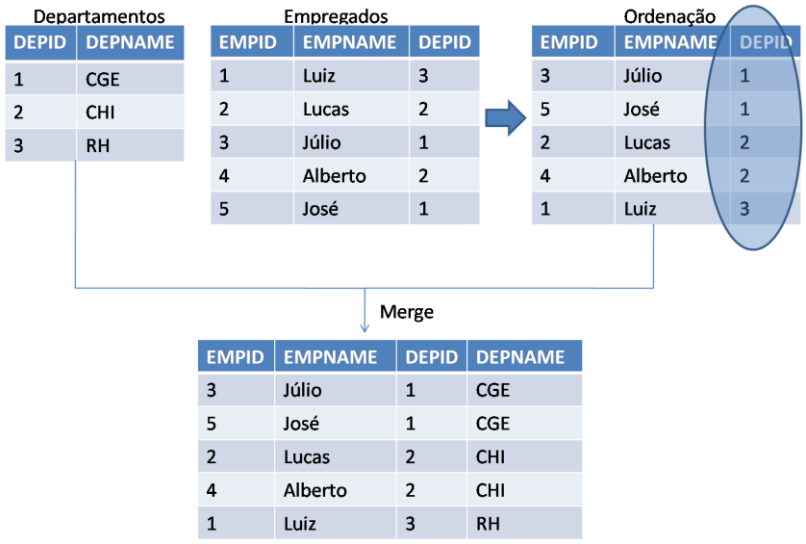


Figura 2: Funcionamento da Junção por Ordenação-Intercalação

O custo desse método pode ser dado pela fórmula (Ramakrishnan, 2003):

Custo = custo de acesso da TABELA1 + custo de acesso da TABELA2 + (custo da ordenação da TABELA1 + custo da ordenação da TABELA2)

Se a TABELA1 ou a TABELA2 já chegam à junção ordenadas, o custo de suas ordenações será zero.

3.2. Junção por Hash

O método Junção por Hash é mais usado quando há tabelas muito grandes participando da junção, e é apenas escolhido se estiver sendo usado um *equi-join* (ou seja, a comparação da junção é uma igualdade). O processador de consultas cria uma tabela *hash* na memória com a menor das duas tabelas, e varre a maior tabela comparando com o valor de *hash*, obtendo assim as linhas resultantes. Oracle por exemplo (Chen, 2007) segue os seguintes passos para execução desse método, que é também representado na Figura 3:

- Faz uma varredura de tabela completa em cada uma das tabelas participantes e divide cada uma com quantas partições possíveis de acordo com a disponibilidade de memória.
- Monta uma tabela *hash* com uma das partições
- Usa a partição correspondente na outra tabela para sondar a tabela *hash*. Todos pares de partições que não cabem na memória são colocados no disco.
- Para cada par de partições (uma de cada tabela), Oracle usa a menor delas para montar a tabela *hash* e a maior delas para sondar a tabela *hash*.

O custo da Junção por Hash pode ser dado pela fórmula (Ramakrishnan, 2003):

Custo = (custo de acesso da TABELA1 * número de partições hash da TABELA2) + custo de acesso da TABELA2

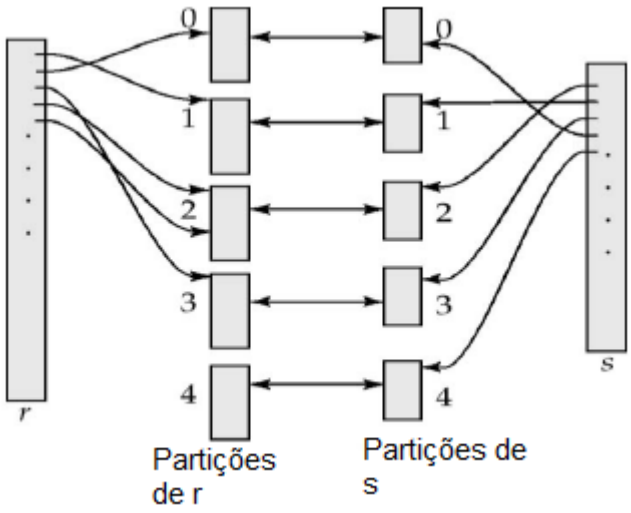


Figura 3: Funcionamento da Junção por Hash de duas tabelas: "r" e "s"

Fonte: Hemalatha (2010)

3.3. Junção por Laço Aninhado

A JLA é o método mais simples dentre todas as junções. Como o nome diz, é um laço de repetição aninhado onde para cada linha encontrada do laço externo que interage em uma das tabelas da junção, ele percorre todas as linhas da outra tabela no laço interno, procurando nas colunas participantes da condição da junção, valores que combinam, para então retornar as linhas resultantes como demonstrado na Figura 4.

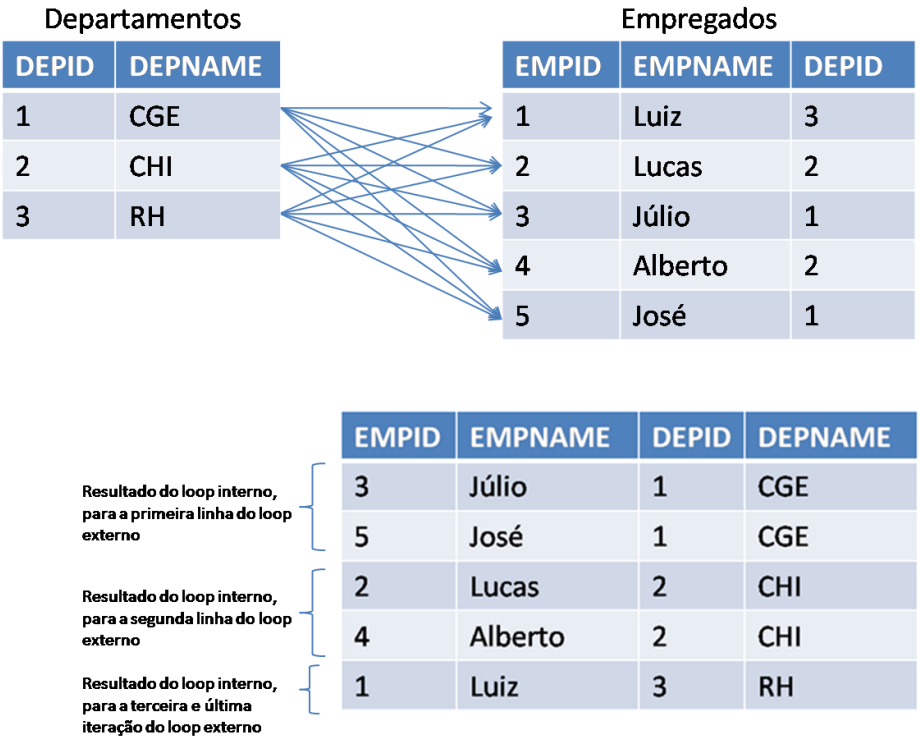


Figura 4: Funcionamento da Junção por Laço Aninhado

A ordem dos laços do algoritmo JLA, de qual tabela será a do laço externo e qual tabela será a do laço interno nem sempre estará na mesma ordem em que as tabelas aparecem na cláusula *from* da consulta. Quem toma essa decisão é o otimizador de consultas do SGBD, que leva em conta principalmente qual delas fazem melhor uso de índices, para então ser a tabela do laço interno e facilitar a pesquisa de cada iteração do laço externo e, além disso, as decisões de diferentes SGBDs variam.

SQLite (Owens, 2006), além de verificar os índices, decide a melhor tabela para ser a do laço interno, escolhendo a tabela em que o custo para acessá-la nunca será reduzido (ou seja, nunca diminuirá sua cardinalidade) se esperar outras tabelas serem processadas primeiro. Esse teste é feito primeiro assumindo que a primeira tabela da cláusula *from* é a do laço interno, e acha seu plano de execução. Então checa se este plano de execução usa qualquer outro termo da cláusula *from* que está "*notReady*", ou seja, ainda não foi executado. Se não for encontrado nenhum *notReady*, esta tabela é a melhor opção para pertencer ao laço interno. É também verificado se existem cláusulas *where* referentes às tabelas participantes das junções, e a tabela com o *where* mais restritivo pode ser melhor opção para se tornar a tabela do laço externo, uma vez que se não respeitada à condição do *where*, não é necessário continuar no laço interno.

De um modo geral, as características de indicação da utilização da JLA são:

- aplicações onde as linhas resultantes precisam ser retornadas rapidamente para visualização. JLA pode retornar as linhas resultantes assim que elas são mescladas;
- quando a segunda tabela da junção pode ser mesclada em uma baixa cardinalidade;
- quando a tabela do laço externo tem uma condição *where* bem restritiva;

O custo da JLA pode ser dado abaixo pela fórmula (Ramakrishnan, 2003):

Custo = custo de acesso da TABELA_EXTERNA + (custo de acesso da TABELA_INTERNA * número de linhas da TABELA_EXTERNA)

4. A OTIMIZAÇÃO

Muitas vezes são feitas consultas com tabelas pequenas, o que torna a JLA o método mais eficiente nesse caso. Mas ele é o método com pior complexidade para execução. Por exemplo, considere uma tabela Empregados com 500 linhas e uma tabela Computadores com 100 linhas e uma consulta solicitando os dados dos empregados que possuam computador de uma determinada marca:

```
SELECT *
FROM EMPREGADOS A
JOIN COMPUTADORES B
ON A.IDEMP = B.IDEMP
WHERE B.MARCA = 'HP';
```

Considere que o sistema escolheu a JLA para executar esta consulta, e a tabela Empregados pertencerá ao laço externo e a tabela Computadores ao laço interno, e que a quantidade de empregados que possuem um computador HP é apenas um. O laço aninhado gerado será da seguinte forma:

```
PARA CADA LINHA DA TABELA EMPREGADOS: (500 linhas)
     PARA CADA LINHA DA TABELA COMPUTADORES: (100 linhas)
          SE MARCA = 'HP' FAÇA O JOIN E RETORNE A LINHA
```

Mesmo que o único empregado que satisfaça as condições da busca seja encontrado na primeira iteração do laço interno, a JLA irá executar mais 499 laços mesmo que eles não retornem nenhum resultado. O objetivo deste trabalho é diminuir o número dessas iterações já que, uma vez que não terá mais tuplas resultantes, a execução do restante das iterações será desnecessária.

Seria muito útil conhecer a quantidade de linhas nas tabelas que respeitam a condição definida na cláusula *where* para suas colunas antes de ser feito qualquer tipo de junção. Com os outros tipos de métodos de junção é impossível fazer isto sem que haja degradação do tempo de execução da consulta, já que seria necessária uma varredura adicional de toda a tabela buscando todos os valores que se igualam ao valor definido na condição da cláusula *where*, antes de fazer a junção.

Como a JLA sempre faz uma varredura completa da tabela do laço interno, é interessante aproveitar, na primeira iteração, para verificar em cada linha se as colunas referentes às colunas participantes da cláusula *where* dessa tabela do laço interno, respeitam a condição definida. Se sim, é incrementado um contador criado especificamente para contar quantas tuplas, que respeitam a cláusula *where*, a tabela tem. Uma vez que a primeira iteração do laço interno foi completamente executada e já se tem a quantidade de registros que podem ser retornados, toda vez que uma nova ocorrência for novamente encontrada nas próximas iterações, este contador será decrementado e assim, ao chegar à zero já se pode parar a execução do laço aninhado, pois não haverá mais linhas resultantes.

Os pré-requisitos desta otimização são que em cada iteração do laço interno se tenha facilmente acesso aos valores de quaisquer colunas da tabela, e não apenas a coluna da condição da junção, para que o custo das comparações em todas as linhas não ultrapasse o custo de todos os laços desnecessários. Além de que a coluna participante da junção no laço interno deve ser não nula. O novo algoritmo seria então:

```
INT CONTADOR = 0;
PARA CADA LINHA DA TABELA A {
      IF(CONDIÇÕES DA CLÁUSULA WHERE PARA A OK){
            PARA CADA LINHA DA TABELA B{
                  IF (PRIMEIRO LOOP DE A){
                        PARA TODAS CONDIÇÕES DA CLÁUSULA WHERE PARA B {
                              IF(VALOR COLUNAS RESPEITAM A CONDIÇÃO)
                                    CONTADOR += 1;
                        }
                        IF(CONDIÇÃO JOIN E WHERE OK){
                              CONTADOR -= 1;
                              FAÇA O JOIN E RETORNE A LINHA
                        }

                  }
                  ELSE{
                        IF(CONDIÇÕES JOIN E WHERE OK){
                              CONTADOR -= 1;
                              FAÇA O JOIN E RETORNE A LINHA
                              IF(CONTADOR == 0)
                                    PARAR EXECUÇÃO

                        }
                  }

            }
      }
}
```

Não foi estimado o custo do novo algoritmo uma vez que ele dependerá da quantidade de incidências contadas pelo contador e aonde (tupla da tabela) elas se encontram. Assim, o otimizador deve optar pelo novo algoritmo para ser usado no lugar da JLA básica, a qual o custo é próximo ao do pior caso da alteração proposta.

Quando a coluna da tabela do laço interno que participa na condição da junção possui índice, este novo algoritmo não é vantajoso uma vez que, com esse recurso, o SQLite não precisa fazer uma varredura completa na tabela, e a pesquisa se torna muito mais eficiente. Mas este trabalho se dedicou à otimização da JLA considerando que as colunas participantes da condição da junção podem muitas vezes não estar indexadas. Nesta última abordagem, o SQLite possui duas situações em que serão estudados os efeitos das alterações que foram explicadas nesta seção.

4.1. Execução sem uso do catálogo

O SQLite possui um catálogo interno em que suas informações (estatísticas sobre os dados nas tabelas, cardinalidade, etc.) são armazenados em uma tabela chamada *sqlite_stat*. Esta tabela pode ser atualizada com informações reais a partir do comando *analyze*. Essas informações são úteis ao otimizador de consultas, pois influem na decisão do melhor plano de execução para determinada consulta.

Uma vez que o catálogo não esteja sendo usado, ou seja, o comando *analyze* nunca foi executado, o SQLite ao executar o laço aninhado, cria uma tabela temporária, e varre uma primeira vez toda a tabela do laço interno para indexar suas colunas. Este processo é importante, pois assim que acabar de varrer a tabela, a pesquisa será feita como se a tabela tivesse índices em suas colunas, o que torna a pesquisa muito mais eficiente.

Apesar de não varrer mais a tabela interna do laço aninhado após a indexação, o SQLite terá percorrido esta tabela uma única vez, e é nesta leitura que será coletada informações de quantas ocorrências se tem de igualdade com o que foi definido nas condições da cláusula *where* para esta tabela. Apesar da capacidade de melhoria do desempenho ser bem menor que se não houvesse esta indexação, pois o laço interno só é executado uma vez, ainda é um estudo importante.

4.2. Execução com catálogo contendo informações desatualizadas

O comando *analyze*, se executado sozinho, irá atualizar as estatísticas das tabelas e dos índices, como a cardinalidade, e salvar estes dados na tabela oculta *sqlite_stat*. Uma vez que este comando tenha sido executado, o SQLite confia que o usuário atualizará as estatísticas com o *analyze* com freqüência.

Quando a consulta é otimizada, caso o *sqlite_stat* informe que a tabela escolhida para ser do laço interno possua poucos registros (ou seja, abaixo de 50 registros, como foi observado nos testes realizados), o SQLite não se dará o trabalho de indexar a tabela do laço interno na primeira repetição e executará um laço aninhado completo, ou seja, para cada linha da tabela do laço externo, varrerá todas as linhas da tabela do laço interno para obter o resultado. Esta decisão é tomada, pois no caso de poucas linhas o custo de indexá-las pode ser maior do que executar o laço aninhado completo.

Desse modo, se o catálogo do sistema for atualizado quando a tabela do laço interno tiver poucas linhas e nunca mais for atualizado mesmo que tenham sido inseridos muitos outros registros nesta tabela, o sistema irá executar um laço aninhado completo, o que dependendo da quantidade de linhas nas tabelas pode ter um péssimo desempenho. Este laço aninhado completo é o algoritmo mais simples de JLA e sua otimização pode ser benéfica em outros bancos de dados. Uma vez que não há mais indexação automática, a quantidade de incidências da cláusula *where* não será mais obtida na iteração do laço de indexação, e sim na primeira iteração de execução da tabela mais interna.

5. Testes e Análise de Resultados

Para realização dos testes foram usados três modelos de dados em três consultas diferentes. Em nenhum deles foram configurados índices para as tabelas nas colunas participantes da junção, pois o estudo deste trabalho se dedica apenas ao funcionamento da JLA e para sua execução básica é necessário desconsiderar índices. O primeiro modelo com sua estrutura (Figura 5), e sua respectiva consulta, foram construídos de modo que a única cláusula *where* referente à tabela do laço interno fosse de uma coluna do tipo *char*.

Figura 5: Primeira estrutura de banco de dados utilizada para testes

```
Select *
from empregados A
join computadores B
on A.idemp = B.idemp
where A.area = 1 and B.marca = 'hp';
```

Após analisar o funcionamento do otimizador, esta consulta foi criada de modo que a tabela Computadores seja sempre escolhida para pertencer ao laço interno na execução, para que se tenha a certeza durante os testes, que a ordem das tabelas nos laços sempre será a mesma.

O segundo modelo, possui a estrutura apresentada na Figura 6. A consulta representada a seguir foi criada objetivando que a cláusula *where* referente à tabela do laço mais interno seja de uma coluna do tipo *integer* para comparar o ganho de desempenho com o do primeiro modelo.

Figura 6: Segunda estrutura de banco de dados utilizada para testes

```
Select *
from clientes A
join compras B
on A.idcli = B.idcli
where A.codest = 4 and B.idpro = 43;
```

Esta consulta foi criada de modo que o otimizador sempre escolha a mesma tabela (compras) para pertencer ao laço interno na execução.

A terceira estrutura (Figura 7) e consulta representadas abaixo foram criadas para testar quando a tabela do laço interno tiver duas cláusulas *where* ambas referentes à colunas do tipo char. Este é o pior caso dos testes, pois a comparação de *strings* é mais demorada.

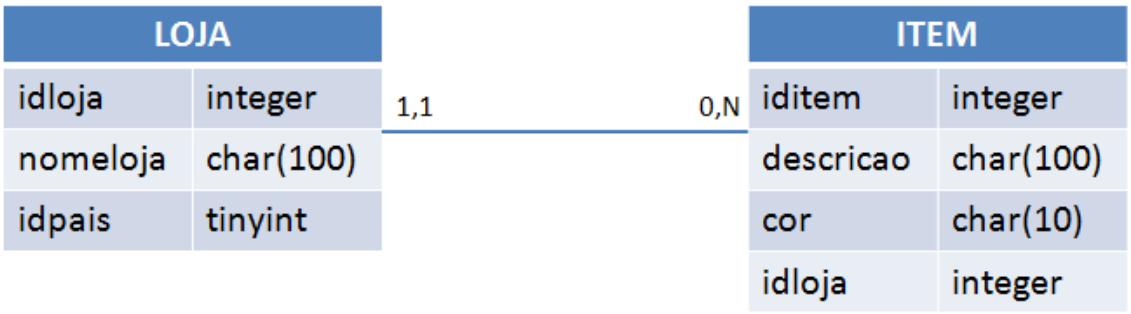


Figura 7: Terceira estrutura de banco de dados utilizada nos testes

```
Select *
from loja A
join item B
on A.idloja = B.idloja
where A.pais = 55 and A.nomeloja = 'Walmart'
and B.descricao = 'TV' and B.cor = 'preta';
```

Esta consulta foi criada de modo que o otimizador escolhesse sempre a mesma tabela (item) para pertencer ao laço interno na execução.

Este trabalho aborda duas opções de execução, sem uso do catálogo, e com uso do catálogo desatualizado. O teste com o catálogo atualizado não se faz necessário no SQLite pois desse modo o sistema não usará o algoritmo de JLA, e sim uma busca indexada que não se beneficiaria do algoritmo proposto.

Para ser possível demonstrar como a alteração afeta os dois tipos de execução, o volume de dados utilizado em cada uma das estruturas, foi de cinco, dez, cinqüenta, cem, duzentas, quatrocentas, oitocentas, mil e seiscentos, três mil e duzentas, e seis mil e quatrocentas linhas em ambas as tabelas pertencentes à junção, onde cada linha de uma tabela corresponde uma diferente tupla da outra tabela. Todas as consultas criadas retornarão apenas duas tuplas resultantes e a posição delas na tabela do laço interno da JLA, serão as duas primeiras no melhor caso, as duas do meio no caso médio e as duas últimas no pior caso. Desta forma, é possível analisar melhor quantas iterações foram poupadas da execução, e o ganho de tempo com isso. A seguir são analisados os resultados para os dois tipos de execução, sem uso do catálogo e com uso de um catálogo desatualizado.

5.1. Testes com execução sem uso do catálogo

O catálogo do sistema do SQLite, que é armazenado na tabela oculta *sqlite_stat*, não será usado caso o comando *analyze* nunca tenha sido executado. Este teste é necessário, pois esta não é uma situação incomum e é importante avaliar como a alteração o afetará. A Tabela 1 apresenta o número de iterações (apenas do laço externo, pois neste caso, o laço interno será iterado sempre apenas uma vez para a indexação) contabilizadas para as três estruturas testadas executadas no programa original e no programa alterado no melhor, pior e caso médio.

Tabela 1: Quantidade de iterações do laço externo, sem uso do catálogo

Registros	Código alterado (melhor caso)	Código alterado (caso médio)	Código alterado (pior caso) e código original
5	2	3	5
10	2	6	10
50	2	26	50
100	2	51	100
200	2	101	200
400	2	201	400
800	2	401	800
1600	2	801	1600
3200	2	1601	3200
6400	2	3201	6400

Como se pode analisar, o número de iterações do laço externo no melhor caso é constante no código alterado: dois, pois são os dois primeiros registros da tabela "computadores". Com a otimização proposta, como já se achou todos os resultados possíveis, a execução para. Já no código original a execução continua e termina de verificar todos os registros da tabela do laço externo.

A seguir são apresentados os tempos de execução medidos para as três estruturas testadas.

5.1.1. Primeira consulta

A Tabela 2 exibe os tempos de execução da primeira consulta no melhor, pior e caso médio, comparados com o código original, que, como apresentou diferenças muito pequenas em todos os testes sem catálogo, colocamos o tempo de execução médio. O tempo de execução foi medido em 10^{-4} segundos.

Tabela 2: Tempo de execução da primeira consulta sem catálogo

Registros	**Código Original** (10^{-4}s)	**Código Alterado** (10^{-4}s)		
		Melhor Caso	Caso Médio	Pior Caso
5	2	3	4	4
10	3	4	5	5
50	5	5	5	5
100	8	6	7	8
200	15	11	17	17
400	31	24	29	31
800	61	51	69	69
1600	138	106	132	147
3200	291	227	278	309
6400	507	375	543	553

A Figura 8 ilustra o ganho de desempenho em cada um dos casos, que foi calculado com o cálculo de *speedup*, tempo de execução antes da alteração dividido pelo tempo de execução depois da alteração.

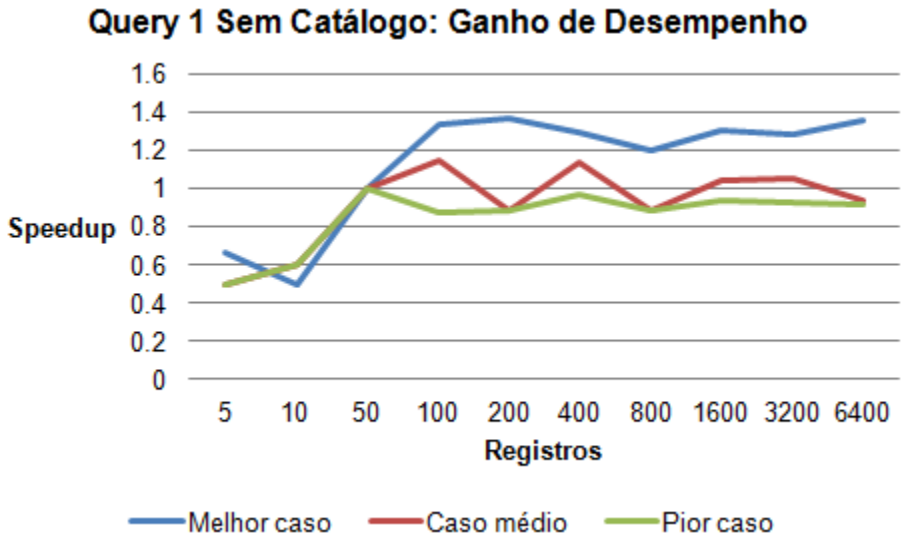


Figura 8: Ganho de desempenho da primeira consulta.

Na Figura 8, é possível interpretar que no melhor caso, apesar do número de iterações da repetição ser drasticamente menor, começa a ganhar desempenho a partir de cinqüenta linhas (no cálculo de *speedup* o ganho de desempenho começa a partir de um). Além disso, a execução fica mais rápida a medida que cadastramos mais dados nas tabelas. No caso médio houve uma intercalação no desempenho dos dois algoritmos e no pior caso o algoritmo alterado teve desempenho pior que o original em todas as situações.

Quando o catálogo não é utilizado, é feito apenas uma iteração na tabela do laço interno. Logo, o ganho de desempenho vem apenas das iterações não executadas da tabela externa, que uma vez que se tenha encontrado todos os resultados, não são mais necessárias.

Apesar de economizar tempo com os laços externos não executados, nos testes do melhor caso analisou-se que o tempo de execução do código alterado foi maior que o código original, quando as tabelas possuem menos de cinqüenta registros. O motivo disso foi que o tempo gasto pelo algoritmo no primeiro e único laço interno foi maior que o tempo salvo dos laços externos não executados. Isso acontece, pois além de fazer a indexação que o código original faz no primeiro laço interno, o código alterado compara os valores da cláusula *where* correspondentes à tabela sendo indexada para contar a quantidade de incidências, o que resulta no código alterado sempre gastar mais tempo no primeiro laço interno que o código original. O tempo gasto é ainda maior se a coluna a ser analisada for do tipo *char*, pois a comparação de *char* é mais trabalhosa que a comparação de *integer*, por exemplo. Logo, só há ganho de desempenho se o tempo salvo das iterações externas não executadas, for maior que o tempo a mais gasto no

primeiro laço interno, o que, como testado, só ocorreu quando houve mais de cinqüenta linhas nas tabelas.

5.1.2. Segunda consulta

Na segunda consulta a cláusula *where* correspondente ao laço interno será de uma coluna do tipo *integer*. O tempo de comparação de um *integer* é muito menor que de *strings*, e por isso se esperava um maior ganho de desempenho. A Tabela 3 mostra o tempo de execução da primeira consulta no melhor, pior e caso médio, comparados com o código original. Já a Figura 9 ilustra o ganho de desempenho em cada um dos casos.

Tabela 3: Tempo de execução da segunda consulta sem catálogo

Registros	**Código Original** (10^{-4}s)	**Código Alterado** (10^{-4}s)		
		Melhor Caso	Caso Médio	Pior Caso
5	2	2	2	2
10	2	2	2	2
50	4	4	5	4
100	7	7	7	6
200	13	12	13	12
400	29	27	29	32
800	55	51	60	53
1600	106	110	112	111
3200	223	221	229	225
6400	456	455	462	466

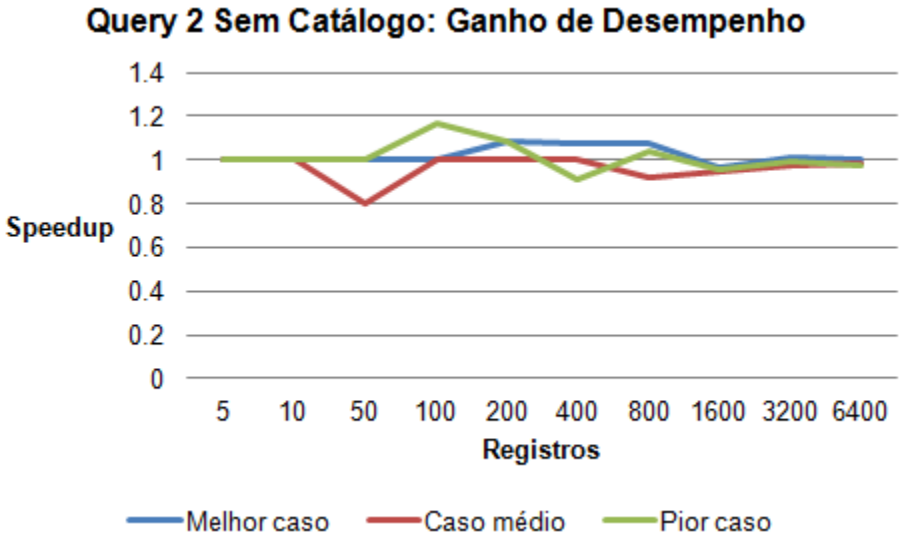


Figura 9: Ganho de desempenho da segunda consulta

Como se pode analisar, a segunda consulta teve um ganho de desempenho pior que a primeira. Apesar de variações, a tendência foi manter o mesmo tempo de execução que o código original, o que talvez torne a alteração não vantajosa neste caso.

Há dois motivos para esse resultado. O primeiro é que não houve perda de desempenho, pois a comparação de *integer* é muito simples, e ao contrário da comparação de *char* não é nada custosa. Isto garante pelo menos que o novo algoritmo não gastará mais tempo que o algoritmo original. O segundo é que do mesmo modo que a comparação de *integer* no código alterado é muito rápida, ela também é no código original, desse modo a execução já é muito rápida e por isto a alteração não foi capaz de aperfeiçoá-la.

5.1.3. Terceira consulta

Nessa consulta há duas expressões na cláusula *where* referentes a duas colunas do tipo *char* na tabela do laço interno da JLA. A Tabela 4 mostra o tempo de execução da primeira consulta no melhor, pior e caso médio. Já a Figura 10 ilustra o ganho de desempenho em cada um dos casos.

Tabela 4: Tempo de execução da terceira consulta sem catálogo

Registros	**Código Original** (10^{-4}s)	**Código Alterado** (10^{-4}s)		
		Melhor Caso	Caso Médio	Pior Caso
5	2	2	2	2
10	2	3	3	3
50	4	5	5	6
100	10	8	9	11
200	17	15	18	21
400	34	31	36	43
800	70	61	74	87
1600	147	126	150	178
3200	304	250	323	375
6400	612	518	636	766

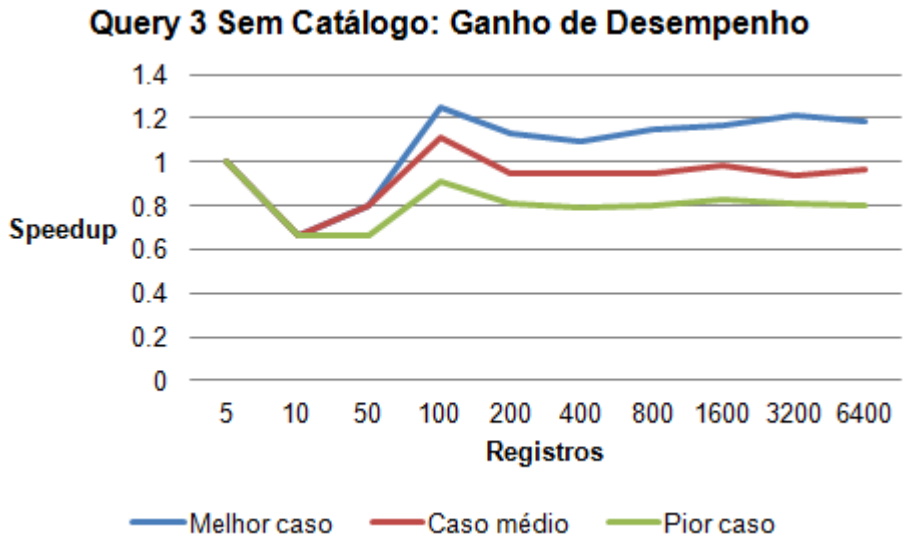


Figura 10: Ganho de desempenho da terceira consulta

A alteração nesse caso, portanto, não foi vantajosa uma vez que a quantidade de ganho de desempenho no melhor caso foi a mesma quantidade de perda de desempenho no pior caso. Isso ocorreu porque nesse caso, o conjunto de comparações de *char* gastou mais tempo que as comparações das outras consultas testadas.

5.2. Testes com catálogo contendo informações desatualizadas

Com o catálogo desatualizado contendo informações de que a tabela do laço interno possui poucas linhas, o SQLite será forçado a fazer uma JLA completa sem indexação, o que é muito mais custoso ao sistema. Esta situação é a que mais se beneficia do código alterado. A Tabela 5 apresenta o número de iterações do laço interno somado com o número de iterações do laço externo, realizados com o código alterado no melhor, pior e caso médio, para todas as consultas.

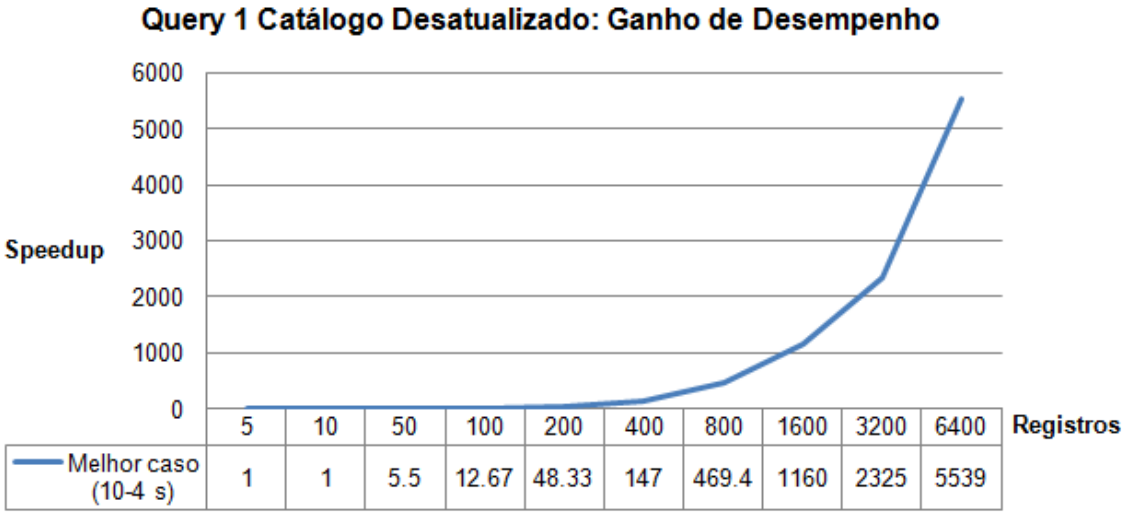
Tabela 5: Quantidade de iterações do laço interno com uso do catálogo desatualizado

Registros	Código alterado (melhor caso)	Código alterado (caso médio)	Código alterado (pior caso) e código original
5	8	9	30
10	13	17	110
50	53	77	2550
100	103	152	10100
200	203	302	40200
400	403	602	160400
800	803	1202	640800
1600	1603	2402	2561600
3200	3203	4802	10243200
6400	6403	9602	40966400

A seguir será analisado o ganho de desempenho para cada um dos modelos de banco de dados montados.

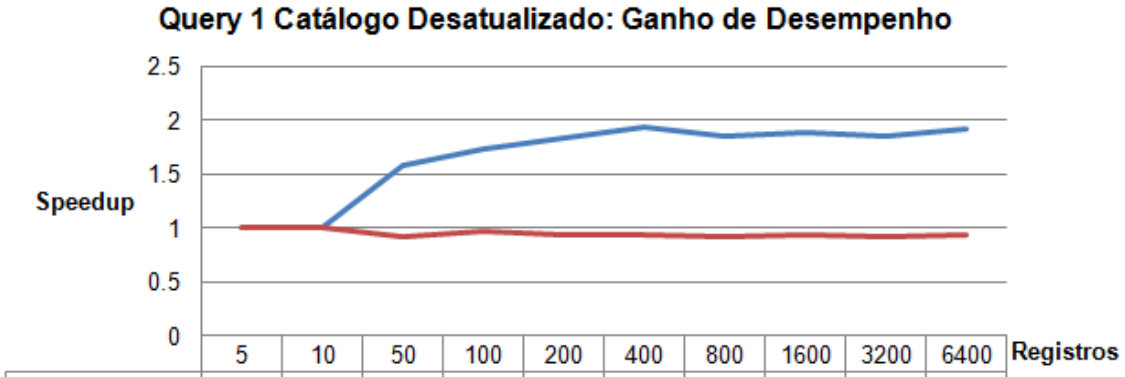
5.2.1. Primeira consulta

Para a primeira consulta, a Figura 11 ilustra o ganho de desempenho do melhor caso, já na Figura 12 é mostrado o ganho de desempenho do caso médio, e pior caso. Como em todos testes, o ganho de desempenho é calculado pelo tempo de execução do código original dividido pelo tempo de execução com o código alterado.



	5	10	50	100	200	400	800	1600	3200	6400
Melhor caso (10-4 s)	1	1	5.5	12.67	48.33	147	469.4	1160	2325	5539

Figura 11: Ganho de desempenho na primeira consulta – melhor caso



	5	10	50	100	200	400	800	1600	3200	6400
Caso médio (10-4 s)	1	1	1.571	1.727	1.835	1.941	1.851	1.883	1.843	1.915
Pior caso (10-4 s)	1	1	0.917	0.974	0.942	0.941	0.917	0.941	0.922	0.945

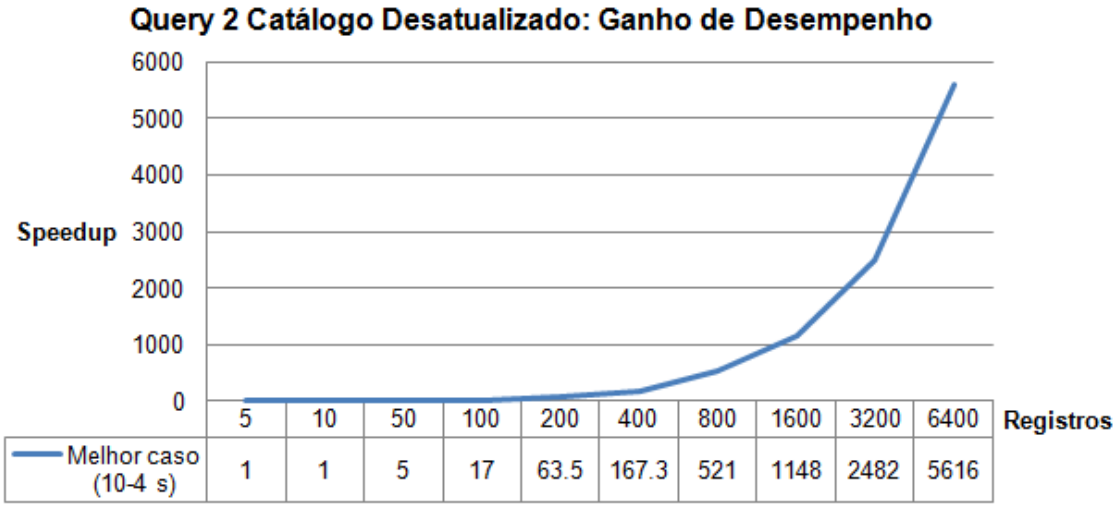
Figura 12: Ganho de desempenho na primeira consulta - caso médio e pior caso

Como se pode analisar, o ganho de desempenho quando o catálogo está desatualizado é certo. No melhor caso, representado pela Figura 11 a melhora foi enorme. No caso médio, o código alterado rodou quase duas vezes mais rápido que o código original, e no pior caso, houve uma perda não significativa de desempenho o que

torna a alteração muito favorável nesta abordagem. Em todas outras duas consultas testadas notou-se o mesmo comportamento, como mostrado a seguir.

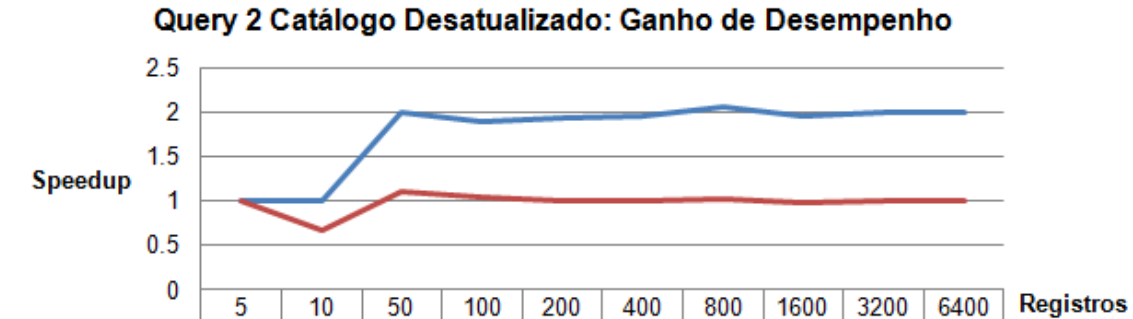
5.2.2. Segunda consulta

A Figura 13 representa o ganho de desempenho do melhor caso, e a Figura 14 ilustra o ganho de desempenho do caso médio, e pior caso.



	5	10	50	100	200	400	800	1600	3200	6400
Melhor caso (10-4 s)	1	1	5	17	63.5	167.3	521	1148	2482	5616

Figura 13: Ganho de desempenho na segunda consulta - melhor caso

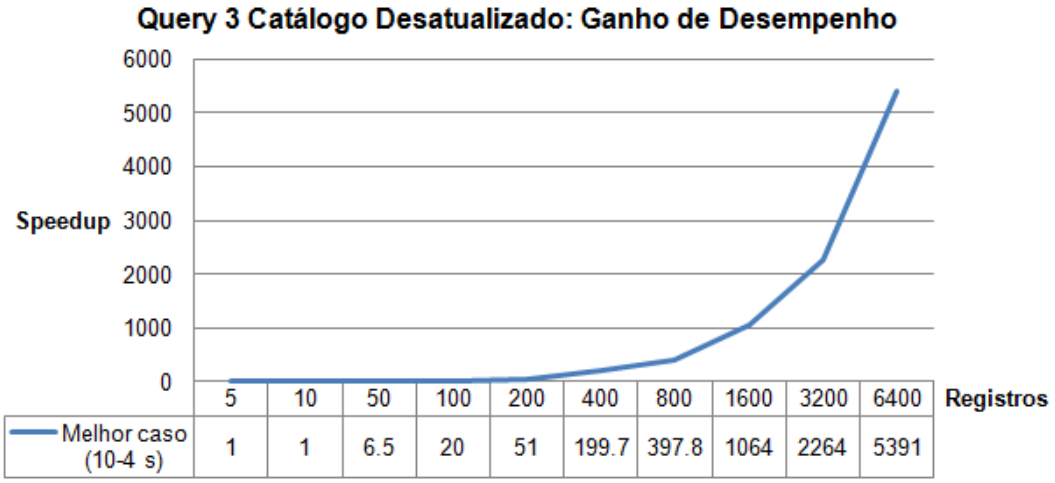


	5	10	50	100	200	400	800	1600	3200	6400
Caso médio (10-4 s)	1	1	2	1.889	1.924	1.953	2.065	1.959	1.992	1.985
Pior caso (10-4 s)	1	0.667	1.111	1.03	1	0.988	1.028	0.982	0.994	0.99

Figura 14: Ganho de desempenho na segunda consulta - caso médio e pior caso

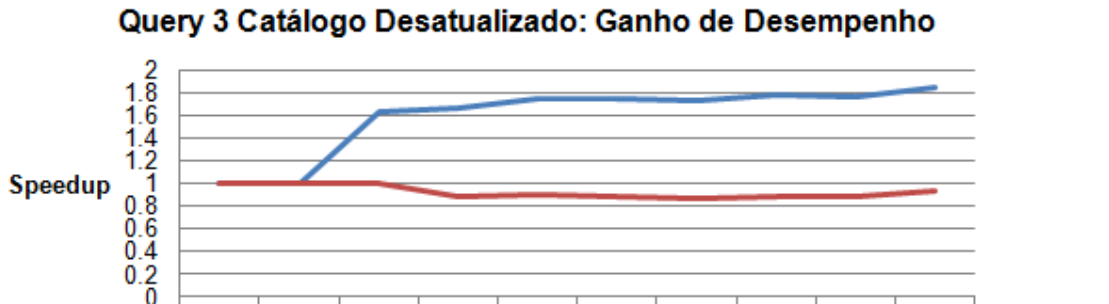
5.2.3. Terceira consulta

A Figura 15 representa o ganho de desempenho do melhor caso, e a Figura 16 ilustra o ganho de desempenho do caso médio, e pior caso.



	5	10	50	100	200	400	800	1600	3200	6400
Melhor caso (10-4 s)	1	1	6.5	20	51	199.7	397.8	1064	2264	5391

Figura 15: Ganho de desempenho na terceira consulta - melhor caso



	5	10	50	100	200	400	800	1600	3200	6400
Caso médio (10-4 s)	1	1	1.625	1.667	1.739	1.736	1.721	1.773	1.759	1.848
Pior caso (10-4 s)	1	1	1	0.889	0.89	0.873	0.866	0.883	0.882	0.926

Figura 16: Ganho de desempenho na terceira consulta - caso médio e pior caso

A alteração foi portanto vantajosa em todas as consultas quando o catálogo está desatualizado.

6. Conclusão

Este trabalho apresentou uma maneira de como a coleta de estatísticas em tempo de execução pode ser usada para implementar uma melhoria no método JLA. Esse método, por apresentar em várias situações um baixo desempenho, e estar presente na maioria SGBDs, justifica um estudo sobre seu aperfeiçoamento.

A otimização proposta neste trabalho foi testada em duas situações no SQLite: com uso do catálogo desatualizado e sem uso do catálogo. Quando o catálogo está desatualizado, informando que há poucos registros nas tabelas, o sistema executa uma JLA simples. Nesse caso, observou-se que a alteração proposta na JLA foi benéfica. Apesar dessa não ser uma situação muito comum no SQLite, e uma alteração da decisão de quando se deve usar a auto-indexação ser mais interessante, deve-se levar em conta que o SQLite foi desenvolvido para executar apenas a JLA e seu algoritmo já é bem

otimizado em relação a outros SGBDs. Logo, bancos de dados mais completos, como MySQL e Oracle, que usam com mais freqüência a JLA simples, se beneficiariam ainda mais.

Já na situação em que não há uso de catálogo, na primeira iteração na tabela do laço interno, o sistema indexa todos registros e então faz uma busca indexada em vez de usar o algoritmo simples de JLA. Foi analisado nesse caso que o novo algoritmo não é vantajoso uma vez que o custo acrescido da primeira iteração do laço interno pode facilmente degradar o tempo de resposta da consulta, caso haja mais condições na cláusula *where*.

É de interesse, em trabalhos futuros, realizar testes com mais condições na cláusula *where* para a tabela do laço interno da JLA e estudar a implementação do algoritmo proposto em outros SGBDs, avaliando se o ganho de desempenho obtido será satisfatório. No caso do SQLite a melhoria foi significativa quando o catálogo do sistema estava desatualizado, pois somente assim usa-se o JLA. Para implementação em outros SGBDs é importante estudar em quais situações o sistema usa a JLA.

Referências Bibliográficas

Chaudhuri, S. a. (1994). Including Group-By in Query Optimization. *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases* (pp. 354-366). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Chen, S. a. (2007). Improving hash join performance through prefetching. *ACM Trans. Database Syst.*

Deshpande, A. a. (2007). Adaptive query processing. *Found. Trends databases* , 1-140.

Hemalatha, G. a. (2010). Optimization of joins using random record generation method. *A2CWiC '10: Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India* (pp. 28:1-28:6). Coimbatore, India: ACM.

Kabra, N. a. (1998). Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data* (pp. 106-117). Seattle, Washington, United States: ACM.

Oracle Corporation and/or its affiliates. (1997). *Nested-Loop Join Algorithms*. Acesso em 15 de 05 de 2011, disponível em MySQL Documentation: http://dev.mysql.com/doc/refman/5.0/en/nested-loop-joins.html

Owens, M. (2006). *The Definitive Guide to SQLite (Definitive Guide).* Berkely, CA, USA: Apress.

Ramakrishnan, R. a. (2003). *Database Management Systems.* New York, NY, USA: McGraw-Hill, Inc.

Selinger, P. G. (1979). Access path selection in a relational database management system. *Proceedings of the 1979 ACM SIGMOD international conference on Management of data* (pp. 23-34). Boston, Massachusetts: ACM.