UMA PROPOSTA DE ALGORITMO PARA A FRAGMENTAÇÃO VIRTUAL ADAPTATIVA

Rafael Soares Sampaio

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Geraldo Bonorino Xexéo
Alexandre de Assis Bento Lima

Rio de Janeiro
Junho de 2023

UMA PROPOSTA DE ALGORITMO PARA A FRAGMENTAÇÃO VIRTUAL ADAPTATIVA

Rafael Soares Sampaio

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Geraldo Bonorino Xexéo
Alexandre de Assis Bento Lima

Aprovada por: Prof. Geraldo Bonorino Xexéo
Prof. Alexandre de Assis Bento Lima
Prof. Geraldo Zimbrão da Silva
Prof. Daniel Cardoso Moraes de Oliveira

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2023

Sampaio, Rafael Soares

Uma Proposta de Algoritmo para a Fragmentação Virtual Adaptativa/Rafael Soares Sampaio. – Rio de Janeiro: UFRJ/COPPE, 2023.

XIV, 100 p.: il.; 29, 7cm.

Orientadores: Geraldo Bonorino Xexéo

Alexandre de Assis Bento Lima

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2023.

Referências Bibliográficas: p. 70 – 72.

1. banco de dados. 2. processamento de consultas paralelo. 3. adaptive virtual partitioning. 4. AVP. 5. multicore. I. , Geraldo Bonorino Xexéo *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Agradeço ao meu orientador Alexandre de Assis Bento Lima pela valiosíssima orientação e suporte, todo o seu conhecimento, experiência e competência me colocaram à frente para a execução deste trabalho. Ele soube ensinar quando era pra ensinar, cobrar quando era pra cobrar, e estimular quando era pra estimular.

Agradeço ao meu orientador Geraldo Bonorino Xexéo pelo incentivo e orientação, foi com poucas palavras que ele desfez pressupostos que me impediam de seguir adiante e me convenceu, pela primeira vez, que eu possuía um trabalho mínimo apresentável.

Agradeço a todos os funcionários e àqueles que trabalham e se dedicam para manter a estrutura da Universidade Federal do Rio de Janeiro funcionando, estrutura da qual eu me beneficiei. Em particular, agradeço a todos os professores e funcionários do Programa de Engenharia de Sistemas e Computação (PESC) da COPPE, que me possibilitaram ter essa experiência engrandecedora de ver como a ciência é feita (e fazer um pouco também), entender os caminhos da pesquisa e os constituintes que compõem o interesse científico, esse aprendizado levarei para sempre.

Agradeço ao Núcleo Avançado de Computação de Alto Desempenho (NACAD) pela disponibilização de recursos tão essenciais para a elaboração do presente trabalho e pela atenção do corpo técnico quando necessário o suporte.

Agradeço à CAPES pelo apoio ao presente trabalho.

Não poderia deixar de agradecer à minha família, que têm sido a escada da minha vida e me ajudado em tantos momentos, com participação em todas as minhas conquistas até hoje.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA PROPOSTA DE ALGORITMO PARA A FRAGMENTAÇÃO VIRTUAL ADAPTATIVA

Rafael Soares Sampaio

Junho/2023

Orientadores: Geraldo Bonorino Xexéo
Alexandre de Assis Bento Lima

Programa: Engenharia de Sistemas e Computação

A Fragmentação Virtual Adaptativa (AVP) é um algoritmo para processamento paralelo de consultas OLAP em *clusters* de computadores com memória distribuída. Nesta dissertação, são propostos dois algoritmos de processamento de consulta *multicore* que contribuem para a evolução da AVP: AVPMSimples e AVPMRandômica. A abordagem é inspirada na própia AVP, com a criação de subpartições virtuais – definidas por predicados de consultas – a serem processadas por *cores* individuais. Enquanto a AVPMSimples se apresenta como algoritmo ingênuo e adota tamanhos fixos para as subpartições, a AVPMRandômica é adaptativa, buscando de forma randômica por um tamanho de subpartição ótimo. Os algoritmos foram acrescentados à implementação padrão da AVP, o ParGRES, e foram testados em uma base de dados do *benchmark* TPC-H com aproximadamente 300 GBytes. Observou-se que a AVPMRandômica tende a se comportar como AVPMSimples, o que indica que esta é uma boa solução, mais simples do que a primeira e com mesmo desempenho. A utilização de múltiplos *cores* proporcionou aceleração no processamento de algumas consultas apenas. A razão pela qual os algoritmos propostos não aceleram nas outras consultas foi constatada como sendo a não aceleração da fase de planejamento das consultas, cujo tempo em geral domina a fase de execução para pequenas subpartições.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A ALGORITHM PROPOSAL FOR ADAPTIVE VIRTUAL PARTITIONING

Rafael Soares Sampaio

June/2023

Advisors: Geraldo Bonorino Xexéo
Alexandre de Assis Bento Lima

Department: Systems Engineering and Computer Science

AVP is an algorithm for parallel processing of OLAP queries in computer clusters with shared nothing architecture. This dissertation proposes two multicore query processing algorithms that contributes to AVP evolution: AVPMSimple and AVPMRandom. The new approach is inspired by AVP itself, with the creation of virtual subpartitions – defined by query predicates – to be processed by individual cores. While AVPMSimple is a naive algorithm that adopts fixed sizes subpartitions, AVPMRandom is adaptive, randomly searching for an optimal subpartition size. Both algorithms were added to ParGRES, the default AVP implementation, and tested against a TPC-H benchmark database of approximately 300 GB. Experimental evaluation shows that AVPMRandom tends to behave like AVPMSimples, indicating that the later is a good solution as it is simpler than the former and presents the same performance. Just a few queries used during the experiments benefits from the use of multiple cores. The reason why the proposed algorithms did not accelerate all queries is the lack of acceleration during the query planning phase, which generally is significantly longer than the execution phase time for small subpartitions.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

AVPMRandom	AVP Multicore Randômica, p. 25
AVPMSimples	AVP Multicore Simples, p. 25
AVP	Adaptive Virtual Partitioning, p. 2
AWS	Amazon Web Services, p. 3
BME	Statistical Multidimensional Database, p. 11
CPU	Central Processing Unit, p. 15
DBA	Database Administrator, p. 25
DQS	Distribuited Query Service, p. 11
ETL	Extract, transform, load, p. 1
FS	Factory Service, p. 11
IBGE	Instituto Brasileiro de Geografia e Estatística, p. 11
IPC	Inter-process Comunication, p. 14
JDBC	Java Database Connectivity, p. 5
NACAD	Núcleo Avançado de Computação de Alto Desempenho, p. 40
NQP	Node Query Processor, p. 38
NUMA	Non-Unified Memory Access, p. 2
OLAP	Online Analytical Processing, p. 1
OLTP	Online Transaction Processing, p. 1
PC	Personal Computer, p. 5
SGBD	Sistema de Gerência de Banco de Dados, p. 1

SINAPAD	Sistema Nacional de Processamento de Alto Desempenho, p. 40
SMT	simultaneous multithreading, p. 20
SM	Shared Memory, p. 17
SN	Shared Nothing, p. 2
SQL	Structured Query Language, p. 5
SVP	Simple Virtual Partitioning, p. 2
TBB	Intel's Threading Building Blocks, p. 20
VP	Virtual Partitions, p. 5

Capítulo 1

Introdução

Sistemas de Gerência de Banco de Dados (SGBD) são utilizados para processar consultas que, na maioria das vezes, requerem dois tipos de processamento: transacional e analítico. As consultas que demandam processamento transacional (OLTP, sigla para *Online Transaction Processing*) são comumente encontradas em aplicações comerciais, são orientadas a transação, com alta vazão e necessitam de alto controle de dados e disponibilidade. Por outro lado, consultas que demandam processamento analítico (OLAP, sigla para *Online Analytical Processing*) são usadas para análise de tendências e previsões, necessitam de dados históricos e são realizadas por consultas complexas que fazem agregações sobre múltiplas dimensões em grandes bases de dados [1].

As consultas OLAP são frequentes em organizações com grandes quantidade de dados que precisam ou desejam obter informações, a partir do processamento destes dados, vantajosas para o negócio e elaboração de relatórios gerenciais.

Existem diversas abordagens e sistemas para processar consultas OLAP. Uma abordagem tradicional é manter os dados históricos em *data warehouses*[1] (termo que significa armazém de dados), estes sofrem um processo de extração, transformação e carga (*Extract, transform, load - ETL*) e são armazenados em um *datamart* usando um esquema lógico chamado "estrela" (além deste, existe também o esquema chamado "floco de neve").

De acordo com [2] existem dois tipos mais comuns de tecnologia OLAP, o Multidimensional OLAP é quando os dados são agregados em multiplas dimensões e carregados em um "cubo" indexado, e o Relational OLAP é quando os dados são mantidos em uma base relacional. Acrescenta-se a essas formas elaboradas uma mais simples, a de colocar todos os dados em uma tabela única com todas as dimensões como atributos, e indexar tal tabela: é uma forma muito eficiente porém dispendiosa em consumo de memória, pois gera muita redundância de dados.

Em especial, foi foco desse trabalho um algoritmo, cujo propósito é o processamento de consultas OLAP, baseado no algoritmo *Adaptive Virtual Partitioning*

(AVP) em inglês, ou em português Fragmentação Virtual Adaptativa, mas daqui em diante será utilizado o termo mais comum que é a sigla em inglês AVP . A AVP, que foi apresentada por [3], é uma evolução da *Simple Virtual Partitioning (SVP)* [4] que implementa balanço de carga e evita a varredura completa, problema que ocorria na SVP. A AVP foi feita para processamento de consultas em *clusters* de computadores com arquitetura *shared nothing (SN)* e se baseia em uma base de dados que possua uma ou mais tabelas fato e tabelas dimensão, portanto pode-se classificá-la como Relational OLAP.

A implementação principal da AVP foi realizada no ParGRES[5], que é um *middleware* que utiliza um SGBD caixa preta como sistema subjascente e implementa o processamento concorrente de consultas por meio do algoritmo AVP. Todo o trabalho de avaliação experimental foi feita a partir dessa implementação.

1.1 Motivação

A AVP foi desenvolvida originalmente em meados dos anos 2000 e não possui capacidades *multicores*. No passado, os processadores eram em sua maioria *singlecore* e havia perspectivas de aumento da velocidade de processamento. Atualmente os *chips* não evoluem mais em velocidade [6], já chegamos próximo ao limite físico, e um dos vetores de crescimento passou a ser o paralelismo nos processadores. O processamento *multicore* se tornou ubíquo nas arquiteturas AMD64, X86 e ARM para celulares. Com o objetivo de ganhar desempenho em consultas OLAP é natural que haja o estímulo de adaptar a AVP para aproveitar essa capacidade atual dos processadores, estendendo-a para um algoritmo *multicore*.

Ao se pesquisar por processamento de consultas concorrente e paralelo, a maior parte dos resultados se refere a processamento em ambiente concorrente (*e.g.* arquiteturas *Non-Unified Memory Access (NUMA)* ou *shared nothing*) e apenas poucos trabalhos se referem ao processamento paralelo multicore, ou seja, processamento executado por vários (*cores*) de um mesmo processador de forma concorrente.

Seja a situação hipotética de uma consulta que leva tempo t para ser processada. Suponha que seja executada em um *cluster* com 10 máquinas, de forma que a combinação de *hardware* e *software* proporcione uma aceleração linear, a consulta passa então a ser processada em tempo $t/10$. Suponha agora que cada máquina do *cluster* possua 4 núcleos e que o *software* obtenha aceleração linear no processamento *multicore*, o novo tempo de processamento então passa a ser $\frac{t/10}{4} = t/40$. Note que para alcançar a mesma aceleração seriam necessárias 40 máquinas no *cluster*! Se ainda considerarmos que todo os processadores atuais possuem capacidades *multicore*, a justificativa para pesquisar e utilizar técnicas de processamento *multicore* se mostra convincente.

1.2 Objetivo

Existem dois objetivos, o primário é incorporar ao já existente algoritmo AVP capacidades *multicores*, o segundo é usar o ParGRES como *middleware* que acrescenta capacidades *multicore* à qualquer SGBD, mesmo aqueles que não são executados em um *cluster*.

A AVP, e principalmente a anterior SVP, foram criados em uma época onde existiam processadores *singlecore* sendo produzidos. Atualmente não faz sentido, em qualquer SGBD, ignorar capacidades *multicore* dos processadores uma vez que todos os fabricados atualmente a possuem. Portanto o objetivo primário é adaptar/-modificar a AVP criando novas variações deste algoritmo que possuem capacidades *multicore*. Os algoritmos criados no presente trabalho foram implementados no ParGRES, que é a implementação de referência da AVP e o objetivo prático é obter o máximo de aceleração possível em relação à AVP original.

No decorrer deste trabalho, observou-se que usando o ParGRES como *middleware* em um SGBD qualquer caixa preta pode-se adicionar a capacidade de processamento *multicore* independente do SGBD subjacente possuir tal funcionalidade ou não, desde que seja para consultas OLAP. O objetivo secundário é então usar o ParGRES com a AVP *multicore* implementada, como *middleware* em um nó único, e obter alguma aceleração decorrente do processamento *multicore*, aproveitando a capacidade do processador que de outra forma seria desperdiçada. Para esse objetivo, qualquer ganho de desempenho do algoritmo *multicore* em relação ao SGBD sendo usado com núcleo único (*singlecore*) já justifica seu uso, apesar de um bom desempenho fortalecer a sua recomendação.

1.3 Metodologia

A metodologia utilizada no trabalho foi:

1. Estudo da AVP e ParGRES - Leitura de trabalhos e estudo dos princípos da SVP, FGVP e AVP.

2. Prática de execução do ParGRES (*Amazon Web Services (AWS)* e Lobo Carneiro) - Execução do ParGRES com diferentes configurações e aprendizado prático de uso do supercomputador Lobo Carneiro. Também foram feitos testes preliminares em um cluster de máquinas virtuais na *Amazon Web Service* (*AWS*).

3. Levantamento Bibliográfico sobre Processmento de Consultas *Multicore* - Decisão da *string* de busca, busca de trabalhos e leitura dos resumos para seleção.

4. Revisão - Leitura dos trabalhos selecionados, e busca e leitura de trabalhos encontrados nas referências destes.

5. Implementação de Algoritmos *Multicore* na AVP - Implementação e testes de diversas propostas de algoritmos *multicore* na AVP.

6. Validação Experimental - Testes finais, em base de dados com o tamanho final, compilação de resultados e produção de gráficos e demais artefatos.

1.4 Organização

No Capítulo 1 o assunto é introduzido, os objetivos e a motivação são declarados e a motivação é apresentada. No Capítulo 2 a AVP é apresentada e explicada de forma cronológica, iniciando pela SVP e passando pela FGVP. No Capítulo 3 são apresentados trabalhos correlatos, que implementam de alguma forma processamento paralelo em SGBDs. No Capítulo 4 são propostos dois algorismo que modificam a AVP para se tornar um algoritmo multicore. O Capítulo 5 apresenta a avaliação experimental feita para testar os algoritmos multicore e avaliar os objetivos do trabalho, e são apresentados e discutidos os resultados. No Capítulo 6 é apresentada a conclusão final do trabalho e os trabalhos futuros.

Capítulo 2

A AVP (Adaptive Virtual Partitioning)

O algoritmo de processamento paralelo de consultas OLAP, a AVP, foi desenvolvido e aperfeiçoado em uma cronologia que se inicia com a formalização de partições virtuais (VP) e o algoritmo de processamento SVP, passando pelo algoritmo mais sofisticado FGVP e finalmente culminando com o estado da arte atual, a AVP, que resolve problemas que podem ocorrer na SVP.

A AVP em particular possui como implementação padrão o ParGRES [5] que é um *middleware* utilizado em *cluster shared nothing*, utilizando um SGBD caixa preta que deve possuir como requisitos aceitar consultas SQL e possuir um conector JDBC. O ParGRES é um sistema projetado para processar de forma eficiente consultas OLAP[7].

2.1 SVP

O algoritmo *Simple Virtual Partitioning* (SVP) foi apresentado em [4] para oferecer uma estrutura de execução de consultas OLAP que se beneficiava de *clusters* de computadores pessoais (PC) comuns, abundantes e relativamente baratos no mercado. São utilizados SGBDs relacionais comerciais, e são executados individualmente e de forma independente em cada PC do *cluster*. A SVP é uma forma eficiente de se executar consultas com paralelismo intra-consulta.

A arquitetura usada como ambiente de execução da SVP foi a *shared nothing* [1], que é natural e intuitiva se pensarmos em agrupamentos de PCs, pois tais máquinas não possuem qualquer forma de compartilhamento de memória, a conexão deve ser feita através de rede local podendo utilizar padrões mais rápidos de transferência para clusters (*e.g. Myrinet* e *InfiniBand*).

A proposta de uso de *clusters shared nothing* na SVP também permite, ide-

almente, um sistema com capacidade de processamento paralelo escalável com o acréscimo de novos PCs no *cluster*, sem modificações significativas nos já existentes ou refatoração de toda a rede de PCs.

O desafio resolvido pela SVP foi como distribuir e coordenar a execução de consultas em um ambiente distribuído com a respectiva arquitetura *shared nothing*, e para tal um dos nós da rede foi escolhido como coordenador, que possui um conhecimento global sobre o estado da consulta individual em cada nó. Os clientes não se comunicam entre si e não possuem qualquer informação sobre o estado de processamento distribuído da consulta, apenas submetem os resultados para o coordenador.

Para poder cumprir seu objetivo de executar consultas em um ambiente distribuído, a SVP introduziu o conceito de partição virtual. Todos os nós do *cluster* possuem uma réplica completa da base de dados, porém cada nó apenas processa uma partição virtual que é delimitada por um intervalo. É utilizada uma chave primária indexada sobre a tabela fato para a delimitação do intervalo. Cada subconsulta inclui no predicado esses limites da partição virtual na chave primária. Dessa forma cada nó processa uma partição virtual da base.

Seja por exemplo a consulta Q_x:

```
SELECT
    o_totalprice as price
FROM
    orders
WHERE
    o_orderdate > date '1994-01-01';
```

As subconsultas Q_{x1} e Q_{x2} são feitas pela manipulação do predicado de consulta, gerando:

```
SELECT
    o_totalprice as price
FROM
    orders
WHERE
    o_orderdate > date '1994-01-01'
    AND o_orderkey >= 0 AND o_orderkey <= 1000;
```

e

```
SELECT
    o_totalprice as price
FROM
    orders
WHERE
```

```
o_orderdate > date '1994-01-01'
AND o_orderkey > 1000;
```

Após a execução das subconsultas, os resultados parciais sofrem um *merge* no coordenador para gerarem o resultado final.

Se a tabela fato não tiver uma chave primária, pode-se sempre criar uma, o que é especialmente recomendável caso a chave existente não seja contínua. Essa chave primária precisa ser indexada com índices *clusterizados*, e o motivo é que pode acontecer, a depender do SGBD e do algoritmo usado [8], de uma grande porção da base ser processada ao invés de apenas a partição virtual, degradando o desempenho de cada nó. Isso anula o benefício do processamento distribuído.

De acordo com [3] devem ser respeitadas as seguintes limitações para evitar uma varredura completa da tabela fato:

1. A fragmentação virtual da tabela fato deve ser completa e disjunta, o que significa que toda tupla está presente em um fragmento, e cada tupla pertence a somente um fragmento, respectivamente.

2. As tuplas da partição virtual devem estar fisicamente ordenadas de acordo com o atributo de fragmentação.

3. A relação fragmentada deve ser indexada no atributo de fragmentação.

4. O SGBD deve obrigatoriamente utilizar o índice para a obtenção das tuplas.

5. Os valores do atributo de fragmentação devem ser uniformemente distribuídos entre as tuplas da relação virtual.

Ainda que respeitadas as condições acima não existem garantias de que o SGBD não fará uma varredura completa da tabela fato, e isso pode ocorrer se o tamanho da partição virtual for grande o suficiente para a induzir o otimizador do SGBD a fazer a varredura.

Para o a otimização de consultas da SVP, como primeiro passo as consultas são dividas nas seguintes categorias:

1. Consultas sem subconsultas que processam alguma tabela fato:

   (a) Consultas que processam apenas uma tabela fato e nenhuma outra.

   (b) Consultas que processam apenas uma tabela fato e um número arbitrário de dimensões.

   (c) Consultas que processam mais de uma tabela fato, todos as junções são *equi-join*, e o grafo da consulta é livre de ciclos.

2. Consultas com subconsultas que se encaixam em alguma categoria de 1.

3. Consultas que não se encaixam nas categorias citadas anteriormente.

Baseando-se nessa categorização o otimizador escolhe quantos nós do *cluster* serão usados para processar cada consulta. Espera-se uma aceleração linear nas consultas 1a e aproximadamente linear nas consultas 1b e 1c. Consultas da classe 3 não são paralelizadas.

De acordo com [8], a vantagem da SVP é a sua simplicidade, porém apresenta como desvantagem a dependência de recursos do SGBD usado. Pode ocorrer a leitura completa de tabelas em SGBDs que não possuem como funcionalidade índices *clusterizados*. A SVP é uma técnica de decomposição estática, em bases com tabelas cujos dados possuem algum viés pode haver perda de desempenho ao se processar partições virtuais, e não é possível fazer qualquer balanço de carga para contornar esse problema.

2.2 FGVP

Foi então proposta a *Fine-Grained Virtual Partitioning* (FGVP) em [9] para contornar o problema apresentado na SVP do otimizador do SGBD fazer a varredura completa das bases ao invés de processar apenas uma partição virtual, problema esse que pode ocorrer dependendo da configuração do sistema, do SGBD usado e da base de dados.

A estratégia utilizada pela FGVP é dividir a base em um número de partições virtuais muito maior que o número de nós do *cluster*, e dessa forma evitar os problemas anteriormente mencionados. Por exemplo, se a execução de uma consulta for feita em um *cluster* com 4 nós, pode-se produzir 64 partições virtuais e enviar 16 partições virtuais para ser processada por cada nó. Acredita-se que isso seja benéfico não só por que evita a varredura da tabela completa pelo otimizador, mas também estruturas temporárias usadas para processar a consulta ficam reduzidas e possivelmente podem ser armazenadas em memória.

A escolha da quantidade de partições é importante, pois um número pequeno pode não evitar a varredura completa de tabelas, e portanto não apresentar melhoras em relação à SVP. Um número grande pode criar uma perda de desempenho devido à comunicação exagerada entre processos. Para a FGVP foi escolhido o número de partições virtuais baseando-se em características do SGBD, o tamanho escolhido foi o maior possível tal que o tamanho de uma partição não provoque uma varredura completa pelo otimizador de consultas.

Não se pode deixar de notar a possibilidade aberta pela FGVP de fazer um balanço de cargas modificando o tamanho das partições virtuais ainda não processadas.

Tal possibilidade foi citada no trabalho que apresentou a SVP ([4]) mas a forma de fazê-la não foi especificada. Já que não é possível interromper uma consulta para alterar o tamanho da partição virtual, o uso de muitas pequenas partições permite, baseando-se em resultados recentes do processamento das subconsultas, alterar o tamanho de partições a serem processadas no futuro.

2.3 AVP

A *Adaptive Virtual Partitioning* (AVP), ou em português Fragmentação Virtual Adaptativa foi proposta em [8] com o objetivo de oferecer uma técnica de processamento com paralelismo intra-consulta que evita os problemas e desvantagens da SVP, e acrescenta à FGVP a capacidade de balanceamento de carga, alterando dinamicamente os tamanhos das partições virtuais a serem processadas para melhorar o desempenho.

A AVP possui níveis de implementação, pois é executada em cada nó e depois exige uma coordenação paralela entre os nós do *cluster*. O algoritmo a ser executado em cada nó se inicia ao receber uma partição virtual para processar, que inicialmente é determinada da mesma forma que na SVP, e em seguida ele quebra a partição em partições menores seguindo o algoritmo adaptativo:

1. Inicie processando um tamanho de partição muito pequena, começando pelo primeiro valor escolhido para ser processado pelo nó.
2. Execute a consulta utilizando esse tamanho de partição e meça o tempo de execução.
3. Aumente o tamanho da partição.
4. Execute a consulta utilizando esse novo tamanho de partição e meça o tempo de execução. Ela deve começar ao final da partição anterior.
5. Se o tempo de execução, dividido pelo tamanho da partição, for menor que o tempo de execução da partição anterior dividido pelo seu respectivo tamanho, retorne ao passo 3.
6. Pare de aumentar o tamanho das partições, um estado estável foi encontrado.
7. Continue executando a consulta até que haja uma deterioração do tempo de execução ou que termine a partição escolhida para o nó, e neste último caso interrompa o processo.

8. Caso haja deterioração do tempo de execução da partição no estado estável, diminua o tamanho da partição e retorne ao passo 2. Ser não houve deterioração retorne ao passo 7.

Quando um nó termina de processar o seu fragmento, ele oferece ajuda para processar fragmentos de outros nós que não tenham acabado ainda, esse passo é fundamental pois é nesse ponto onde ocorre o balanceamento de cargas, retirando partições de nós que estão ocupados e direcionando-as para serem processadas por nós ociosos. As fases do processamento da consulta pelo *cluster* são:

1. **Fragmentação virtual inicial:** De forma idêntica à SVP, a tabela fato é dividida em partições virtuais de igual tamanho e cada partição é enviada para cada nó.

2. **Ajuste dos tamanhos dos fragmentos:** Nessa fase a AVP quebra as partições em partições menores e processa a consulta de acordo com o algoritmo da seção 2.3. Isso evita a possível varredura da tabela completa que pode ocorrer na SVP.

3. **Redistribuição de carga:** Utiliza uma técnica dinâmica para redistribuir a carga de nós ocupados para nós ociosos, resolvendo o problema ocasionado por dados com viés.

4. **Encerramento:** Finaliza a execução paralela da consulta e compõe o resultado a partir dos resultados das consultas de cada partição.

2.3.1 Redistribuição de Carga

A redistribuição de cargas resolve o problema de desbalanceamento que pode ocorrer na SVP se as partições possuírem distribuições de dados irregulares e com diferentes vieses. A implementação do balanço de cargas foi descrita em [3] e utiliza uma organização lógica dos nós em uma rede de difusão de mensagem, de forma a evitar sobrecarga e centralização de trabalho em um nó.

2.3.2 Composição dos resultados

Os três algoritmos citados anteriormente, SVP, FGVP e AVP dividem uma tabela fato em partições virtuais a serem processadas por diferentes nós. Seja a consulta Q aplicada na relação R, cujo resultado é $Res(R)$ e sejam as subconsultas $Q_1, Q_2, ..., Q_M$ sobre as partições virtuais $P_1, P_2, ..., P_M$ cujos resultados individuais são $Res(P_1), Res(P_2), ..., Res(P_M)$, o resultado da consulta Q é definido pelo operador ∇ sobre os resultados das subconsultas sobre as partições: $Res(R) = \nabla Res(P_i)$.

Para cada tipo de consulta um operador ∇ diferente é utilizado. Em consultas de varredura simples, sem qualquer função de agregação, o operador ∇ é substituído pelo operador de união generalizada: $Res(R) = \bigcup_i Res(P_i)$. Já para consultas com agregação, a composição dos resultados das subconsultas para se obter o resultado da consulta não é trivial. As respectivas operações de composição de consultas foram definidas em [10] e podem ser vistas na tabela abaixo:

Função	Composiçao resultante
MAX(P)	$MAX([MAX(P_1), MAX(P_2), ..., MAX(P_M)])$
MIN(P)	$MIN([MIN(P_1), MIN(P_2), ..., MIN(P_M)])$
SUM(P)	$SUM([SUM(P_1), SUM(P_2), ..., SUM(P_M)])$
COUNT(P)	$SUM([COUNT(P_1), COUNT(P_2), ..., COUNT(P_M)])$
AVG(P)	$\frac{SUM([SUM(P_1),SUM(P_2),...,SUM(P_M)])}{SUM([COUNT(P_1),COUNT(P_2),...,COUNT(P_M)])}$

Tabela 2.1: Composição de resultado sobre a partição P para formar uma função de agregação.

2.4 Evolução do ParGRES/AVP

O trabalho [10] aborda o aspecto do ParGRES de exigir replicação total da base em todos os nós do cluster. É proposta uma distribuição eficiente de dados que combina particionamento físico e virtual. A técnica denominada *Chained Declustering* é adotada como técnica de particionamento. A validação experimental foi feita em um protótipo de sistema denominado SmaQSS DBC. O trabalho também implementa uma nova técnica de distribuição de cargas denominada QueCh, que foi testada no ParGRES.

O trabalho [11] testou o ParGRES em um conjunto de dados reais obtido do censo doméstico brasileiro feito pelo Instituto Brasileiro de Geografia e Estatística (IBGE) no ano 2000. As consultas foram extraídas da *Statistical Multidimensional Database* (BME), que é uma ferramenta fornecida pelo IBGE com a base, já organizada em fatos e dimensões, e consultas *ad-hoc* OLAP sobre a mesma. Como resultado o ParGRES produziu aceleração linear e em alguns casos super-linear, se mostrando uma alternativa viável para consultas analíticas em grandes volumes de dados.

O trabalho [12] apresenta o GParGRES, uma derivação do ParGRES para ser executado em *grids*, que implementa *split* no nível do *grid*. O GParGERS encapsula o ParGRES e acrescenta outros componentes arquiteturais para controlar a execução como o *Distribuited Query Service (DQS)* e o *Factory Service (FS)*. A validação experimental foi feita em dois *clusters* compondo um *grid*, os resultados mostram aceleração quase linear sem qualquer otimização no SGBD ou da rede.

O trabalho [13] desenvolveu o C-ParGRES, uma extensão do ParGRES projetada para ser executada como um serviço de nuvem e que explora características como provisionamento de recursos sob demanda e elasticidade. Além disso o C-ParGRES permite criar multiplos *clusters* independentes com diferentes bases de dados para serem acessados por diferentes usuários. Essas capacidades específicas para a nuvem foram obtidas com a implementação dos módulos "*dimensioner*" e "*deployment*". A validação experimental foi feita com uma base de dados do IBGE, os resultados mostram aceleração sublinear na maioria das consultas e a previsão de custos para 1 ano de uso foi abaixo da licença de outros SGBDs.

Capítulo 3

Trabalhos Correlatos

Aparentemente processamento distribuído e paralelo (no sentido de ser executado por múltiplos *cores*) são duas coisas muito correlacionadas, pois ambas se tratam de usar entidades diferentes de processamento para processar uma consulta. Tal afirmação é muito enganosa, de acordo com [1] as implementações paralelas possuem como fatores críticos o algoritmo de processamento paralelo de consulta, a disposição de dados e o balanço de carga. Essas diferenças ocorrem por que a comunicação e a quantidade de entidades de processamento pode variar bastante entre ambientes de processamento multicore e distribuído.

Uma das classificações de processamento paralelo distingue duas categorias: paralelismo *intra-operador* ocorre quando agentes diferentes de processamento processam partes diferentes de um mesmo operador. Já o paralelismo *inter-operador* ocorre quado dois ou mais operadores são processados em paralelo, possivelmente com a aplicação de técnicas de pipeline.

Apesar de existirem abundantes trabalhos e propostas de SGBDs, para diferentes propósitos e arquiteturas, que implementam execução de consultas concorrente, os trabalhos apresentados aqui foram selecionados por possuírem de forma explícita execução *multicore*, onde é levado em consideração especificidades desse tipo específico de paralelismo.

Os trabalhos selecionados para esta revisão foram obtidos usando as *strings* de busca:

```
"parallel query" OR "parallel join" OR "parallel aggregate" OR
"parallel sql" OR "Parallelizing SQL" OR "Parallelizing query"
OR ""(intra + query + multicore)
```

```
((parallel OR parallelizing) AND (query OR join OR aggregate OR
sql OR scan)) OR ((intraquery OR intra-query or "intra query")
AND multicore)
```

A primeira foi utilizada no agregador *Google Scholar*[1], e a segunda no *Scopus*[2].

A escolha inicial foi feita lendo-se o título e eliminando-se os trabalhos que não cobriam o tema de processamento paralelo em ambientes *multicore*, em seguida foram lidos os resumos e eliminados os trabalhos seguindo o mesmo critério descrito para a eliminação de trabalhos pelo título.

O último passo foi ler os artigos selecionados e buscar nas referências, de forma *ad hoc* outros artigos que possuem o mesmo tema e apresentam propostas que despertaram interesse. Portanto o critério de seleção foi uma mistura de uma técnica tradicional de revisão usando *strings* de busca com a técnica Bola de Neve, usando decisões *ad hoc*.

3.1 Trabalhos

3.1.1 *Encapsulation of Parallelism in the Volcano Query Processing System*

Em oposições a outros trabalhos que utilizam o modelo *bracket*, este trabalho [14] optou por um modelo de operadores pois o modelo *bracket* possui como desvantagens necessitar de um agendador externo, o que demanda desenvolvimento além dos operadores. Além disso, no modelo *brackets* cada operador controla seu próprio fluxo de entrada/saída o que demanda muitas comunicações entre processos (IPC) ,podendo deteriorar o desempenho.

Em relação ao design geral, o Volcano possui como essência operadores que implementam a interface *open*, *next* e *close*. Cada operador executa uma tarefa em uma árvore de execução de consultas e adota o conceito de *anonymous inputs or streams*, que significa que para o operador é indiferente se o fluxo de dados é proveniente de outro operador ou de uma entrada de dados. Para gerenciar um mesmo operador que pode ser instanciado mais de uma vez em uma única consulta, é implementado um "*state record*" através de um *hash map*.

O responsável pela implementação da paralelização no Volcano é o operador *exchange*, que possui a mesmas interface e princípios dos outros operadores. Este operador é inserido em pontos da árvore de execução e é responsável pelo paralelismo vertical e horizontal, que serão descritos em seguida.

O paralelismo vertical, ou *pipeline*, é implementado pelo operador *exchange* através da criação de um novo processo. Ocorre um *fork* do processo que estava processando a consulta e o processo pai passa a ser o consumidor, já o processo filho passa a ser o produtor com o auxílio de uma estrutura denominada *port* que é mantida em

[1]https://scholar.google.com/
[2]https://www.scopus.com/

memória compartilhada para a sincronização. Ambos os processos se comunicam por uma interface de *iterator* (*open*, *next*, *close*) porém com uma especificidade, a comunicação utiliza IPC ao invés de chamadas de funções.

O autor distingue duas formas de paralelismo horizontal: o paralelismo *bushy* onde diferentes CPUs executam diferentes sub-árvores da árvore de execução e o paralelismo *intra-operador*, onde diferentes CPUs processam o mesmo operador porém em conjuntos diferentes de dados.

O paralelismo do tipo *bushy* já é naturalmente alcançado pela colocação do operador *exchange* na árvore de consulta, já o paralelismo *intra-operador* é implementado com o auxílio de múltiplas filas na estrutura *port*. Dessa forma diversos pares de consumidores e produtores executam em porções diferentes de dados a consulta em paralelo. O mesmo procedimento de *fork* utilizado no paralelismo vertical se aplica aqui também.

Testes foram feitos especialmente com objetivo de avaliar o desempenho ao se variar o tamanho dos *buffers* e pacotes usados na comunicação entre operadores e processos. A conclusão é que o paralelismo vertical compensa até para planos de execução com árvore curta, e o tamanho do pacote de comunicação deve ser de 250 ou mais registros para otimizar o desempenho do operador *exchange*.

3.1.2 *Parallel Query Processing in Databases on Multicore Architectures*

A proposta do trabalho [15] reside na criação de operadores relacionais opacos que encapsulam a execução relacional assíncrona e podem ser acoplados a outros operadores, e são utilizados sem que se saiba detalhes da sua implementação. A ideia é semelhante a utilizada em [14], porém a abordagem desse trabalho é mais madura, permite paralelismo *intra-operador* e resolve o problema da preservação de ordem.

O encapsulamento do paralelismo é combinado com um plano de otimização de consulta em duas fases. Na primeira fase a otimização estática comum ocorre, baseando-se em um conjunto de condições de contorno para evitar o problema NP-difícil. Na segunda fase as *threads* de execução são rebalanceadas, otimizando a execução da consulta.

Assume-se que os operadores são baseados no modelo de iterador, que possui como interface os métodos *open()*, *next()* e *close()*. É proposto o operador $ASYNC$, que pode ser colocado entre quaisquer operadores em um plano de consulta e cria *threads* que paralelizam a execução.

São criados três tipos de operadores $ASYNC$, o $ASYNC_{pass}$ é utilizado para criar um pipeline onde os dados fluem para serem processados, em paralelo, por um trecho do plano de execução. O $ASYNC_{fifo}$ ordena o resultado obtido de um

trecho do plano de consulta que foi executado em paralelo. O $ASYNC_{sort}$ é utilizado para a execução de paralelismo intra-operador, e também preserva a ordem de uma execução em paralelo.

A primeira fase de otimização é estática e considera o "*left-deep operator trees*", assumindo o resultado como próximo do ótimo para evitar a solução do problema NP-difícil. Essa fase considera como parâmetros a memória disponível para cada operador $ASYNC$ ($async\_max\_buffer$) e a quantidade máxima de fragmentos de pipeline ativos em dado momento ($async\_max\_threads$).

A segunda fase de otimização é dinâmica e ocorre em tempo de execução do *pipeline*. Considerando uma relação de produtor/cosumidor entre os estágios do pipeline, quando o produtor precisa esperar por um *buffer* de entrada completo ele aumenta o tamanho do seu próprio *buffer* para ser usado no próximo ciclo de processamento, isso é repetido até que um balanço entre produtor e consumidor seja alcançado. Se o *buffer* em questão estiver no início do *pipeline*, um novo *pipeline* é adicionado.

Foram feitos testes experimentais da proposta, conclui-se que para consultas adequadas ao paralelismo houve aceleração aproximadamente linear, já consultas *ad-hoc* apresentaram uma redução do tempo de execução em mais de 30%. Porém, existem consultas não adequadas para a paralelização proposta, assim como consultas que são executadas de forma muito rápida, nestes casos houve perda de desempenho.

3.1.3 *Query Processing on Multi-Core Architectures*

O trabalho [6] declara como o novo desafio dos engenheiros e desenvolvedores de bases de dados implementar o paralelismo nas consultas, uma vez que não se espera mais que os processadores acelerem sua velocidade de processamento como acontecia nos últimos 30 anos. A tendência futura da indústria é produzir processadores com vários núcleos.

Define-se como categorias de processamento paralelo o paralelismo *inter-operador*, onde vários operadores são executados em paralelo, e o processamento *intra-operador*, onde um único operador é executado em paralelo.

Para explorar o potencial do paralelismo, é necessário estar atento à implementação física do armazenamento no SGBD. O trabalho usa fragmentação vertical pois esta abordagem se mostrou como uma forma eficiente de gerenciar *cache* em outros trabalhos.

Uma visão abstrata de *multicore* é criada para proporcionar ao desenvolvedor de operadores uma arquitetura *multicore* padrão focando em *caches* e *cores*, e abstraindo detalhes de diferentes CPUs com diferentes arquiteturas internas. Também são adicionados operações lógicas não existentes para agrupar *cores* e *caches* em cha-

mados "*workgroup*", que podem tanto processar a mesma operação em diferentes partes dos dados quanto decompor uma operação em operações menores. Também é possível uma combinação híbrida, que incorpora as duas alternativas descritas.

O trabalho introduz o conceito de meta operadores, que são operadores usados no processo de paralelização de consultas, e que possuem implementação específica de cada arquitetura com suas particularidades, tendo como responsabilidades gerência de memória, *cache*, alocação e desalocação de estruturas. Os meta operadores encapsulam especificidades da arquitetura e dão liberdade aos desenvolvedores para criar operadores, como uma junção por exemplo, sem se preocupar com detalhes da arquitetura. Exemplos de meta operadores são o *TransferData*, o *AllocWorkGroup(C, S)* e o *AssignTaskToWorkGroup(W, G, T)*.

Juntando todos os componentes acima: implementação física (fragmentação vertical), visão abstrata de multicore e meta operadores cria-se um *framework* para processamento paralelo de consultas, onde meta operadores abstraem especificidades da arquitetura e criam operações pertencentes à visão. O programadores devem fazer operadores que operem usando os meta-operadores com o objetivo de a incorporar o paralelismo.

3.1.4 *Query Processing and Optimization of Parallel Database System in Multi Processor Environments*

Este trabalho [16] dá semelhante importância a dois fatores do processamento paralelo de consultas, a geração do plano de consulta e a execução propriamente, e busca otimizar o processo como um todo. É dado destaque às duas arquiteturas principais para processamento paralelo: *shared memory (SM)* e *shared nothing* (SN).

É de responsabilidade do SGBD gerenciar a sincronização das tarefas, a concorrência, compartilhar recursos, posicionar os dados adequadamente e o escalonamento da rede. A organização dos dados é essencial para o balanço de carga e a sincronização exige travas em arquiteturas *shared nothing* ou com ligações fracas entre os componentes. O ideal é dividir os dados em diferentes conjuntos, de forma que o processamento de uma tarefa não interfira em outra, esses conjuntos podem ser obtidos por particionamento vertical ou horizontal.

Após observações e reflexões sobre as condições presentes nas diferentes arquiteturas são propostas duas estratégias, a primeira é indicada para arquiteturas em que a memória não é compartilhada:

1. Separar as relações em relação aos predicados que são usados nas consultas.

2. Atribuir cada trecho da relação para um processador individual com sua memória.

3. Transferir os resultados intermediários com os predicados para um processador único obter o resultado final da consulta.

4. Repetir o passo 3 até que todos os predicados estejam processados.

A segunda abordagem é recomendada para arquiteturas que possuem memória compartilhada:

1. Distribuir as tuplas da relação entre vários discos/unidades de memória, criando réplicas.

2. Em caso de seleção/retenção de elementos, aplicar o mecanismo de trava para blocos, para apenas uma CPU acessar uma porção de dados.

3. Em caso de consultas de atualização, aplicar o mecanismo de trava para registros.

3.1.5 *Scheduling threads for intra-query parallelism on multicore processors*

O trabalho [17] aborda o seguinte problema: suponha um ambiente *multicore* com memória compartilhada e possivelmente *caches* compartilhados, cada core pode possuir mais de uma *thread*, e várias *threads* são usadas para processar uma consulta usando o modelo *intra-consulta*, qual a melhor alocação de cada *thread* para operadores em um dado plano de consulta?

Existem duas alternativas ingênuas sugeridas em trabalhos sobre sistemas paralelos, a primeira é atribuir para cada operador uma *thread*, e a segunda é usar todas as *threads* disponíveis distribuindo-as entre os operadores. Devido às características únicas dos sistemas multicore essas abordagens falham em extrair o poder máximo de processamento de consultas de sistemas *multicore*.

O problema completo cresce exponencialmente com o número de *threads* disponíveis, e também é agravado pelo número de operadores no plano de consulta rapidamente tornando-se intratável. As soluções propostas no trabalho fazem uso de heurísticas, e utilizando um mínimo de processamento extra encontram uma boa solução para a alocação de *threads*. Uma análise formal sobre o problema indica que o algoritmo de agendamento pode ser reduzido ao algoritmo de agendamento de tarefas paralelas, que é comprovadamente NP-Completo.

Foram propostos 5 algoritmos de busca:

1. *GreedyLevel*

2. *ExpandChildren*

3. *ExpandDescendants*

4. *ExpandSibling*

5. *ExpandHybrid*

Para testar o desempenho dos algoritmos propostos e comparar com outras soluções, foram geradas 8 árvores *bushy* de tarefas binárias usando os parâmetros: *threads* [4,32], *tasks* [3,63], custo de execução de operador [100, 100000]milissegundos e curvas de escalabilidade linear, polinomial#1, polinomial#2 e logarítmica.

Os testes de quão ótimo os algoritmos são foram feitos obtendo-se a solução ótima através de enumeração exaustiva. No entanto, para tais testes o número de *threads* na geração das árvores foi limitado a 9, além desse valor não é mais praticável obter a solução ótima para fazer as comparações. Como resultado todos os algoritmos, nas 8 consultas/árvores, obtiveram desempenho superior ao algoritmo ingênuo, alcançando a solução ótima em acima de 80% dos casos.

Os próximos testes são de melhoramento, e medem o quão superior é o desempenho em relação ao algoritmo ingênuo em diferentes variações de parâmetros.

Em particular a curva de escalabilidade foi um parâmetro inserido no modelo que determina um comportamento concorrente que visa simular características da tarefa que pode exigir sincronia ou não, e também características do *hardware* que pode apresentar gargalos. Foram feitos testes para avaliar como diferentes curvas de escalabilidade possibilitavam maior ou menor melhoramento dos algoritmos apresentados e os resultados mostram que, para diferentes números de *threads* e diferentes curvas de escalabilidade diferente algoritmos se destacam.

A análise do melhoramento de desempenho ao se variar o número de tarefas da árvore também mostra resultados heterogêneos, com diferentes algoritmos obtendo o melhor desempenho em diferentes números de tarefas, mas destaca-se o *ExpandHybrid* que obteve o maior melhoramento para um número de *threads* $N = 16$ e o algoritmo *GreedyLevel* que se destacou com o maior melhoramento para um número de *threads* $N = 4$.

A última análise feita foi variar o parâmetro *threads* e medir o melhoramento. Novamente o resultado é heterogêneo, com destaque para o algoritmo *ExpandHybrid* que para um número de tarefas $K = 7$ obteve o maior melhoramento para valores acima de 4 *threads*, e nos testes com um número de tarefas $K = 31$ obteve o maior melhoramento somente acima de 8 *threads*.

3.1.6 *A Comparative Study of Implementation Techniques for Query Processing in Multicore Systems*

No trabalho[18] foram introduzidos os tipos de paralelismo implementados pelos fabricantes de chip, que são o *pipeline* e o *simultaneous multithreading (SMT)*. Argumenta-se que o desempenho do processamento multicore está relacionado à falhas de *cache* (*cache misse*), uma vez que a operação de ir buscar dados em memória, ou até em níveis mais baixos do *cache*, é muito custosa.

Os gargalos provocados pelo acesso à memória são de três tipos: acessos sobrepostos provocam falhas de *cache* pois a cada acesso um possível dado diferente e sem localidade é inserido no *cache*, áreas comuns de escritas criam travas uma vez que a escrita precisa ser sincronizada entre diferentes *threads* e gerenciamento de memória pode provocar gargalos pois o sistema operacional intervém para alocar o melhor tamanho da porção de memória, já alocações no *bufferpool* sofrem com paginação.

Para contornar os gargalos citados acima, a implementação dos algoritmos foi feita com estruturas de dados específicas. A biblioteca utilizada foi a *Intel's Threading Building Blocks (TBB)* e foram aproveitados os padrões de paralelismo oferecidos como iteração paralela, redução funcional, varredura de prefixo e *pipeline*. As estruturas criadas foram filas *lock-free* de memória com porções em tamanhos que crescem de forma exponencial, de modo a evitar o problema de gerenciamento de memória. Já para evitar leituras e escritas sobrepostas foram criadas duas estruturas, o vetor expansivo e a matriz de fragmentação.

Com o auxílio das estruturas de dados desenvolvidas, foram implementados as operações padrão de consultas: seleção, fragmentação hash, ordenação, *equi-join* e agregação (*min*, *max*, *sum* e *count*). Para cada uma dessas operações mais de um algoritmo foi criado ou escolhido a partir de outros trabalhos, porém o algoritmo que usa as estruturas foi escolhido por apresentar vantagens em relação aos demais.

Para avaliar experimentalmente os algoritmos propostos foram feitos testes em três processadores diferentes, um Intel Xeon E5420, um AMD Opteron 1385 e um Intel Xeon X5550, todos compatíveis com a biblioteca *TBB*. As consultas são de *microbenchmarks* em oposição a consultas complexas, para medir com mais precisão o comportamento dos algoritmos, e foram usados os primeiros 10 atributos do *benchmark* Wisconsin.

Foram feitos testes de variações de parâmetros dos algoritmos (granulação e tamanho do fragmento) com os algoritmos *hash*, *count* e *split*, testes variando-se as *threads* considerando esses parâmetros, testes com diferentes fatores de seletividade para o algoritmo de varredura e diferentes cardinalidades para os os algoritmos *split*, *count* e *hash*.

De um modo geral, todos os algoritmos obtiveram acelerações, exceto o algo-

ritmo ingênuo para junção que não acelerou. Conclui-se que os fatores citados como gargalos de fato causam perda de desempenho, e que quanto menos uniforme são as distribuições dos dados sobre quais a consulta é feita, mais específica a técnica precisa ser para obter bom desempenho. Os algoritmos possuem dependência do *hadware* em relação ao desempenho, podendo ser mais adequados para um perfil ou outro de processador/hierarquia de memória. Por fim recomenda-se o o uso de modelo de custos para extrair o melhor desemepnho em casos complexos e arbitrários.

3.1.7 *Database Hash-Join Algorithms on Multithreaded Computer Architectures*

O trabalho[19] introduz a necessidade de adaptar os SGBDs e estratégias de processamento de consultas para sistemas *multicore*, que é o novo paradigma de design que os processadores devem seguir. Nesse contexto, o principal gargalo é no acesso à memória principal e falhas de *cache*. O trabalho foca no desenvolvimento de um algoritmo específico crucial no processamento de consultas, o *Hash-Join*.

O estudo foi conduzido em um Intel Pentium 4 Prescot, baseado na microarquitetura *Intel NetBurst*. Este é o primeiro processador de propósito geral a ter SMT. Para realizar os testes diversos, um número arbitrário de tuplas foi criado em duas relações que possuem chaves de 10 bytes. Uma das relações possui 400MB e outra 200MB, sendo que cada tupla da relação maior corresponde a uma tupla da menor em uma junção.

O algoritmo proposto divide o algoritmo *Hash-Join* em três fases, a primeira fase é o algoritmo de fragmentação, a segunda fase é um algoritmo de construção que cria uma tabela *hash* e a terceira fase é a fase de junção, que executa propriamente a junção das tuplas e armazena o resultado em uma tabela.

As três fases citadas que constituem o algoritmo de junção *hash* são então submetidas a uma estratégia *multicore* que consiste em criar duas *threads*, cada ~~thread~~uma fica responsável por fragmentar uma tabela, após a menor das tabelas ser fragmentada o trabalho de construção da tabela *hash* pode começar a ser executado pela *thread*. A última fase, de junção, pode começar desde que a menor tabela já tenha passado pela fase de construção da tabela *hash* e a maior tabela já tenha passado pela fragmentação, neste caso as fases de construção e junção são feitas em conjunto.

Os resultados variam de acordo com o tamanho das tuplas e de forma inversamente proporcional a estas, mas a maior aceleração ocorre na fase de junção, com valores entre 36% e 53%, na fase de fragmentação os valores ficam entre 17% e 27%. Considerando o tempo total de todas as fases, o fator de aceleração fica entre os valores (aproximadamente) 1,3 e 1,45.

3.2 Propriedades dos Trabalhos

Em seguida são apresentadas duas tabelas com as propriedades de cada trabalho, de forma comparativa. Cada tabela possui um conjunto distinto de propriedades:

- Tipo de paralelismo - O tipo de paralelismo implementado ou abordado pelo trabalho (*e.g. intra-operator*).
- Arquitetura - A arquitetura na qual o trabalho foi implementado, pode ser um simples computador pessoal (*PC*) ou ambiente distribuído.
- Memória - Se refere à arquitetura de memória utilizada pelo procesamento *multicore* na qual o trabalho foi testado ou implementado.
- Paradigma - É o paradigma de programação do paralelismo *multicore*, pode ser memória compartilhada ou troca de mensagens.
- Adaptativo - Se a implementação é adaptativa ou não.
- Replicação da Base - Se o trabalho possui réplicas da base para o processamento paralelo, pode ser total, parcial ou não possui réplicas.
- Operadores - Lista os operadores implementados ou contemplados pelo trabalho, alguns implementam um operador específico (*e.g. Hash-Join*).

Ao final da tabela, as propriedades dos dois algoritmos propostos no presente trabalho (AVPMSimples e AVPMRandom) são colocadas para comparação e avaliação perante os trabalhos correlatos.

	Tipo de paralelismo	Arquitetura	Memória	Paradigma
Encapsulation of Parallelism in the Volcano Query Processing System	*bushy, intra-operador*	PC	compartilhada	memória compartilhada
Parallel Query Processing in Databases on Multicore Architectures	*bushy, intra-operador, Grafo Direcionado Acíclico*	PC	compartilhada	memória compartilhada
Query Processing on Multi-Core Architectures	*bushy, intra-operador*	PC	compartilhada	memória compartilhada
Query Processing and Optimization of Parallel Database System in Multi Processor Environments	*intra-operador, inter-operador*	*PC, NUMA, Shared Nothing*	compartilhada e não compartilhada	memória compartilhada, troca de mensagens
A Comparative Study of Implementation Techniques for Query Processing in Multicore System	*bushy, inter-operador*	PC	compartilhada	memória compartilhada
Database Hash-Join Algorithms on Multithreaded Computer Architectures	*intra-operador*	PC	compartilhada	memória compartilhada
Scheduling threads for intra-query parallelism on multicore processors	*intra-consulta*	PC	compartilhada	memória compartilhada
AVPMsimples	*intra-consulta*	*Shared Nothing*	compartilhada	troca de mensagens
AVPMandom	*intra-consulta*	*Shared Nothing*	compartilhada	troca de mensagens

Tabela 3.1: Tabela comparativa com propriedades dos trabalhos apresentados - tabela 1.

	Adaptativo	Replicação da Base	Operadores
Encapsulation of Parallelism in the Volcano Query Processing System	Não	Não	todos
Parallel Query Processing in Databases on Multicore Architectures	Sim	Não	todos
Query Processing on Multi-Core Architectures	Não	Não	todos
Query Processing and Optimization of Parallel Database System in Multi Processor Environments	Não	Total	todos
A Comparative Study of Implementation Techniques for Query Processing in Multicore System	Não	Não	*seleção, fragmentação hash, ordenação, equi-join, agregação (min, max, sum e count)*
Database Hash-Join Algorithms on Multithreaded Computer Architectures	Não	Não	*Hash-Join*
Scheduling threads for intra-query parallelism on multicore processors	Não	Não	todos
AVPMsimples	Sim	Total	todos
AVPMandom	Sim	Total	todos

Tabela 3.2: Tabela comparativa com propriedades dos trabalhos apresentados - tabela 2.

Capítulo 4

Proposta de AVP *Multicore*

Foram elaborados dois algoritmos que expandem o já existente AVP para um algoritmo *multicore* de processamento de consultas OLAP.

Todos os algoritmos apresentados consistem em dividir, de alguma forma, uma partição a ser processada em várias subpartições a serem processadas individualmente por cada *core*, e após o processamento paralelo os resultados são compostos. Como uma mesma consulta é executada por vários *cores*, em paralelo, pode-se classificar os algoritmos como algoritmos de paralelismo *intra-consulta*.

Levando em consideração as ideias e definições da AVP e SVP, estas subpartições são a rigor subpartições virtuais, uma vez que os nós possuem uma réplica completa da base e apenas definem a subpartição por predicados. No entanto, daqui em diante serão descrita como simplesmente subpartição.

O parâmetro comum a todos os algoritmos é a quantidade de *threads* usadas, cada *core* executa uma *thread* sendo possível um número maior de *threads* que de *cores*, porém algumas ficariam paradas durante parte do processamento, o que pode causar uma perda de desempenho. Esse parâmetro pode ser inserido em um arquivo de configuração pelo *Database Administrator (DBA)* permitindo maior flexibilidade na configuração do sistema. Pode não ser simples ajustar a quantidade de *threads* usadas de modo a otimizar os recursos do sistema, portanto existe a possibilidade de omitir essa configuração, nesse caso o sistema usa todos os *cores* disponíveis que foram detectados.

A primeira abordagem é a AVP *Multicore* Simples (AVPMSimples), que consiste em dividir toda e qualquer consulta da AVP em N subpartições e executá-las em paralelo, onde N é a quantidade de *cores*. Este é o algoritmo mais simples e ingênuo, pois ele não utiliza qualquer otimização para determinar o tamanho da subpartição a ser usado, tampouco leva em consideração o estágio e o tamanho da partição atual usada pela AVP.

A segunda abordagem é a AVP *Multicore* Randômica (AVPMRandom), ela considera que o tamanho da subpartição deve ser ajustado com o objetivo de otimizar a

execução e considera que esse tamanho pode variar de acordo com o estágio em que se encontra a consulta e tamanho de partição usada pela AVP em cada momento. É um algoritmo adaptativo, que busca de forma dinâmica e randômica por um novo tamanho de subpartição, começando por um tamanho pequeno, na tentativa de encontrar o tamanho ótimo.

4.1 Fragmentação

A fragmentação adotada é derivada da fragmentação original da SVP ([4]). Seja uma tabela fato que possui uma chave primária sintética. Existem vantagens em criar uma chave sintética, mas no caso da AVP isso é necessário pois essa chave pode ser controlada de forma a ser contínua, o que permite o algoritmo dinâmico da AVP encontrar o tamanho da partição sem ter que lidar com bases que possuem valores de chave faltantes. A relação R é então dividida pela AVP nas partições: $R = P_1, P_2, ..., P_M$.

Cada uma dessas partições P_i será processada por um nó do cluster, que dividirá em N subpartições, onde N pode ser dinâmico ou não, a ser determinado pelo algoritmo *multicore* de processamento de consulta. Então serão formadas as subpartições $P_{i,j}$ onde i se refere à partição obtida pela fragmentação da AVP e j é o índice que determina a subpartição dentro da respectiva partição, a ser processada pelo algoritmo *multicore*, ficando a relação dividida: $R = P_{1,1}, P_{1,2}, ..., P_{2,1}, P_{2,2}, ..., P_{M,1}, P_{M,2}, ..., P_{M,N}$.

Ao invés de considerar apenas uma subpartição isolada $P_{i,j}$, o algoritmo adaptativo AVPMRandom tenta ajustar o tamanho total das subpartições a serem processadas de uma só vez de forma paralela, ou seja, se estão sendo usados N *cores* define-se o **tamanho das subpartições** como $NP_{i,j}$.

A AVP possui replicação total da base em todos os nós do *cluster*, porém no contexto *multicore* todos os nós possuem acesso a uma partição que está em memória ou disco, e cada *core* processa uma subpartição usando a estratégia de fragmentação. Portanto o acréscimo de capacidades *multicore* à AVP não provoca a criação de novas réplicas da base.

4.2 Composição dos Resultados

O algoritmo AVP original produz como resultados das consultas sobre cada partição virtual $P_1, P_2, ..., P_M$ $Res(P_1), Res(P_2), ..., Res(P_M)$. Pode-se extrapolar essa afirmação para o caso *multicore* onde o resultado do processamento sobre cada subpartição $P_{1,1}, P_{1,2}, ..., P_{2,1}, P_{2,2}, ..., P_{M,1}, P_{M,2}, ..., P_{M,N}$ é

$Res(P_{1,1}), Res(P_{1,2}), ..., Res(P_{2,1}), Res(P_{2,2}), ..., Res(P_{M,1}), Res(P_{M,2}), ..., Res(P_{M,N})$ respectivamente.

Note que as subpartições, assim como as partições virtuais da AVP formam um conjunto disjunto e completo, logo as mesmas regras de composição de resultados usadas para a AVP descritas em Subseção 2.3.2 podem ser aplicadas à AVP *multicore*, bastando ao invés de compor todos os resultados das consultas sobre as partições virtuais, usar todos os resultados das consultas sobre as subpartições.

4.3 AVPMSimples

A AVPMSimples é o algoritmo mais simples e o primeiro que foi idealizado. Ainda que seja simples não pode-se fazer qualquer conclusão prévia a respeito do seu desempenho pois apesar de, intuitivamente ele não possuir uma configuração ótima para todos os tamanhos de partição que ocorem durante uma consulta, ele também evita algumas perdas de desempenho de outros algoritmos mais complexos.

A AVPMSimples consiste em, ao receber uma determinada consulta em uma partição da AVP, dividir esta partição em N subpartições, onde N é a quantidade de *cores*, e cada *core* executa a consulta em sua subpartição designada. Observe que a AVP pode usar diferentes tamanhos de partições em diferentes momentos do método adaptativo, e seria intuitivo que a quantidade de subpartições deveria variar de acordo com o tamanho da partição, porém a AVPMSimples ignora esse comportamento e sempre divide a partição recebida da AVP em um número fixo de N subpartições, por esse motivo ela é simples. O algoritmo está descrito abaixo de duas formas, em português e em pseudocódigo. A descrição em pseudocódigo possui algumas funções auxiliares usadas para expressar funções que não fazem parte do algoritmo e são importantes para o seu entendimento:

- $Fragmenta(P_i, incio, fim)$ - Produz a subpartição $P_{i,j}$ obtida à partir da partição P_i usando como limites da subpartição os valores de $incio$ e fim, o resultado é obtido pela aplicação de predicados na consulta.
- $Thread.processa(P_{i,j})$ - Processa a partição $P_{i,j}$ de forma paralela, ou seja, expressa a criação de *threads*, cada uma executando a consulta em uma subpartição diferente.
- $compoeResultados(Res(P_{i,j}))$ - Envia os resultados $Res(P_{i,j})$ para uma unidade responsável pela composição dos resultados.

AVPMSimples em português:

1. Seja N um parâmetro do algoritmo, tipicamente N = número de *cores* da máquina.

2. Ao receber para processamento uma partição P_i, divida em N subpartições com tamanhos iguais $P_i = P_{i,0}, P_{i,1}, ...P_{i,N}$ (caso não seja divisível, a última subpartição pode conter tuplas extras, na quantidade do resto da divisão $|P_i|/N$).

3. Para cada *core* execute a consulta em uma subpartição $P_{i,j}$, de forma paralela.

4. Envie os resultados do processamento sobre as partições, $Res(P_{i,0}), Res(P_{i,1}), ...Res(P_{i,N})$ para a composição de resultados.

5. Aguarde a próxima partição P_{i+1} e retorne para o passo 2, caso não exista, termine.

Algorithm 1 AVPMSimples

```
N ← n                                   ▷ n é tipicamente o número de cores da máquina
i ← 0
while P_i ≠ null do                     ▷ P_i é a partição i
    limite ← |P_i|//N                   ▷ divisão inteira
    resto ← |P_i|%N                     ▷ resto da divisão
    for j = 0, 1, 2, ... N do
        if j ≠ N − 1 then
            P_{i,j} ← Fragmenta(P_i, j × limite, (j + 1) × limite)
        else
            P_{i,j} ← Fragmenta(P_i, j × limite, resto + (j + 1) × limite)
        end if
        Res(P_{i,j}) ← Thread.processa(P_{i,j})     ▷ processamento em paralelo
        compoeResultados(Res(P_{i,j}))
    end for
    i ← i + 1
end while
```

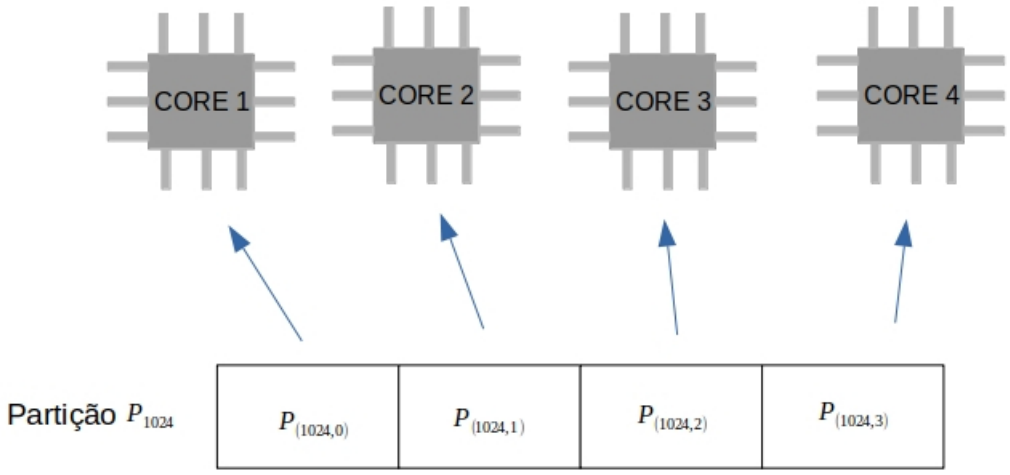

Figura 4.1: Ilustração do funcionamento da AVPMSimples processando a partição 1024.

4.4 AVPMRandômica

O fato da AVPMSimples ignorar o estágio da AVP e o tamanho atual da partição usada levou à elaboração de um algoritmo que considerasse isso, este é a AVPM-Randômica.

Este algoritmo tem uma abordagem diferente do anterior, ao invés de dividir a partição em subpartições na quantidade de *cores*, ele divide em um número arbitrário, que constituem uma fila a ser processada pelos *cores*. À medida que um *core* termina sua subpartição ele pega a próxima ainda não processada da fila, até que não haja mais nenhuma. Devido à sua natureza dinâmica, as subpartições futuras pode ter seu tamanho ajustado durante a execução, o que acaba alterando o tamanho da fila.

O algoritmo considera que o tamanho ótimo das subpartições depende do tamanho atual da partição, assim como do estágio da AVP (estável ou aumentando a partição). Como não se sabe nenhum modelo que governe o seu desempenho, esse algoritmo é dinâmico na busca por um tamanho supostamente ótimo de subpartição.

O algoritmo deve ser adaptativo e ao encontrar o tamanho de partições supostamente ótimo, deve sempre investigar se esse tamanho não mudou, para maior ou para menor, já que a AVP pode ter evoluído para um novo estágio. Para implementar tal comportamento foi escolhida uma natureza randômica na busca de novos tamanhos,

ou seja, o algoritmo sempre busca aleatoriamente tamanhos de subpartições maiores ou menores que o atual, e caso encontre um que melhore o desempenho, o torna o tamanho atual.

Comparando com o algoritmo ingênuo AVPMSimpes, percebe-se que este algoritmo apresenta uma perda de desempenho ao procurar pelo tamanho de subpartições ótimo por que nesse processo de busca, é necessário processar várias subpartições com um tamanho que pode ser bem distante do ótimo. Observe que sempre que o estado da AVP muda, o tamanho ótimo pode mudar e pode haver essa perda de desempenho recorrente. Apesar de encontrar um tamanho bom, não existem garantias de que é de fato o tamanho ótimo, pode ser por exemplo um máximo local.

Para explicar como esse algoritmo funciona é recomendável pensar em termos de tamanho das subpartições, ao invés de subpartições individuais. Um tamanho é gerado e dividido igualmente entre os *cores* disponíveis, estes começam a requisitar subpartições para processarem e o primeiro que termina anota o tempo de execução como sendo o tempo de execução daquele tamanho de subpartições. Compara-se o tempo de execução deste tamanho de subpartições com o anterior decide-se pelo tamanho que obteve o melhor tempo. Em seguida um novo tamanho randômico é gerado. Note que essa abordagem evita barreiras na programação concorrente, pois variantes desse algoritmo usando barreiras foram implementadas em testes preliminares e apresentaram desempenho inferior.

Três parâmetros *ad hoc* são usados no algoritmo, o primeiro é o tamanho inicial das subpartições, que foi escolhido para ser 1024. A escolha deste valor se deve ao fato de uma subpartição final, ou encontrada como supostamente ótima, tipicamente possuir quantidades acima de 16.000 tuplas e o valor escolhido é bem inferior, como se espera de um valor inicial que será ajustado pelo algoritmo adaptativo. O tamanho mínimo de uma partição da AVP é 1024, portanto não possui nenhum efeito adotar um valor menor no início da fase de processamento.

O segundo parâmetro é a tolerância que os tempos de execução de tamanhos de subpartições podem diferir até que se conclua que um é maior que o outro, essa tolerância é de 15 %. A escolha desse valor foi baseada em execuções do algoritmo em ambiente de testes e observação de que esse valor provocava um funcionamento correto e esperado do algoritmo.

O terceiro é o intervalo no qual os novos tamanhos de subpartições são gerados de forma randômica, que é 5 %, ou seja, os tamanhos são gerados no intervalo $[\text{tamanho anterior} * (1 - 0.05), \text{tamanho anterior} * (1 + 0.05)]$. A escolha desse parâmetro também foi baseada na experiência obtida em ambiente de testes. O valor não é tão pequeno, o que provocaria uma convergência lenta para o valor final, e não é tão grande, o que provocaria grandes oscilações na evolução do tamanho das

subpartições.

O algoritmo está descrito abaixo de duas formas, em português e em pseudocódigo. Esta última forma possui algumas funções auxiliares já usadas no algoritmo AVPMSimples, e além dessas possui as funções extras:

- $Time()$ - Captura a data e horário atual.
- $Random(incio, fim)$ - Gera um número pseudoaleatório do tipo *float* no intervalo $[incio, fim]$.

AVPMRandômica em português.

1. Inicie com os valores tamanho das subpartições = 1024, tolerância = 0.15 e limite de variação = 0.05.
2. Cada *thread*, requisite uma subpartição da fila para processarem, e após a primeira *thread* terminar o tempo de processamento por tupla é anotado: tempo atual = tempo de processamento/número de tuplas processadas.
3. Gere um novo tamanho atual de subpartições no intervalo [tamanho anterior $*$ $(1 - 0.05)$, tamanho anterior $*$ $(1 + 0.05)$].
4. Cada *thread*, requisite uma subpartição da fila para processarem, e após a primeira *thread* terminar o tempo de processamento por tupla é anotado: tempo atual = tempo de processamento/número de tuplas processadas.
5. Se tempo atual $>$ tempo anterior $*$ $(1$ + tolerância$)$ então tamanho das subpartições atual = tamanho das subpartições anterior. Em caso contrário o tamanho das subpartições atual se mantém.
6. Caso restem uma quantidade menor de tuplas a serem processadas que o tamanho atual, tamanho atual = quantidade restante de tuplas
7. Enquanto houver tuplas a serem processadas, volte ao estágio 3.

Algorithm 2 AVPMRandômica

$N \leftarrow n$ ▷ n é tipicamente o número de *cores* da máquina
$tolerancia \leftarrow 0.15$
$limiteVariacao \leftarrow 0.05$
$tamanhoSubparticoes \leftarrow 1024$
$tempoProcessamentoAnterior \leftarrow 0$
$tamanhoSubparticoesAnterior \leftarrow 0$
while $P_i \neq null$ **do** ▷ P_i é a partição i
  $tuplasProcessadas \leftarrow 0$
  while $tuplasProcessadas < |P_i|$ **do**
    if $tamanhoSubparticoes > |P_i| - tuplasProcessadas$ **then**
      $tamanhoSubparticoes \leftarrow |P_i| - tuplasProcessadas$
    end if
    $limite \leftarrow tamanhoSubparticoes//N$ ▷ divisão inteira
    $resto \leftarrow tamanhoSubparticoes\%N$ ▷ resto da divisão
    $tempoProcessamento \leftarrow Time()$
    for $j = 0, 1, 2, \ldots N$ **do**
      if $j \neq N - 1$ **then**
        $P_{i,j} \leftarrow Fragmenta(P_i, j \times limite, (j+1) \times limite)$
      else
        $P_{i,j} \leftarrow Fragmenta(P_i, j \times limite, resto + (j+1) \times limite)$
      end if
      $Res(P_{i,j}) \leftarrow Thread.processa(P_{i,j})$ ▷ processamento em paralelo
      $comopoeResultados(Res(P_{i,j}))$
    end for
    $tempoProcessamento \leftarrow Time() - tempoProcessamento$
    if $tamanhoSubparticoesAnterior \neq 0$ & $\frac{tempoProcessamento}{tamanhoSubparticoes} > \frac{tempoProcessamentoAnterior \times (1+tolerancia)}{tamanhoSubparticoesAnterior}$ **then**
      $tamanhoSubparticoes \leftarrow tamanhoSubparticoesAnterior$
    end if
    $tempoProcessamentoAnterior \leftarrow tempoProcessamento$
    $tamanhoSubparticoesAnterior \leftarrow tamanhoSubparticoes$
    $tuplasProcessadas \leftarrow tuplasProcessadas + tamanhoSubparticoes$
    $tamanhoSubparticoes \leftarrow tamanhoSubparticoes + Random(-1, +1) \times tamanhoSubparticoes \times limiteVariacao$
  end while
  $i \leftarrow i + 1$
end while

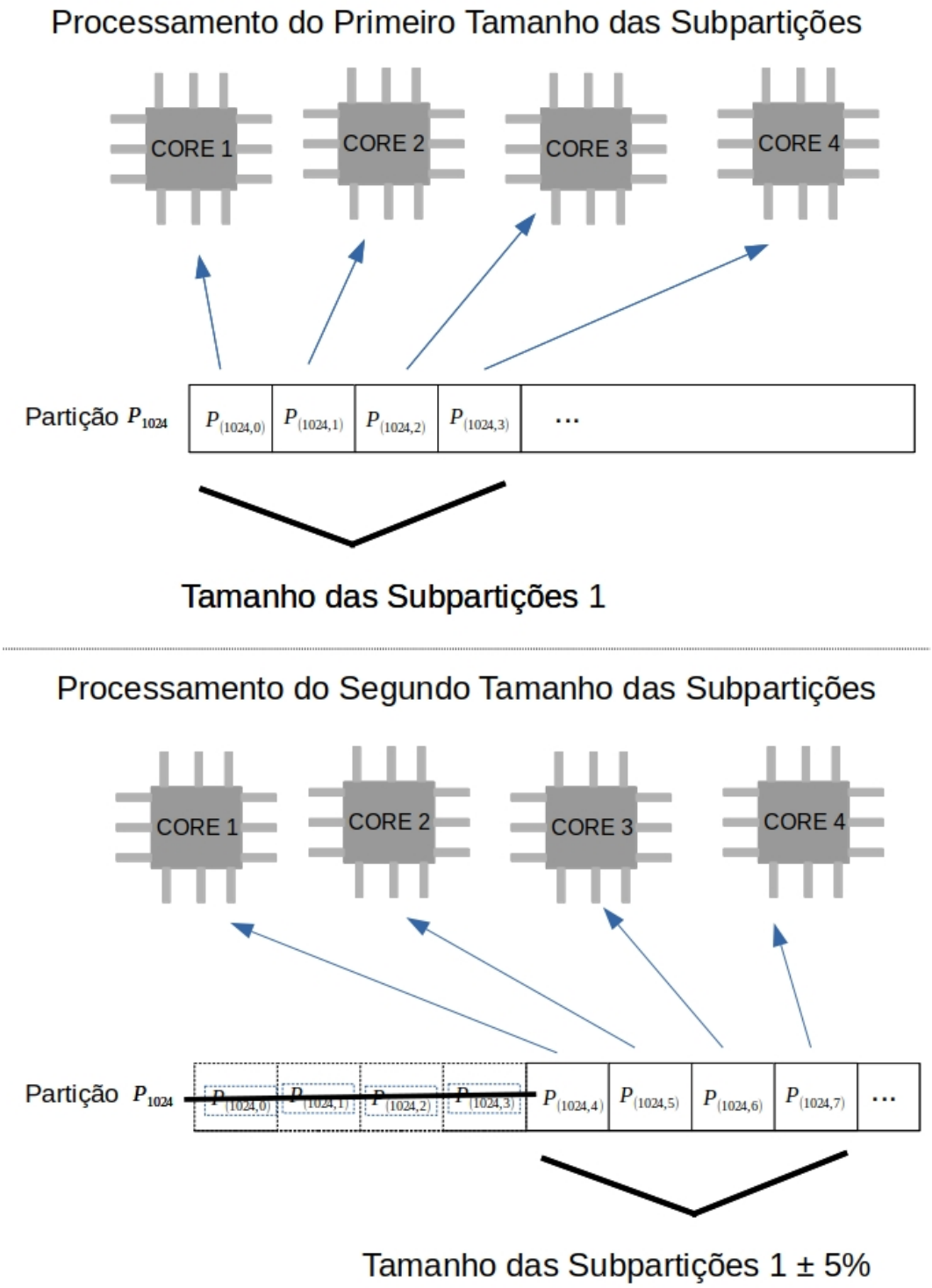

Figura 4.2: Ilustração do funcionamento da AVPMRandômica processando a partição 1024 com duas etapas, o processamento do primeiro e segundo tamanhos das subpartições.

Capítulo 5

Avaliação Experimental

Como a implementação dos dois algoritmos *multicore* foram feitas com base na implementação da AVP ParGRES, daqui em diante será referido o nome "ParGRES" para designar o ParGRES já com capacidades *multicores* implementadas. Já o termo "ParGRES original" será usado para designar o ParGRES original, sem capacidades *multicores*. Em todos os casos testados, os testes *mutilcore* englobam os dois algoritmos implementados, a AVPMSimples e a AVPMRandômica.

Os experimentos foram elaborados com duas propostas de testes em mente, a primeira é o uso do ParGRES servindo como *middleware* em um nó único, esse caso de uso permite utilizar um SGBD que não possui capacidade *multicore* como caixa preta delegando ao ParGRES a funcionalidade de promover o processamento de consultas paralelo. A ideia é permitir o paralelismo de consultas em computadores comuns que possuem múltiplos núcleos e executam algum SGBD sem capacidades multicores.

O segundo caso de uso que foi testado é o ParGRES em um *cluster* com arquitetura *shared nothing*, que é em seu uso típico, porém com capacidades *multicore* implementadas com o objetivo de promover o paralelismo e aumentar ainda mais o desempenho das consultas.

Ao contrário do primeiro caso de uso, a viabilidade dos algoritmos *multicore* propostos dependem do seu desempenho superar o do ParGRES original, se isso não ocorrer não existe justificativa para o seu uso. Por esse motivo os testes traçam um comparativo entre os algoritmos *multicore* e o ParGRES original, variando-se a quantidade de nós e *cores*.

Um parâmetro importante que se deseja saber ao fazer testes experimentais é a partir de quantos cores, supostamente, o ParGRES multicore supera o ParGRES original. Isso por que a implementação das capacidades multicore provoca alguma perda de desempenho com a criação de novas estruturas e sincronia do paralelismo, portanto espera-se que o ParGRES original supere o multicore para uma pequena quantidade de cores.

Os testes foram executados com o ParGRES original e *multicore*, variando a quantidade de nós da rede entre os valores 1, 2, 4, 8 e a quantidade de *cores*, para a implementação *multicore*, entre os valores 1, 2, 4, 8 e 16. Para estes testes o PostgreSQL foi configurado de forma a impedir o processamento *multicore* nativo, ou seja, em modo *singlecore*.

O foco dos experimentos é demonstrar o desempenho das implementações *multicores* em comparação com a AVP original, mas ao mesmo tempo os testes também mostram a comparação entre os dois algoritmos *multicore* implementados, a AVPMSimples e a AVPMRandom. A AVPMSimples sempre produz um tamanho de subpartição $P_{M,N}$ igual ao tamanho da partição P_M dividida pelo número de *cores* N, porém o tamanho das subpartições produzidas pela AVPMRandom é variável. Para avaliar a evolução do tamanho dessas subpartições, foi produzido um log que mostra o tamanho em cada ciclo de execução de consultas, ou seja, em cada execução de um tamanho de partições. O objetivo deste log é analisar e responder as perguntas:

- O tamanho de partições converge para algum valor? É importante saber pois esse é o melhor valor encontrado pelo algoritmo.
- O tamanho de partições se mantém ou oscila no decorrer da consulta, a depender da partição da base consultada?

5.1 PostgreSQL 11.9

Apesar de o ParGRES ter sido desenvolvido usando uma versão anterior do PostgreSQL, a versão mais recente que foi utilizada e testada por mais de um usuário é a 11.9 (incluindo o autor deste trabalho), por esse motivo o SGBD subjacente utilizado foi o PostgreSQL 11.9.

A versão do PostgreSQL utilizada já possui capacidades *multicores*, e para os testes foi necessário configurá-lo de forma específica. Em todos os testes o PostgreSQL foi mantido com um *core* único para processar as consultas. A configuração se deu pela edição do arquivo *postgresql.conf* que se encontra dentro do diretório da base de dados do PostgreSQL. O conteúdo original obtido da distribuição do PostgreSQL foi mantido, e pode ser visto no apêndice A.

Para o caso de rodar o ParGRES *multicore*, com o PostgreSQL *singlecore* deve-se acrescentar ao final do arquivo de configuração os parâmetros *max_worker_processes = 0*, *max_parallel_workers = 0* e *max_parallel_workers_per_gather = 0*. O trecho acrescido ao final do arquivo pode ser visto no apêndice A.1.

Observe que basta acrescentar ao final do arquivo de configuração para os parâmetros fazerem efeito, pois mesmo que já estejam declarados no início do arquivo o PostgreSQL sempre considera a última atribuição dos parâmetros.

5.2 TPC-H

O TPC-H[20] é um *benchmark* elaborado pela organização TPC voltado para apoio à tomada de decisões, com consultas *ad hoc* com alto grau de complexidade, que examinam grandes volumes de dados. A especificação das tabelas assim como o *software qgen* que gera a base, e as consultas podem ser encontradas em [21].

O *benchmark* TPC-H é muito popular como base usada para avaliação de desempenho, diversos trabalhos com propostas para processamento de consultas OLAP ou cargas mistas (OLAP + OLTP) o utilizam, como por exemplo [22], [23], [24] e [25]. Como o ParGRES pertence a categoria de sistemas projetados para consultas OLAP o uso do TPC-H permite, em algum grau, a comparação entre estas soluções.

O TPC-H especifica oito tabelas, sendo as tabelas de dimensão *Region*, *Nation*, *Supplier*, *Part*, *Customer* e *Partsupp* e as tabelas fato *Orders* e *Lineitem*. Ao gerar a base de dados com o *qgen* é especificado o parâmetro de escala *SF* para dimensionar a base gerada. A quantidade de registros gerados em cada tabela: $|Supplier| = SF * 10.000$, $|Customer| = SF * 150.000$, $|Part| = SF * 200.000$, $|Partsupp| = SF*800.000$, $|Orders| = SF*1.500.000$, $|Lineitem| = SF*6,001,215$, $|Region| = 5$ e $|Nation| = 25$.

No presente trabalho foi utilizado o valor SF=190, de modo a gerar uma base com aproximadamente 300GB.

O TPC-H possui 22 consultas, algumas de atualização e outras analíticas, porém nesse trabalho foram escolhidas apenas as consultas Q1, Q4, Q6, Q12 e Q18 que representam um conjunto de consultas OLAP com diferentes complexidades, apenas a Q18 sofreu uma alteração, foi retirada a cláusula *"LIMIT 100"* devido à limitação do ParGRES, que não possui capacidade de processar tal consulta. A seguir estão as consultas:

Q1:

```
select l_returnflag, l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1-l_discount))
                        as sum_disc_price,
    sum(l_extendedprice * (1-l_discount) * (1+l_tax))
                                        as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
```

```
lineitem
where l_shipdate <= date '1998-12-01' - interval '90 day'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Q4:

```
select o_orderpriority, count(*) as order_count
from orders
where o_orderdate >= date '1993-07-01'
    and o_orderdate < date '1993-07-01' + interval '3 month'
    and exists ( select * from lineitem
        where l_orderkey = o_orderkey
        and l_commitdate < l_receiptdate )
group by o_orderpriority
order by o_orderpriority;
```

Q6:

```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_shipdate >= date '1994-01-01'
    and l_shipdate < date '1994-01-01' + interval '1 year'
    and l_discount between .06 - 0.01 and .06 + 0.01
    and l_quantity < 24;
```

Q12:

```
select l_shipmode,
        sum( case when o_orderpriority = '1-URGENT' or
                       o_orderpriority = '2-HIGH'
                  then 1 else 0
             end ) as high_line_count,
        sum( case when o_orderpriority <> '1-URGENT' and
                       o_orderpriority <> '2-HIGH'
                  then 1 else 0
             end ) as low_line_count
from
orders, lineitem
where o_orderkey = l_orderkey
    and l_shipmode in ('MAIL', 'SHIP')
    and l_commitdate < l_receiptdate
    and l_shipdate < l_commitdate
```

```
    and l_receiptdate >= date '1994-01-01'
    and l_receiptdate < date '1994-01-01' + interval '1 year'
group by l_shipmode
order by l_shipmode;
```

Q18:

```
select c_name, c_custkey, o_orderkey,
        o_orderdate, o_totalprice, sum(l_quantity)
from customer, orders, lineitem
where o_orderkey in (select l_orderkey from lineitem
    group by l_orderkey
    having sum(l_quantity) > 300 )
    and c_custkey = o_custkey and o_orderkey = l_orderkey
group by c_name, c_custkey, o_orderkey,
        o_orderdate, o_totalprice
order by o_totalprice desc, o_orderdate
limit 100;
```

De acordo com [26], existem duas formas de testar sistemas paralelos, a primeira é a forma cache-transiente que ocorre quando os dados do processamento não estão na memória cache. A segunda é a cache-persistente, que ocorre quando os dados do processamento estão na memória cache, normalmente esta última forma possui melhor desempenho. A forma correta de fazer testes é simulando a forma como o usuário típico vai usar o sistema, e no caso de um sistema de consultas OLAP é a forma cache-transiente pois as consultas são ad hoc e feitas somente uma vez em horários pré-definidos para relatórios, não costuma haver repetição e frequência. Logo as consultas foram executadas nessa ordem: Q1, Q4, Q6, Q12 e Q18 e foram repetidas 3 vezes, o desvio padrão foi mostrado para indicar possíveis pontos com alta variabilidade nos tempos de execução.

5.3 Implementação

Para implementar os algoritmos no ParGRES, foi necessário adaptar o componente arquitetural *Node Query Processor (NQP)* , que passou a possuir uma nova classe responsável pelo processamento de cada partição da AVP. Essa nova classe foi denominada *MultiCoreQueryExecutorAVPMSimpes.java* e *MultiCoreQueryExecutorAVPRAM.java* para os algoritmos AVPMSimples e AVPMRandom respectivamente.

Os diagramas de classes das figuras 5.1 e 5.2 mostram as classes com as relações que possuem com a classe *QueryExecutor.java*, que também sofreu pequenas modificações.

A classe *MultiCoreQueryExecutorAVPMSimpes.java* recebe uma partição de *QueryExecutor.java* para ser processada, a divide em N subpartições e envia cada uma para a classe *CoreAgentAVPMSimples.java*. Esta última implementa a interface *Runnable*, o que faz dela uma *thread* que pode ser executada em paralelo com outras classes do mesmo tipo, cada uma executando a consulta na sua respectiva subpartição. Após todas as *threads* terminarem sua tarefa, a classe *MultiCoreQueryExecutorAVPMSimpes.java* recebe os resultados parciais e retorna uma lista.

A classe *MultiCoreQueryExecutorAVPRAM.java* recebe uma partição de *QueryExecutor.java* para ser processada, a divide em N subpartições que compõem o tamanho das subpartições. Em seguida cada subpartição fica em uma fila que recebe requisições da classe *CoreAgentAVPRAM.java* que por sua vez implementa a interface *Runnable*, o que faz dela uma *thread* que pode ser executada em paralelo com outras classes do mesmo tipo. Após cada classe terminar sua subpartição o resultado é enviado para a classe *MultiCoreQueryExecutorAVPRAM.java* que os armazena em uma lista, uma nova subpartição é requisitada pela classe *CoreAgentAVPRAM.java* até que toda a partição seja processada e a tarefa se encerre.

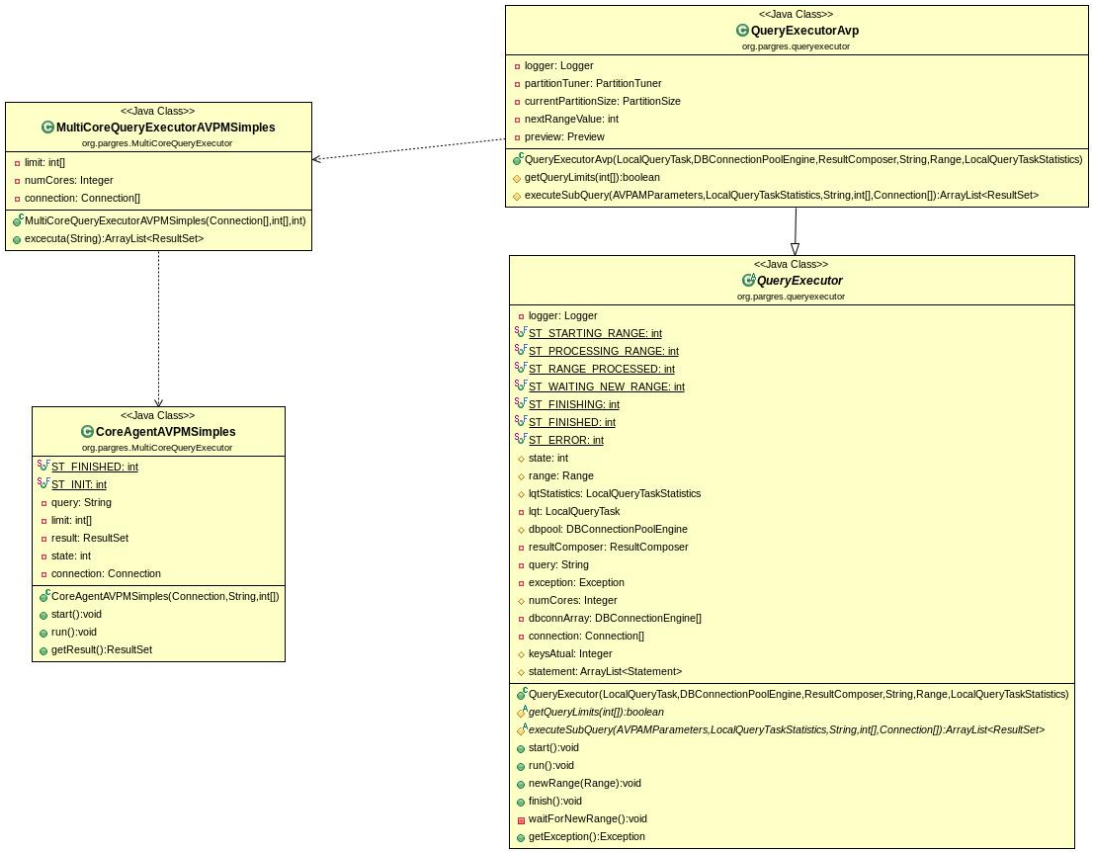

Figura 5.1: Diagrama de classes com a classe *MultiCoreQueryExecutorAVPMSimpes.java* e suas ligações.

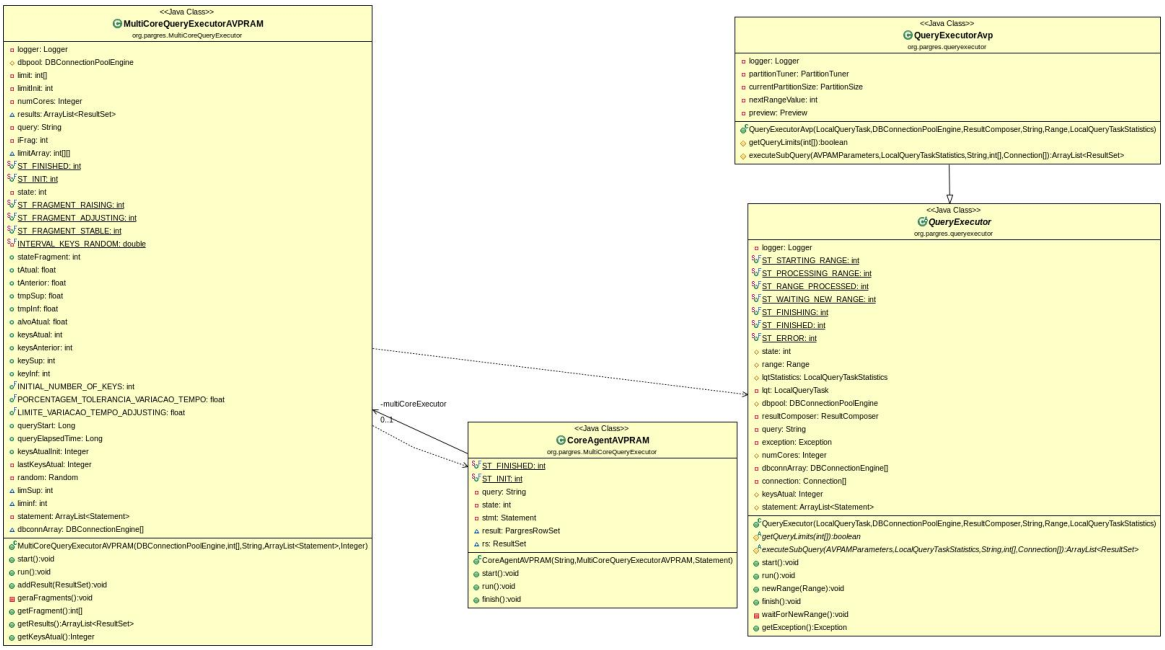

Figura 5.2: Diagrama de classes com a classe *MultiCoreQueryExecutorAVPRAM.java* e suas ligações.

Diversas outras modificações de menor alcance foram necessárias nos componente de Log, na parte responsável pela criação de conexões com a base de dados, na leitura e processamento de opções de configuração e na composição de resultados.

5.4 Configuração

Todos os testes foram executados no supercomputador Lobo Carneiro - SGI ICE X, pertencente ao Núcleo Avançado de Computação de Alto Desempenho (NACAD)[27] da COPPE/UFRJ, parte integrante do Sistema Nacional de Processamento de Alto Desempenho (SINAPAD), constituindo-se em seu Laboratório de Serviço Especializado em Engenharia.

O Lobo Carneiro possui as seguintes especificações:

- 504 CPUs Intel Xeon E5-2670v3 (Haswell): 6048 Cores
- Cores/Nó Processamento (reais): 24 Cores
- Cores/Nó processamento com Hyper-Threading (HT): 48 cores
- Total de Nós de processamento: 252
- Memória por nó de processamento: 64 GBytes
- Total de Memória RAM: 16 TBytes (distribuída)

- Sistema de arquivo paralelo: Intel Luster (500 TBytes)
- Armazenamento em disco: 60 TBytes
- Rede: Infiniband FDR - 56 Gbs (Hypercube)
- Sistemas operacional: Suse Linux Enterprise (SLE)

5.5 Resultados e Discussões

Antes de expor os resultados das execuções dos algoritmos, cabe refletir e enumerar as razões que podem levar qualquer algoritmo a ter um desempenho melhor ou pior que outro, gargalos da execução e características da implementação.

Em relação aos algoritmos propostos, os seguintes fatores podem influenciar no desempenho:

- Estruturas e fatores de programação - A implementação dos algoritmos propostos sobre o ParGRES original envolve, além de outras modificações, a criação de uma classe que gerencia a execução *multicore* e de *threads*. A classe que gerencia, no caso da AVPMSimples é muito simples, mas no caso da AVPMRandom é responsável por receber os tempos de execução das threads e calcular os próximos tamanhos das subpartições. Apesar de nehum algoritmo implementado ser bloqueante, existem variáveis com travas de escrita para sincronia do paralelismo.
- Composição dos resultados - A composição dos resultados está descrita na tabela 2.1. Nota-se que essa composição é dependente dos resultados parciais obtidos de cada processamento de cada subpartição. O número de subpartições, no caso da AVPMSimples é o número de partições multiplicado pelo número de *cores*, porém no caso da AVPMRandom é multiplicado pelo número de subpartições que o algoritmo determinar. Esse fator pode aumentar o tempo de composição de resultados.
- Condições de contorno - O algoritmo AVPMRandom em particular decide um tamanho das subpartições para serem processadas pelos *cores*, porém esse tamanho pode ser apenas um pouco menor que o tamanho da partição, de forma que falte uma pequena porção da partição a ser processada. Essa pequena porção pode ser de um tamanho não ótimo (de fato pode ser bem distante do ótimo), e não é uma escolha do algoritmo, apenas uma condição de contorno que aparece no fim do processamento de cada partição.

- Varreduras completas - O SGBD pode, a depender do contexto e da consulta a ser executada, fazer varreduras completas das tabelas. Esse já era um problema conhecido e evitado desde a FGVP.

Além dos fatores diretamente relacionados a algoritmos, não se pode ignorar que eles estão sendo executados em computadores com um sistema operacional que possui diversas rotinas sendo executadas de forma multitarefa, possivelmente com acessos concorrentes a recursos compartilhados como memória e cache. Logo é possível haver variação e até pontos fora da curva na execução das consultas.

Para cada consulta dentro do conjunto escolhido, foram feitas 3 análises para avaliar o trabalho e verificar se os objetivos foram alcançados. A primeira análise foi a de aceleração, foi feita uma figura (*e.g.* para a consulta Q1 5.3) dividida em dois gráficos: os algoritmos AVPMSimples e AVPMRandom. Cada gráfico possui como eixo vertical o tempo da consulta e horizontal a quantidade de cores. Variou-se a quantidade de cores pois o objetivo do trabalho é mostrar o comportamento *multicore*, e cada curva do gráfico corresponde a uma quantidade de nós usada na configuração. Em cada gráfico estão também os pontos com os tempos obtidos com a execução do ParGRES original, variando-se a quantidade de nós e com apenas 1 *core*.

O eixo horizontal do gráfico está em escala logarítmca para facilitar a visualização de aceleração com a variação de *cores*, porém essa escala dificulta a visualização de eventuais acelerações com valores de tempo (eixo vertical) baixos. Para possibilitar uma melhor comparação numérica também estão presentes três tabelas, uma do ParGRES original, outra com o algoritmo AVPMSimples e outra do AVPMRandom (*e.g.* tabelas 5.1, 5.2 e 5.3 para a Q1), que possuem os mesmos dados mostrados no gráfico. Todos os dados obtidos nestes gráficos e nas tabelas possuem o desvio padrão para evidenciar possíveis comportamentos de grandes variações de tempo.

A segunda análise foi comparar o tamanho da partição, que é o mesmo que o tamanho das subpartições para a AVPMSimples, com o tamanho das subpartições para a AVPMRandom (*e.g.* para a consulta Q1 o gráfico pode ser visto em 5.4). O objetivo destes gráficos é evidenciar a tendência constatada do tamanho das subpartições do AVPMRandom convergir para o tamanho da partição. Em outras palavras, o algoritmo AVPMRandom busca de forma dinâmica um tamanho de subpartições candidato a ótimo, e tende a encontrar o tamanho da partição, que é o tamanho usado pelo algoritmo ingênuo AVPMSimples. Neste gráfico o eixo vertical possui o tamanho em quantidade de tuplas, e o eixo horizontal possui a contagem da subpartição processada pela AVPMRandom.

Em relação a estes gráficos da segunda análise, a AVPMRandom processou dezenas de milhares, e para algumas consultas até centenas de milhares de subpartições, no entanto tal quantidade de dados não cabe em uma imagem e geram um gráfico

no qual não se pode visualizar nenhuma informação. Por isso foram selecionadas apenas as primeiras 500 subpartições. O comportamento da execução do algoritmo na completude segue o mesmo padrão destas 500 subpartições iniciais, sendo as conclusões deste gráfico válidas.

Para a confecção do gráfico da segunda análise, foi eliminada o tamanho da subpartição de contorno. Observe abaixo o trecho do log de execução de onde os dados para este gráfico foram extraídos, ele mostra o processamento de uma partição que foi dividida em 6 tamanhos de subpartição, cada um a ser processado em paralelo pelos *cores*. No último tamanho de subpartição o valor de *"size"* expressa o tamanho da partição que é 16384, o valor de *"before"* expressa a escolha do algoritmo pelo tamanho de subpartição 3833, porém o valor de *"after"* revela que, devido à condição de contorno, o tamanho da subpartição foi 3585. Esse último valor de *"after"* é um dos que foram retirados do gráfico.

```
DEBUG  - VP: 31,745 - 48,129. Size = 16,384
DEBUG  - size, before, after : 16,384 1,095 1,095
DEBUG  - size, before, after : 16,384 2,899 2,899
DEBUG  - size, before, after : 16,384 1,738 1,738
DEBUG  - size, before, after : 16,384 2,907 2,907
DEBUG  - size, before, after : 16,384 4,160 4,160
DEBUG  - size, before, after : 16,384 3,833 3,585
DEBUG  - QE AVP - Elapsed Time: 38 Estimated time: 565.0/
```

A exibição desse valor de contorno estava tornando o gráfico de difícil visualização, com muitos vales profundos, com um dado que não reflete a escolha do algoritmo pelo tamanho de subpartição. Note que essa mesma particularidade pode provocar uma perda de desempenho da AVPMRandom em relação a AVPMSimples.

A terceira análise se trata da examinação e apresentação dos planos de consulta, que foram colhidos na execução de cada consulta. Foi feita a análise dos tempos e custos de cada etapa da execução.

Para cada uma das consultas analisadas, foram expostos dois planos de consulta. O primeiro plano possui seletividade de 16000 tuplas da tabela fato, que é um tamanho típico de uma partição, e dessa forma simula o processo de fragmentação que a AVP faz. O segundo plano possui seletividade de 1/4 de uma partição típica, ou seja, 4000 tuplas da tabela fato e simula o processo de criação de 4 subpartições (equivalente à AVPMSimples com 4 cores). Isso nos permite simular e avaliar como o SGBD lida com essa consulta modificada no processo de criação de subpartições à partir de uma partição.

5.5.1 Consulta Q1

A figura 5.3 e as tabelas 5.1, 5.2 e 5.3 possuem os tempos de execução da consulta Q1 usando diferentes configurações de nós e *cores*, para os algoritmos AVPMSimples e AVPMRandom, além do ParGRES original em apenas um nó.

Observa-se pelo gráfico uma grande variação, evidenciada pelo alto desvio padrão, e pelo gráfico mais sinuoso para pequenas quantidade de nós. Apesar disso as maiores acelerações são vistas nesses dados obtidos com poucos nós, com 1 nó por exemplo, a AVPMSimples reduziu o tempo de consulta de 36,11 para 13,45 minutos e a AVPMRandom reduziu de 35,07 para 17,11 minutos. Já em quantidades maiores de nós, a aceleração fica mais discreta e pode ser melhor visualizada pela tabela. Com 8 nós a AVPMSimples reduziu o tempo de 4,1 para 1.92 minutos, e a AVPMRandom de 4,19 para 2,02 minutos.

Outro fator relevante é observado entre as execuções com 1 e 2 *cores*, ocorreu tanto na AVPMSimples quanto na AVPMRandom com 2, 4 e 8 nós uma desaceleração ou aceleração mais suave. Isso se deve aos custos de composição de resultados e de estruturas e fatores de programação (e.g. travas de leitura e escrita de variáveis para sincronização) que crescem com o número de *cores*.

Comparando-se os resultados da AVPMSimples com a AVPMRandom, pode-se ver que a AVPMSimples foi melhor na maioria dos casos, mas por uma diferença de tempo bem pequena, e teve um comportamento qualitativo semelhante, apresentando o mesmo padrão. Os fatores condições de contorno e custo de composição explicam o desempenho inferior da AVPMRandom.

Comparando os algoritmos *multicore* com o ParGRES original, observa-se que os algoritmos *multicore* possuem um desempenho ligeiramente inferior na configuração usando apenas 1 *core*, e apenas em dois casos (AVPMSimples e AVPMRandom com 2 nós) o desempenho com dois *cores* também foi superado pelo ParGRES original, devido à perdas de desempenho da implementação *multicore*. Em geral houve aceleração e o uso do paralelismo *multicore* apresentou melhores resultados que o ParGRES original, com diferenças mais acentuadas para números elevados de *cores*.

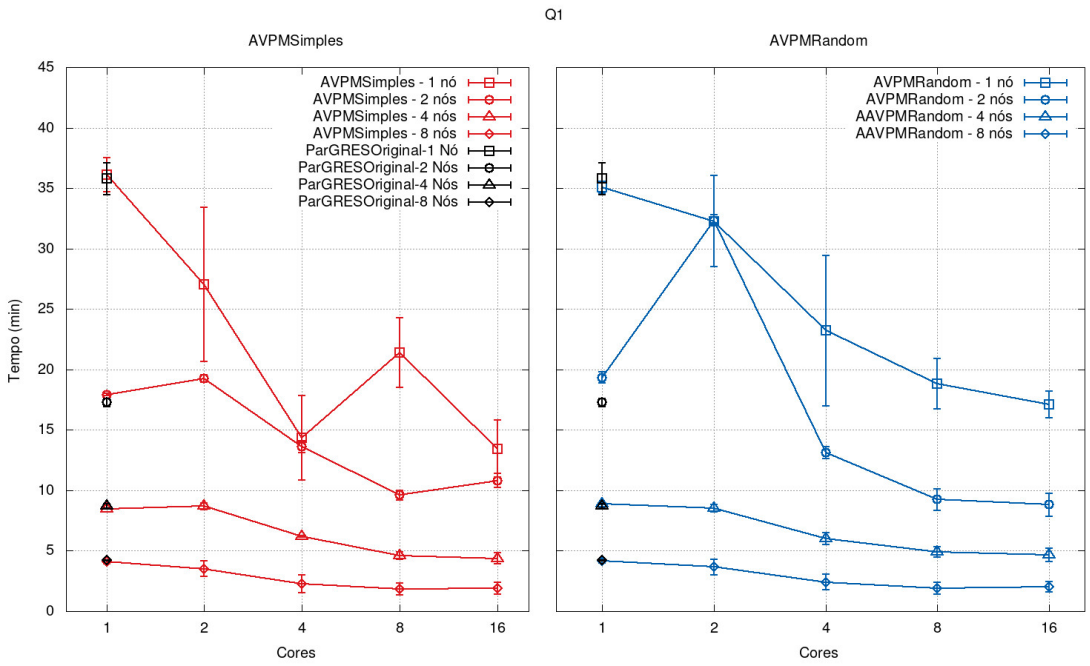

Figura 5.3: Comparação entre o ParGRES original e a aceleração dos algoritmos AVPMSimples e AVPMRandom para a consulta Q1.

Q1 - ParGRES Original	
Nós/Cores	1 Core
1 Nó	$35,78 \pm 1,3$
2 Nós	$17,26 \pm 0,36$
4 Nós	$8,69 \pm 0,17$
8 Nós	$4,22 \pm 0,05$

Tabela 5.1: Tempos de execução ± desvio padrão (em minutos) da consulta Q1 usando o ParGRES original, variando-se os nós com a quantidade fixa de 1 *core*.

Q1 - AVPMSimples					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$36,11 \pm 1,43$	$27,04 \pm 6,35$	$14,37 \pm 3,5$	$21,4 \pm 2,87$	$13,45 \pm 2,37$
2 Nós	$17,92 \pm 0,49$	$19,25 \pm 1,68$	$13,61 \pm 0,89$	$9,6 \pm 0,7$	$10,82 \pm 0,59$
4 Nós	$8,48 \pm 0,04$	$8,68 \pm 0,2$	$6,19 \pm 0,08$	$4,6 \pm 0,31$	$4,38 \pm 0,44$
8 Nós	$4,1 \pm 0,15$	$3,5 \pm 0,6$	$2,27 \pm 0,72$	$1,82 \pm 0,48$	$1,92 \pm 0,48$

Tabela 5.2: Tempos de execução ± desvio padrão (em minutos) da consulta Q1 usando o algoritmo AVPMSimples, variando-se os nós e *cores*.

Q1 - AVPMRandom					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$35,07 \pm 0,51$	$32,27 \pm 3,75$	$23,22 \pm 6,22$	$18,8 \pm 2,08$	$17,11 \pm 1,08$
2 Nós	$19,33 \pm 0,47$	$32,33 \pm 0,47$	$13,12 \pm 0,47$	$9,23 \pm 0,89$	$8,8 \pm 0,93$
4 Nós	$8,9 \pm 0,09$	$8,54 \pm 0,31$	$5,99 \pm 0,5$	$4,91 \pm 0,41$	$4,66 \pm 0,56$
8 Nós	$4,19 \pm 0,19$	$3,66 \pm 0,63$	$2,42 \pm 0,67$	$1,9 \pm 0,5$	$2,02 \pm 0,42$

Tabela 5.3: Tempos de execução ± desvio padrão (em minutos) da consulta Q1 usando o algoritmo AVPMRandom, variando-se os nós e *cores*.

O que pode-se ver no gráfico 5.4 é que o tamanho da partição logo evolui para pouco mais de 16000, já o tamanho das subpartições evolui de forma sinuosa, devido à sua natureza randômica, até o tamanho da partição. Em seguida, a AVPMRandom continua de forma dinâmica e randômica buscando novos tamanho das subpartições ótimos, o que produz as quedas no gráfico, porém na maioria dos casos ele retorna muito rapidamente para o tamanho da partição. Isso evidencia que a AVPMRandom converge ao comportamento da AVPMSimples e o tamanho das subpartições encontrado como o melhor tende a ser o tamanho da partição.

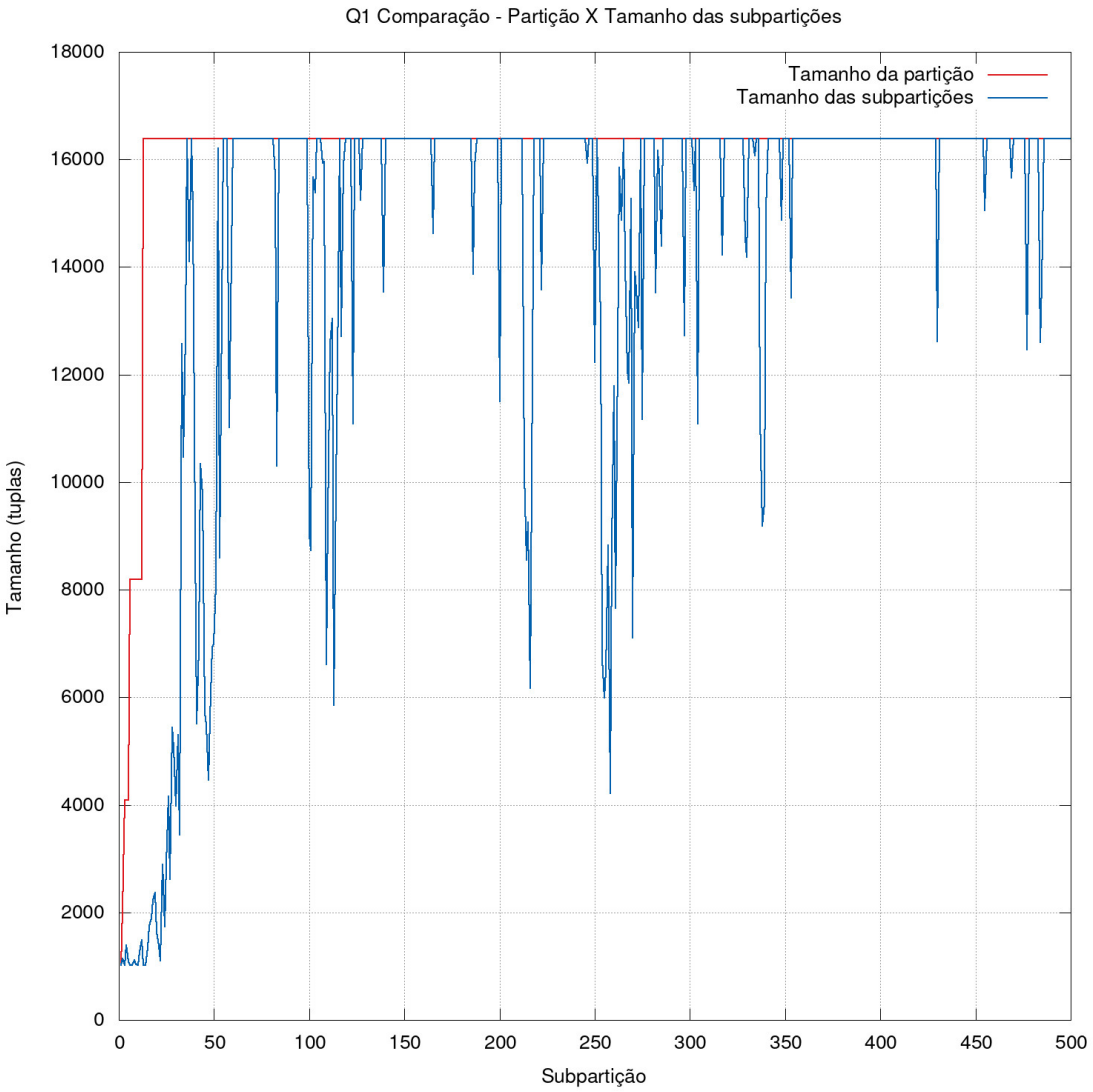

Figura 5.4: Comparação entre o tamanho das subpartições da AVPMSimples (igual ao tamanho da partição) com o da AVPMRandom, evoluindo durante o processamento da consulta Q1.

Abaixo estão os planos de consulta simplificados com análise dos tempos e custos de cada etapa, os planos completos podem ser vistos no apêndice B.1. Os planos são da consulta simulada sobre uma partição e selecionando somente um quarto da partição, respectivamente:

```
------------------------------------------------------------
HashAggregate
   ->  Index Scan
Planning Time: 243.535 ms
Execution Time: 166.102 ms
(8 rows)
------------------------------------------------------------
```

```
------------------------------------------------------------
HashAggregate
   ->  Index Scan
Planning Time: 19.156 ms
Execution Time: 6.074 ms
(8 rows)
------------------------------------------------------------
```

Analisando os planos, observa-se que em ambas as consulta, em partição e subpartição, houve aceleração significativa dos tempos de planejamento e execução.

Ao final, a consulta em partição leva 409 ms e a consulta sobre a subpartição leva 25 ms, ocorre uma aceleração significativa que é compatível com a aceleração mostrada no gráfico (5.3) da consulta Q1.

A consulta Q1 é uma consulta relativamente simples em comparação com outras, ela acessa apenas uma tabela, a *lineitem* que também é a única tabela fato acessada, possui diversas agregações e dois agrupamentos. Analisando o plano de consulta podemos ver que tanto a operação de varredura indexada quanto a agregação apresentaram aceleração significativa. A consulta apresentou aceleração e desempenho melhor com os vários *cores* que com o ParGRES original.

5.5.2 Consulta Q4

A figura 5.5 e as tabelas 5.4, 5.5 e 5.6 possuem os tempos de execução da consulta Q4 usando diferentes configurações de nós e *cores*, para os algoritmos AVPMSimples e AVPMRandom, além do ParGRES original em apenas um nó.

Observa-se pelo gráfico acelerações maiores e significativas para uma pequena quantidade de nós. Com 1 nó por exemplo, a AVPMSimples reduziu o tempo de consulta de 41,99 para 26,49 minutos e a AVPMRandom reduziu de 39,9 para 21,55 minutos. Conforme a quantidade de nós aumenta a aceleração vai ficando mais discreta, até que para 8 nós ou não ocorre nenhuma ou a aceleração não é significativa para ambos os algoritmos.

Outro fator relevante é observado entre as execuções com 1 e 2 *cores*, ocorreu tanto na AVPMSimples com 2, 4 e 8 nós quanto na AVPMRandom com 4 e 8 nós uma desaceleração ou aceleração mais suave. Isso se deve aos custos de composição de resultados e de estruturas e fatores de programação (e.g. travas de leitura e escrita de variáveis para sincronização) que crescem com o número de *cores*. Após essa perda inicial houve aceleração em todos os casos, inclusive para consultas com 8 nós.

Comparando-se os resultados da AVPMSimples com a AVPMRandom, pode-se ver que a AVPMRandom teve um desempenho melhor apenas com 1 nó e principal-

mente para grandes quantiaddes de *cores*, nos outros dados a AVPMSimples teve um desempenho melhor. Ambos tiveram um comportamento qualitativo semelhante.

Comparando os algoritmos *multicore* com o ParGRES original, a primeira coisa a se observar é que para 4 e 8 nós não houve aceleração dos algoritmos *multicore* e o desempenho foi muito próximo ao do ParGRES original, nos outros casos eles possuem um desempenho ligeiramente inferior na configuração usando apenas 1 *core*, e apenas em dois casos (AVPMSimples e AVPMRandom com 2 nós) o desempenho com 2 *cores* também foi superado pelo ParGRES original, devido à perdas de desempenho da implementação *multicore*. Para pequenas quantidades de nós houve aceleração e uso do paralelismo *multicore* apresentou melhores resultados que o ParGRES original, com diferenças significativas para números elevados de *cores*.

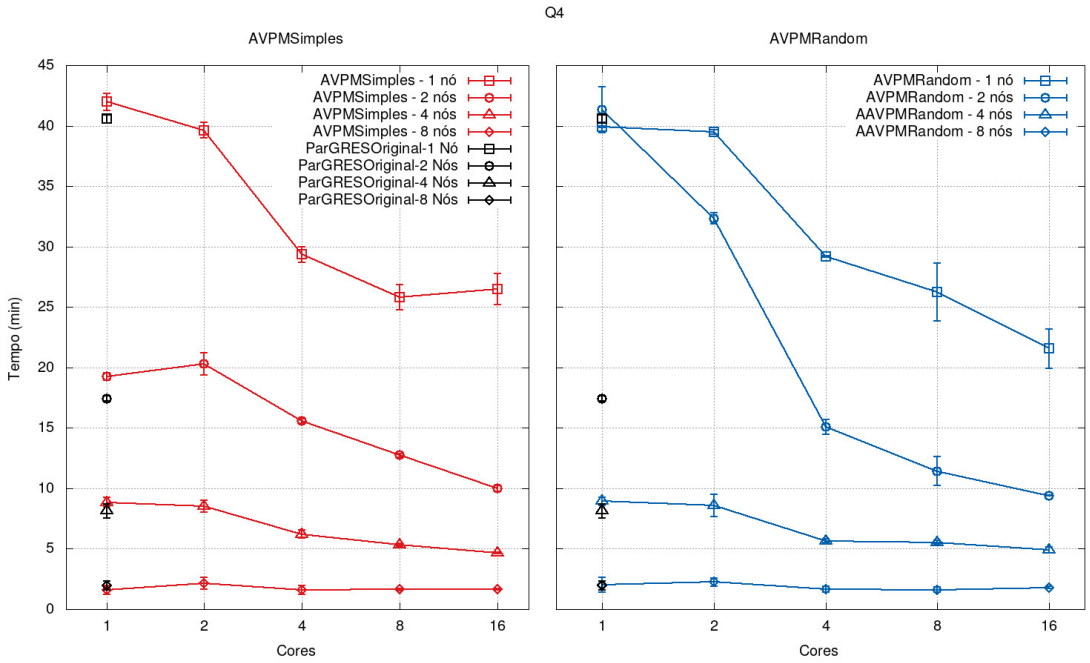

Figura 5.5: Comparação entre o ParGRES original e a aceleração dos algoritmos AVPMSimples e AVPMRandom para a consulta Q4.

Q4 - ParGRES Original	
Nós/Cores	1 Core
1 Nó	$40,57 \pm 0,34$
2 Nós	$17,44 \pm 0,2$
4 Nós	$8,13 \pm 0,56$
8 Nós	$1,97 \pm 0,36$

Tabela 5.4: Tempos de execução $\pm$ desvio padrão (em minutos) da consulta Q4 usando o ParGRES original, variando-se os nós com a quantidade fixa de 1 *core*.

Q4 - AVPMSimples					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$41,99 \pm 0,71$	$39,61 \pm 0,64$	$29,34 \pm 0,67$	$25,81 \pm 1,02$	$26,49 \pm 1,3$
2 Nós	$19,24 \pm 0,93$	$20,32 \pm 0,9$	$15,58 \pm 0,92$	$12,74 \pm 0,93$	$10,01 \pm 0,66$
4 Nós	$8,83 \pm 0,4$	$8,54 \pm 0,48$	$6,22 \pm 0,33$	$5,31 \pm 0,12$	$4,67 \pm 0,07$
8 Nós	$1,61 \pm 0,38$	$2,14 \pm 0,49$	$1,59 \pm 0,35$	$1,63 \pm 0,1$	$1,66 \pm 0,06$

Tabela 5.5: Tempos de execução ± desvio padrão (em minutos) da consulta Q4 usando o algoritmo AVPMSimples, variando-se os nós e *cores*.

Q4 - AVPMRandom					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$39,9 \pm 0,08$	$39,48 \pm 0,19$	$29,19 \pm 0,05$	$26,23 \pm 2,4$	$21,55 \pm 1,64$
2 Nós	$41,33 \pm 1,89$	$32,33 \pm 0,47$	$15,06 \pm 0,62$	$11,43 \pm 1,18$	$9,41 \pm 0,06$
4 Nós	$8,93 \pm 0,32$	$8,6 \pm 0,92$	$5,67 \pm 0,18$	$5,54 \pm 0,15$	$4,9 \pm 0,25$
8 Nós	$2,02 \pm 0,64$	$2,24 \pm 0,33$	$1,65 \pm 0,24$	$1,61 \pm 0,21$	$1,8 \pm 0,08$

Tabela 5.6: Tempos de execução ± desvio padrão (em minutos) da consulta Q4 usando o algoritmo AVPMRandom, variando-se os nós e *cores*.

O que pode-se ver no gráfico 5.6 é que o tamanho da partição logo evolui para pouco mais de 16000, já o tamanho das subpartições evolui de forma sinuosa, devido a sua natureza randômica, até o tamanho da partição. Em seguida, a AVPMRandom continua de forma dinâmica e randômica buscando novos tamanho das subpartições ótimos, o que produz as quedas no gráfico, porém na maioria dos casos ele retorna muito rapidamente para o tamanho da partição. Isso evidencia que a AVPMRandom tende a se comportar como a AVPMSimples e o tamanho das subpartições encontrado como o melhor tende a ser o tamanho da partição.

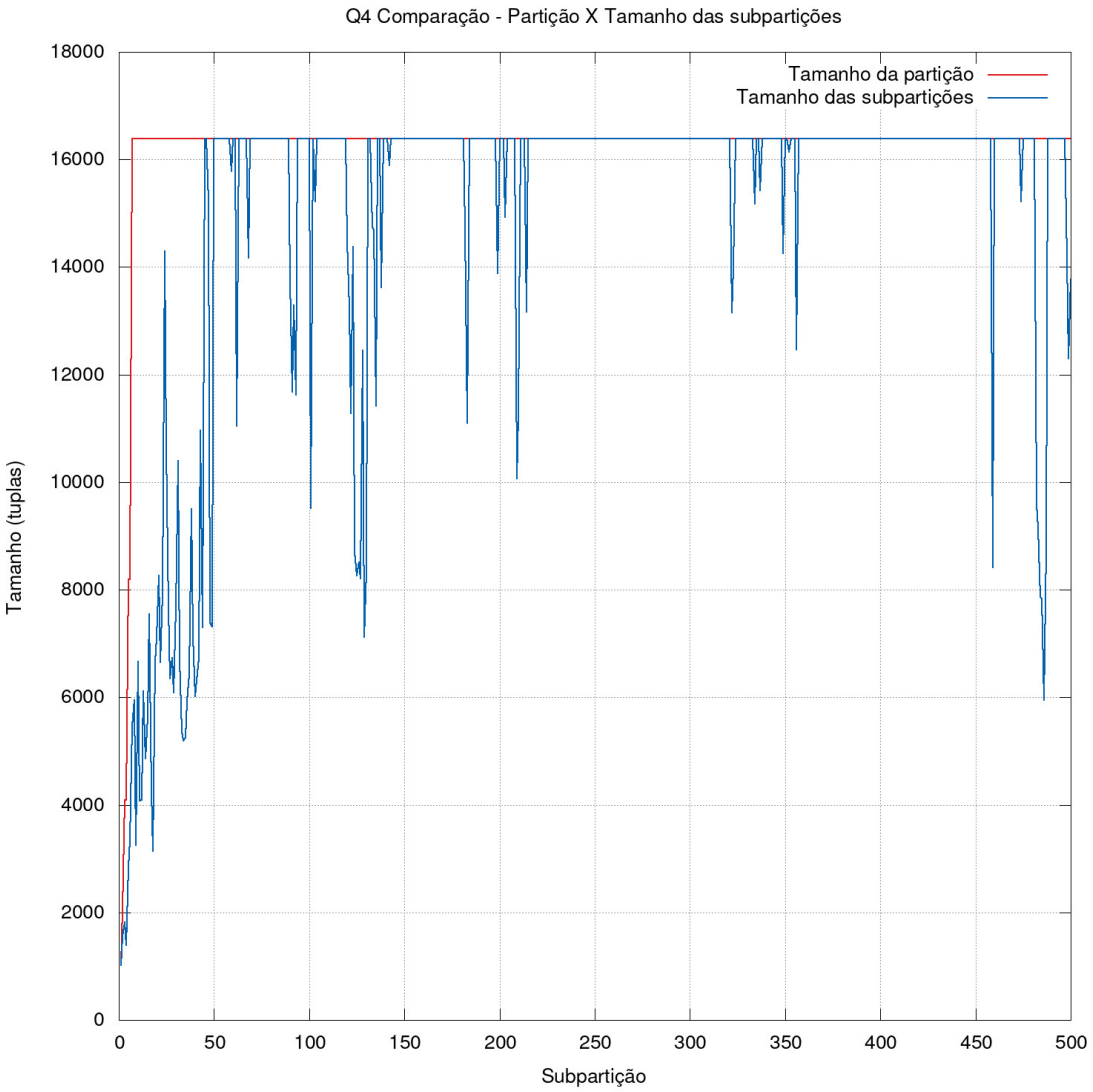

Figura 5.6: Comparação entre o tamanho das subpartições da AVPMSimples (igual ao tamanho da partição) com o da AVPMRandom, evoluindo durante o processamento da consulta Q4.

Abaixo estão os planos de consulta simplificados com análise dos tempos e custos de cada etapa, os planos completos podem ser vistos no apêndice B.2. Os planos são da consulta simulada sobre uma partição e selecionando somente um quarto da partição, respectivamente:

```
-----------------------------------------------------------------
GroupAggregate
   ->  Sort
         ->  Nested Loop Semi Join
               ->  Index Scan
               ->  Index Scan
Planning Time: 175.823 ms
```

```
Execution Time: 76.379 ms
(16 rows)
-------------------------------------------------------------

-------------------------------------------------------------
GroupAggregate
   ->  Sort
         ->  Nested Loop Semi Join
               ->  Index Scan
               ->  Index Scan
Planning Time: 26.343 ms
Execution Time: 0.585 ms
(16 rows)
-------------------------------------------------------------
```

Analisando os planos, observa-se que em ambas as consulta, em partição e subpartição, houve aceleração significativa dos tempos de planejamento e execução, com destaque para o tempo de execução que acelerou mais de 130 vezes.

Ao final, a consulta em partição leva 252 ms e a consulta sobre a subpartição leva 29 ms, ocorre uma aceleração significativa que é compatível com a aceleração mostrada no gráfico (5.5) da consulta Q4.

A consulta Q4 apresenta uma complexidade média e maior que outras consultas como a Q1, ela utiliza uma tabela fato, a *orders*, porém no predicado de seletividade recorre a uma sub-consulta em outra tabela fato, a *lineitem*. Além disso possui apenas uma agregação e um agrupamento. Ao analisar os planos de consultas, observa-se que houve aceleração significativa em todos as operações, com exceção de operações associadas à varredura da tabela fato *lineitem*, isso ocorre por que essa varredura está em uma subconsulta e é necessário a varredura completa, independente da seletividade do predicado que determina o tamanho da subpartição.

5.5.3 Consulta Q6

A figura 5.7 e as tabelas 5.7, 5.8 e 5.9 possuem os tempos de execução da consulta Q6 usando diferentes configurações de nós e *cores*, para os algoritmos AVPMSimples e AVPMRandom, além do ParGRES original em apenas um nó.

No gráfico fica bastante visível o comportamento de desacelerar na passagem de 1 *core* para 2, isso ocorre em todos os casos, e depois ocorre uma aceleração. O fato é que essa aceleração tardia só ocorre em relação ao dado obtido com 2 *cores*, sendo a aceleração total inexistente, tanto para o algoritmo AVPMSimples quanto o AVPMRandom. A única ocorrência de aceleração, para 2 nós do AVPMRandom,

pode representar um dado privilegiado pelas condições do hardware e sistema operacional no momento da execução, não se pode concluir que o algoritmo acelera em quaisquer condições.

Além disso os dados para uma pequena quantidade de nós apresentam maior variabilidade, o que pode ser observado tanto pela variação dos valores para cima e para baixo quanto pelos desvios padrão, que em particular no algoritmo AVPMRandom foram relevantes.

Comparando-se os resultados da AVPMSimples com a AVPMRandom, pode-se ver que ambos possuem desempenho muito parecido, com números muito próximos hora com a AVPMSimples mais rápida hora com a AVPMRandom mais rápida. A exceção é com 2 nós, nesse caso a AVPMSimples começou mais rápida com 1 e 2 *cores*, porém a aceleração tardia tornou a AVPMRandom mair rápida com 8 e 16 *cores*.

Em todos os casos o desempenho da AVPMSimples e da AVPMRandom foi inferior ao do ParGRES original. Se considerarmos apenas 1 *core*, podem haver pontos muito próximos, porém à medida que os *cores* aumentam o desempenho dos algoritmos *multicore* se degradam, permanecendo o ParGRES original superior.

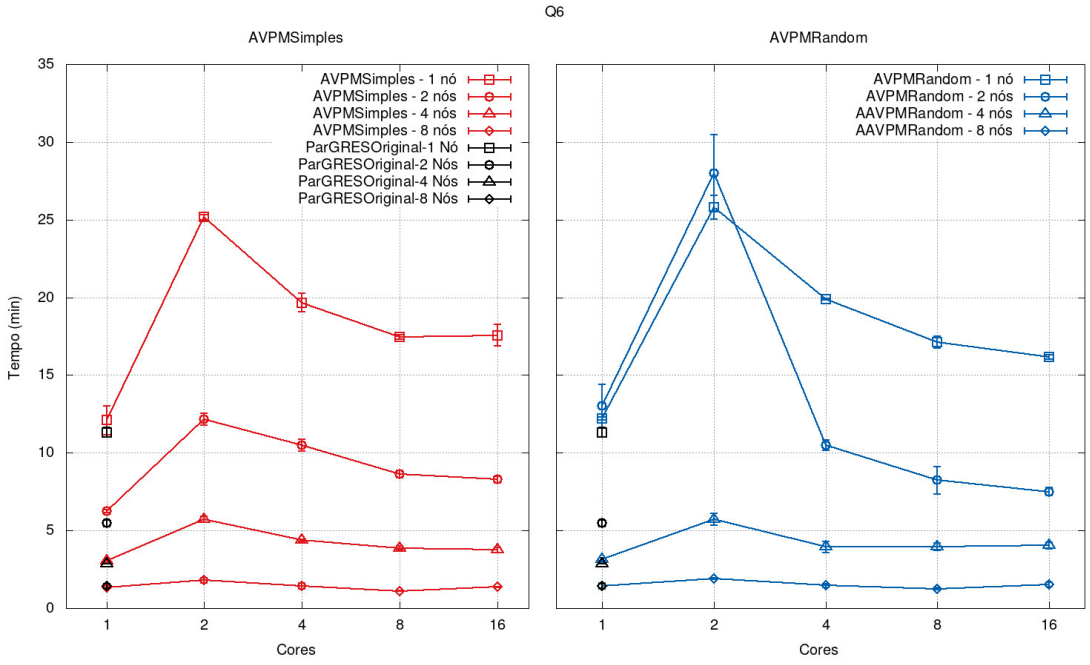

Figura 5.7: Comparação entre o ParGRES original e a aceleração dos algoritmos AVPMSimples e AVPMRandom para a consulta Q6.

Q6 - ParGRES Original	
Nós/Cores	1 Core
1 Nó	$11,32 \pm 0,32$
2 Nós	$5,47 \pm 0,22$
4 Nós	$2,85 \pm 0,01$
8 Nós	$1,44 \pm 0,15$

Tabela 5.7: Tempos de execução ± desvio padrão (em minutos) da consulta Q6 usando o ParGRES original, variando-se os nós com a quantidade fixa de 1 *core*.

Q6 - AVPMSimples					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$12,09 \pm 0,95$	$25,16 \pm 0,15$	$19,66 \pm 0,6$	$17,46 \pm 0,14$	$17,57 \pm 0,7$
2 Nós	$6,24 \pm 0,3$	$12,18 \pm 0,67$	$10,49 \pm 0,42$	$8,64 \pm 0,18$	$8,29 \pm 0,69$
4 Nós	$3,05 \pm 0,1$	$5,72 \pm 0,2$	$4,37 \pm 0,06$	$3,87 \pm 0,12$	$3,75 \pm 0,18$
8 Nós	$1,33 \pm 0,08$	$1,79 \pm 0,16$	$1,41 \pm 0,19$	$1,12 \pm 0,02$	$1,37 \pm 0,03$

Tabela 5.8: Tempos de execução ± desvio padrão (em minutos) da consulta Q6 usando o algoritmo AVPMSimples, variando-se os nós e *cores*.

Q6 - AVPMRandom					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$12,21 \pm 0,1$	$25,81 \pm 0,76$	$19,89 \pm 0,05$	$17,13 \pm 0,38$	$16,17 \pm 0,14$
2 Nós	$13,0 \pm 1,41$	$28,0 \pm 2,45$	$10,49 \pm 0,34$	$8,24 \pm 0,89$	$7,51 \pm 0,25$
4 Nós	$3,14 \pm 0,11$	$5,71 \pm 0,38$	$3,94 \pm 0,36$	$3,96 \pm 0,25$	$4,03 \pm 0,21$
8 Nós	$1,41 \pm 0,19$	$1,9 \pm 0,04$	$1,46 \pm 0,11$	$1,25 \pm 0,07$	$1,52 \pm 0,13$

Tabela 5.9: Tempos de execução ± desvio padrão (em minutos) da consulta Q6 usando o algoritmo AVPMRandom, variando-se os nós e *cores*.

O que pode-se ver no gráfico 5.8 é que o tamanho da partição logo evolui para pouco mais de 16000, já o tamanho das subpartições oscila e de forma um pouco mais demorada que em outras consultas e evolui até o tamanho da partição. Em seguida, a AVPMRandom continua de forma dinâmica e randômica buscando novos tamanho das subpartições ótimos, o que produz as quedas no gráfico, porém na maioria dos casos ele retorna muito rapidamente para o tamanho da partição. Isso evidencia que a AVPMRandom tende a se comportar como a AVPMSimples e o tamanho das subpartições encontrado como o melhor tende a ser o tamanho da partição.

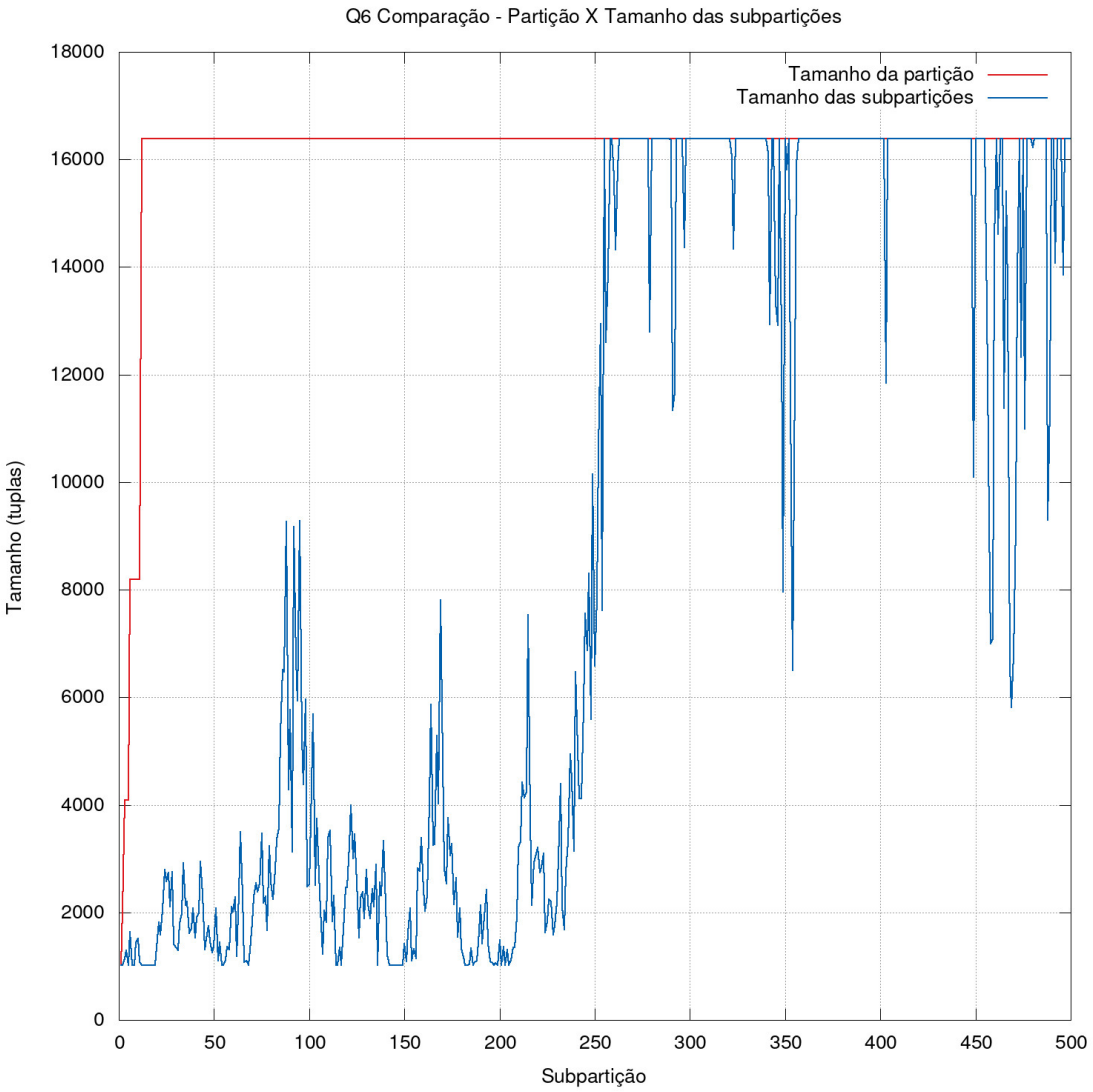

Figura 5.8: Comparação entre o tamanho das subpartições da AVPMSimples (igual ao tamanho da partição) com o da AVPMRandom, evoluindo durante o processamento da consulta Q6.

Abaixo estão os planos de consulta simplificados com análise dos tempos e custos de cada etapa, os planos completos podem ser vistos no apêndice B.3. Os planos são da consulta simulada sobre uma partição e selecionando somente um quarto da partição, respectivamente:

```
---------------------------------------------------------------
Aggregate
   ->  Index Scan
Planning Time: 18.164 ms
Execution Time: 4.610 ms
(7 rows)
---------------------------------------------------------------
```

```
------------------------------------------------------------
Aggregate
   ->  Index Scan
Planning Time: 18.897 ms
Execution Time: 1.250 ms
(7 rows)
------------------------------------------------------------
```

Analisando os planos, observa-se que os tempos de planejamento apresentaram uma pequena desaceleração, e apesar dos tempos de execução apresentarem uma aceleração mais significativa, o tempo de planejamento é uma ordem de grandeza maior e domina o tempo de execução.

Ao final, a consulta sobre partição com o tempo de planejamento leva 22 ms e a consulta sobre a subpartição leva 20 ms, apesar de haver uma discreta aceleração, não é o que se esperava devido à redução de 4 vezes do tamanho da partição para constituir a subpartição. Isso explica o por que de não ter havido aceleração significativa para essa consulta, como mostrado no gráfico de aceleração(5.7) da consulta Q6.

A consulta Q6 é possivelmente a mais simples entre as selecionadas, ela possui apenas uma agregação e consulta apenas uma tabela, a *lineitem*, sem qualquer ordenação ou agrupamento. Analisando os planos de consulta, observa-se que houve aceleração em todas as etapas, porém a etapa de varredura indexada apresentou uma aceleração discreta, no geral a aceleração da fase de execução foi menor que em outras consultas. O único fator constatado que explica o fato de essa consulta não ter acelerado nos testes de aceleração foi o tempo de planejamento que não acelerou com a seletividade da subpartição, tempo esse que domina o tempo de execução.

5.5.4 Consulta Q12

A figura 5.9 e as tabelas 5.10, 5.11 e 5.12 possuem os tempos de execução da consulta Q12 usando diferentes configurações de nós e *cores*, para os algoritmos AVPMSimples e AVPMRandom, além do ParGRES original em apenas um nó.

No gráfico fica bastante visível o comportamento de desacelerar na passagem de 1 *core* para 2, isso ocorre em todos os casos, e depois ocorre uma aceleração. O fato é que essa aceleração tardia só ocorre em relação ao dado obtido com 2 *cores*, sendo a aceleração total inexistente, tanto para o algoritmo AVPMSimples quanto o AVPMRandom. A única ocorrência de aceleração, para 2 nós do AVPMRandom, pode representar um dado privilegiado pelas condições do hardware e sistema operacional no momento da execução, não se pode concluir que o algoritmo acelera em quaisquer condições.

Além disso os dados para uma pequena quantidade de nós apresentam maior variabilidade, o que pode ser observado tanto pela variação dos valores para cima e para baixo quanto pelos desvios padrão, que em particular no algoritmo AVPM-Random foram relevantes.

Comparando-se os resultados da AVPMSimples com a AVPMRandom, pode-se ver que ambos possuem desempenho muito parecido, com números muito próximos hora com a AVPMSimples mais rápida hora com a AVPMRandom mais rápida. A exceção é com 2 nós, nesse caso a AVPMSimples começou mais rápida com 1 e 2 *cores*, porém a aceleração tardia tornou a AVPMRandom mair rápida com 8 e 16 *cores*.

Em todos os casos o desempenho da AVPMSimples e da AVPMRandom foi inferior ao do ParGRES original. Se considerarmos apenas 1 *core*, podem haver pontos muito próximos, porém a medida que os *cores* aumentam o desempenho dos algoritmos *multicore* se degradam, prevalecendo o ParGRES original superior.

O comportamento qualitativo de ambos os algoritmos na Q12 foram muito semelhantes aos da Q6.

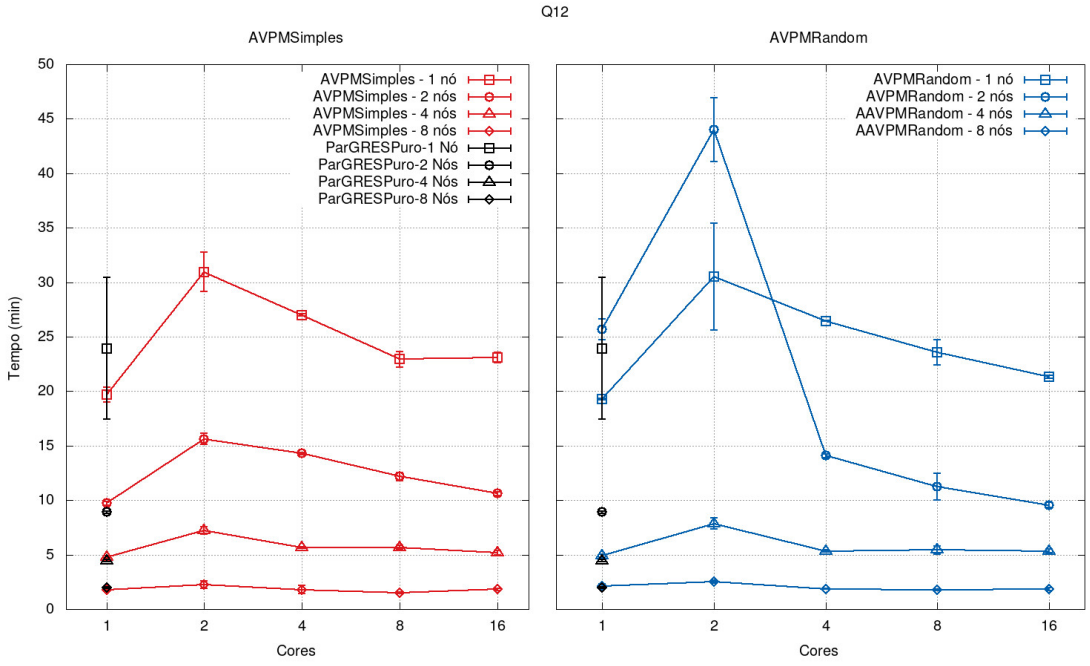

Figura 5.9: Comparação entre o ParGRES original e a aceleração dos algoritmos AVPMSimples e AVPMRandom para a consulta Q12.

Q12 - ParGRES Original	
Nós/Cores	1 Core
1 Nó	$23,94 \pm 6,51$
2 Nós	$8,9 \pm 0,09$
4 Nós	$4,44 \pm 0,01$
8 Nós	$1,97 \pm 0,09$

Tabela 5.10: Tempos de execução ± desvio padrão (em minutos) da consulta Q12 usando o ParGRES original, variando-se os nós com a quantidade fixa de 1 *core*.

Q12 - AVPMSimples					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$19,67 \pm 0,67$	$30,95 \pm 1,81$	$27,0 \pm 0,1$	$22,94 \pm 0,73$	$23,09 \pm 0,49$
2 Nós	$9,71 \pm 0,06$	$15,63 \pm 0,8$	$14,28 \pm 0,83$	$12,18 \pm 0,44$	$10,61 \pm 0,99$
4 Nós	$4,76 \pm 0,15$	$7,22 \pm 0,36$	$5,65 \pm 0,07$	$5,66 \pm 0,17$	$5,18 \pm 0,2$
8 Nós	$1,8 \pm 0,03$	$2,25 \pm 0,34$	$1,8 \pm 0,37$	$1,5 \pm 0,05$	$1,81 \pm 0,1$

Tabela 5.11: Tempos de execução ± desvio padrão (em minutos) da consulta Q12 usando o algoritmo AVPMSimples, variando-se os nós e *cores*.

Q12 - AVPMRandom					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$19,26 \pm 0,07$	$30,51 \pm 4,9$	$26,44 \pm 0,04$	$23,58 \pm 1,18$	$21,33 \pm 0,14$
2 Nós	$25,67 \pm 0,94$	$44,0 \pm 2,94$	$14,08 \pm 0,15$	$11,23 \pm 1,22$	$9,55 \pm 0,23$
4 Nós	$4,9 \pm 0,14$	$7,86 \pm 0,52$	$5,34 \pm 0,1$	$5,43 \pm 0,37$	$5,33 \pm 0,17$
8 Nós	$2,13 \pm 0,18$	$2,5 \pm 0,13$	$1,86 \pm 0,04$	$1,75 \pm 0,08$	$1,81 \pm 0,14$

Tabela 5.12: Tempos de execução ± desvio padrão (em minutos) da consulta Q12 usando o algoritmo AVPMRandom, variando-se os nós e *cores*.

O que pode-se ver no gráfico 5.10 é que o tamanho da partição logo evolui para pouco mais de 16000, já o tamanho das subpartições evolui de forma sinuosa, devido à sua natureza randômica, até o tamanho da partição. Em seguida, a AVPMRandom continua de forma dinâmica e randômica buscando novos tamanho das subpartições ótimos, o que produz as quedas no gráfico, porém na maioria dos casos ele retorna muito rapidamente para o tamanho da partição. Isso evidencia que a AVPMRandom tende a se comportar como a AVPMSimples e o tamanho das subpartições encontrado como o melhor tende a ser o tamanho da partição.

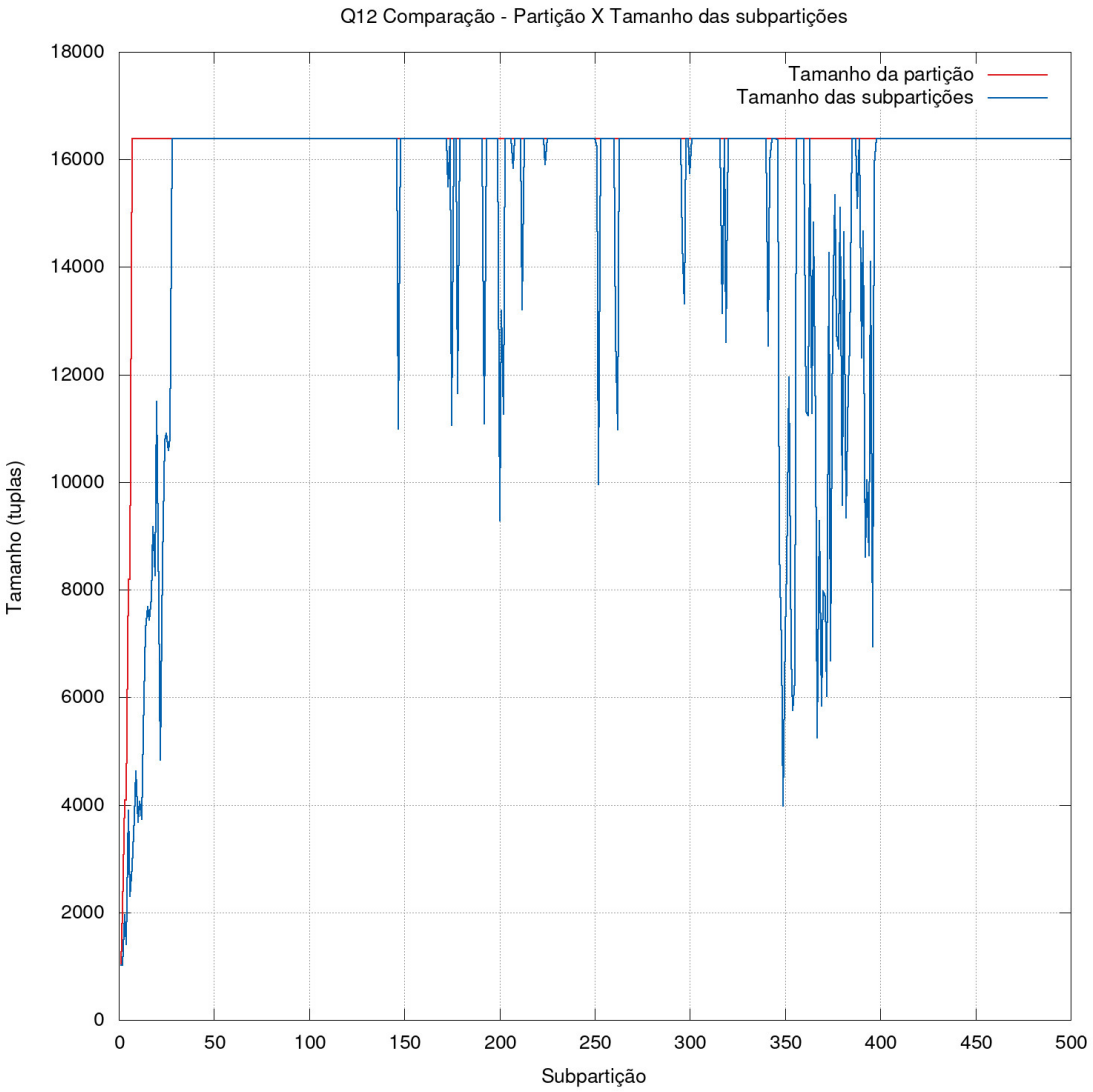

Figura 5.10: Comparação entre o tamanho das subpartições da AVPMSimples (igual ao tamanho da partição) com o da AVPMRandom, evoluindo durante o processamento da consulta Q12.

Abaixo estão os planos de consulta simplificados com análise dos tempos e custos de cada etapa, os planos completos podem ser vistos no apêndice B.4. Os planos são da consulta simulada sobre uma partição e selecionando somente um quarto da partição, respectivamente:

```
-----------------------------------------------------------------
GroupAggregate
   ->  Sort
         ->  Nested Loop
               ->  Index Scan
               ->  Index Scan
Planning Time: 22.834 ms
```

```
Execution Time: 11.928 ms
(14 rows)
-----------------------------------------------------------------

-----------------------------------------------------------------
GroupAggregate
   ->  Sort
         ->  Nested Loop
               ->  Index Scan
               ->  Index Scan
Planning Time: 23.247 ms
Execution Time: 1.574 ms
(14 rows)
-----------------------------------------------------------------
```

Analisando os planos, observa-se que os tempos de planejamento apresentaram uma pequena desaceleração, e apesar dos tempos de execução apresentarem uma aceleração mais significativa, o tempo de planejamento é significativamente maior e domina o tempo de execução.

Ao final, a consulta sobre partição com o tempo de planejamento leva 34 ms e a consulta sobre a subpartição leva 24 ms, apesar de haver uma pequena aceleração, não é o que se esperava devido à redução de 4 vezes do tamanho da partição para constituir a subpartição. Isso explica o por que de não ter havido aceleração significativa para essa consulta, como mostrado no gráfico de aceleração(5.9) da consulta Q12.

A consulta Q12 possui relativa complexidade pois acessa duas tabelas fatos, *orders* e *lineitem*, possui duas agregações, uma ordenação e um agrupamento. Os planos de consulta mostram aceleração em todas as etapas do processamento e houve aceleração significativa do tempo de execução. O único fator constatado que explica o fato de essa consulta não ter acelerado nos testes de aceleração foi o tempo de planejamento que não acelerou com a seletividade da subpartição, tempo esse que domina o tempo de execução.

5.5.5 Consulta Q18

A figura 5.11 e as tabelas 5.13, 5.14 e 5.15 possuem os tempos de execução da consulta Q18 usando diferentes configurações de nós e *cores*, para os algoritmos AVPMSimples e AVPMRandom, além do ParGRES original em apenas um nó.

No gráfico fica bastante visível o comportamento de desacelerar na passagem de 1 *core* para 2, isso ocorre em quase todos os casos, e depois ocorre uma aceleração.

O fato é que essa aceleração tardia só ocorre em relação ao dado obtido com 2 *cores*, sendo a aceleração total inexistente, tanto para o algoritmo AVPMSimples quanto o AVPMRandom. A única exceção é no AVPMRandom com 2 nós, o tempo da consulta com 1 e 2 *cores* são anormalmente altos, o que pode indicar pontos fora da curva, e depois ele acelera. Não se pode concluir que o algoritmo acelera em quaisquer condições.

Além disso os dados para uma pequena quantidade de nós apresentam maior variabilidade, o que pode ser observado tanto pela variação dos valores para cima e para baixo quanto pelos desvios padrão.

Comparando-se os resultados da AVPMSimples com a AVPMRandom, pode-se ver que ambos possuem desempenho muito parecido, com números muito próximos hora com a AVPMSimples mais rápida hora com a AVPMRandom mais rápida. A exceção é com 2 nós, nesse caso a AVPMRandom obteve valores de tempo de execução anormalmente altos para 1 e 2 *cores*.

Em todos os casos o desempenho da AVPMSimples e da AVPMRandom foi inferior ao do ParGRES original. Aqui ocorre o inverso do que ocorreu com as consultas Q6 e Q12, com valores de nós baixos o ParGRES original foi muito superior em desempenho, porém com muitos nós os desempenhos ficaram bem próximos.

O comportamento qualitativo de ambos os algoritmos na Q18 foram muito semelhantes aos da Q6 e da Q12.

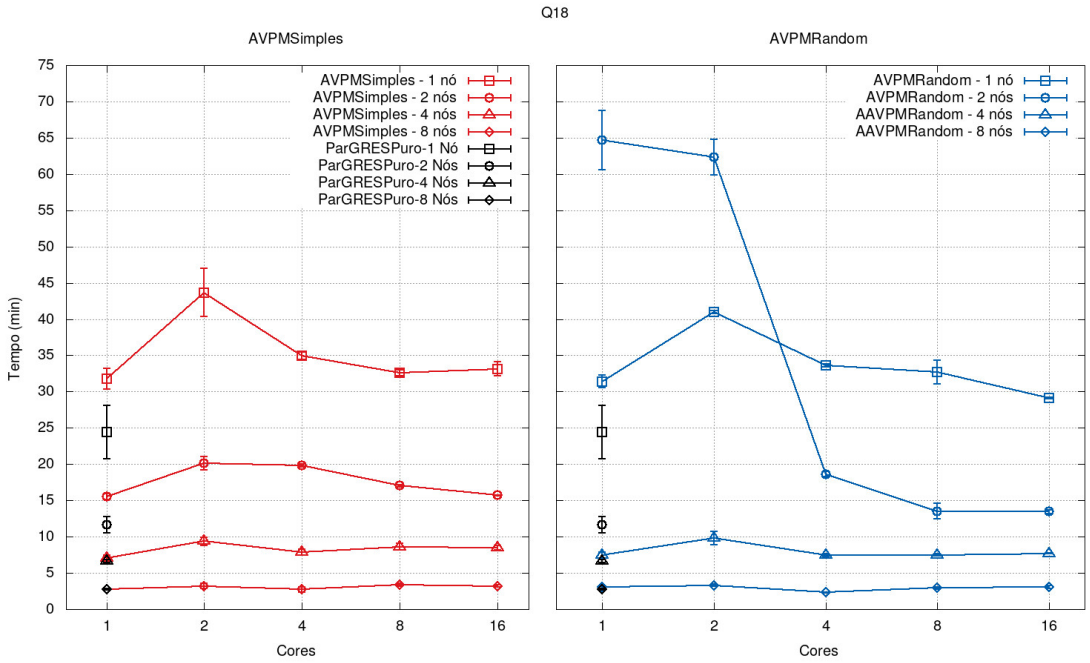

Figura 5.11: Comparação entre o ParGRES original e a aceleração dos algoritmos AVPMSimples e AVPMRandom para a consulta Q18.

Q18 - ParGRES Original	
Nós/Cores	1 Core
1 Nó	$24,45 \pm 3,7$
2 Nós	$11,62 \pm 1,11$
4 Nós	$6,66 \pm 0,2$
8 Nós	$2,76 \pm 0,1$

Tabela 5.13: Tempos de execução ± desvio padrão (em minutos) da consulta Q18 usando o ParGRES original, variando-se os nós com a quantidade fixa de 1 *core*.

Q18 - AVPMSimples					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$31,77 \pm 1,45$	$43,68 \pm 3,31$	$34,94 \pm 0,51$	$32,62 \pm 0,39$	$33,14 \pm 0,94$
2 Nós	$15,52 \pm 0,17$	$20,14 \pm 1,15$	$19,81 \pm 0,81$	$17,02 \pm 0,61$	$15,72 \pm 1,43$
4 Nós	$7,03 \pm 0,39$	$9,35 \pm 0,53$	$7,88 \pm 0,37$	$8,63 \pm 0,42$	$8,46 \pm 0,31$
8 Nós	$2,76 \pm 0,02$	$3,18 \pm 0,37$	$2,72 \pm 0,35$	$3,39 \pm 0,21$	$3,21 \pm 0,16$

Tabela 5.14: Tempos de execução ± desvio padrão (em minutos) da consulta Q18 usando o algoritmo AVPMSimples, variando-se os nós e *cores*.

Q18 - AVPMRandom					
Nós/Cores	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
1 Nó	$31,39 \pm 0,88$	$40,95 \pm 0,23$	$33,62 \pm 0,25$	$32,69 \pm 1,63$	$29,14 \pm 0,08$
2 Nós	$64,67 \pm 4,11$	$62,33 \pm 2,49$	$18,61 \pm 0,35$	$13,5 \pm 1,07$	$13,44 \pm 0,24$
4 Nós	$7,5 \pm 0,41$	$9,83 \pm 0,91$	$7,45 \pm 0,2$	$7,5 \pm 0,19$	$7,69 \pm 0,04$
8 Nós	$3,04 \pm 0,26$	$3,31 \pm 0,12$	$2,33 \pm 0,04$	$2,93 \pm 0,16$	$3,06 \pm 0,24$

Tabela 5.15: Tempos de execução ± desvio padrão (em minutos) da consulta Q18 usando o algoritmo AVPMRandom, variando-se os nós e *cores*.

O que pode-se ver no gráfico 5.12 é que o tamanho da partição logo evolui para pouco mais de 16000, já o tamanho das subpartições evolui de forma sinuosa, devido à sua natureza randômica, até o tamanho da partição. Em seguida, a AVPMRandom continua de forma dinâmica e randômica buscando novos tamanho das subpartições ótimos, o que produz as quedas no gráfico, porém na maioria dos casos ele retorna muito rapidamente para o tamanho da partição. Isso evidencia que a AVPMRandom tende a se comportar como a AVPMSimples e o tamanho das subpartições encontrado como o melhor tende a ser o tamanho da partição.

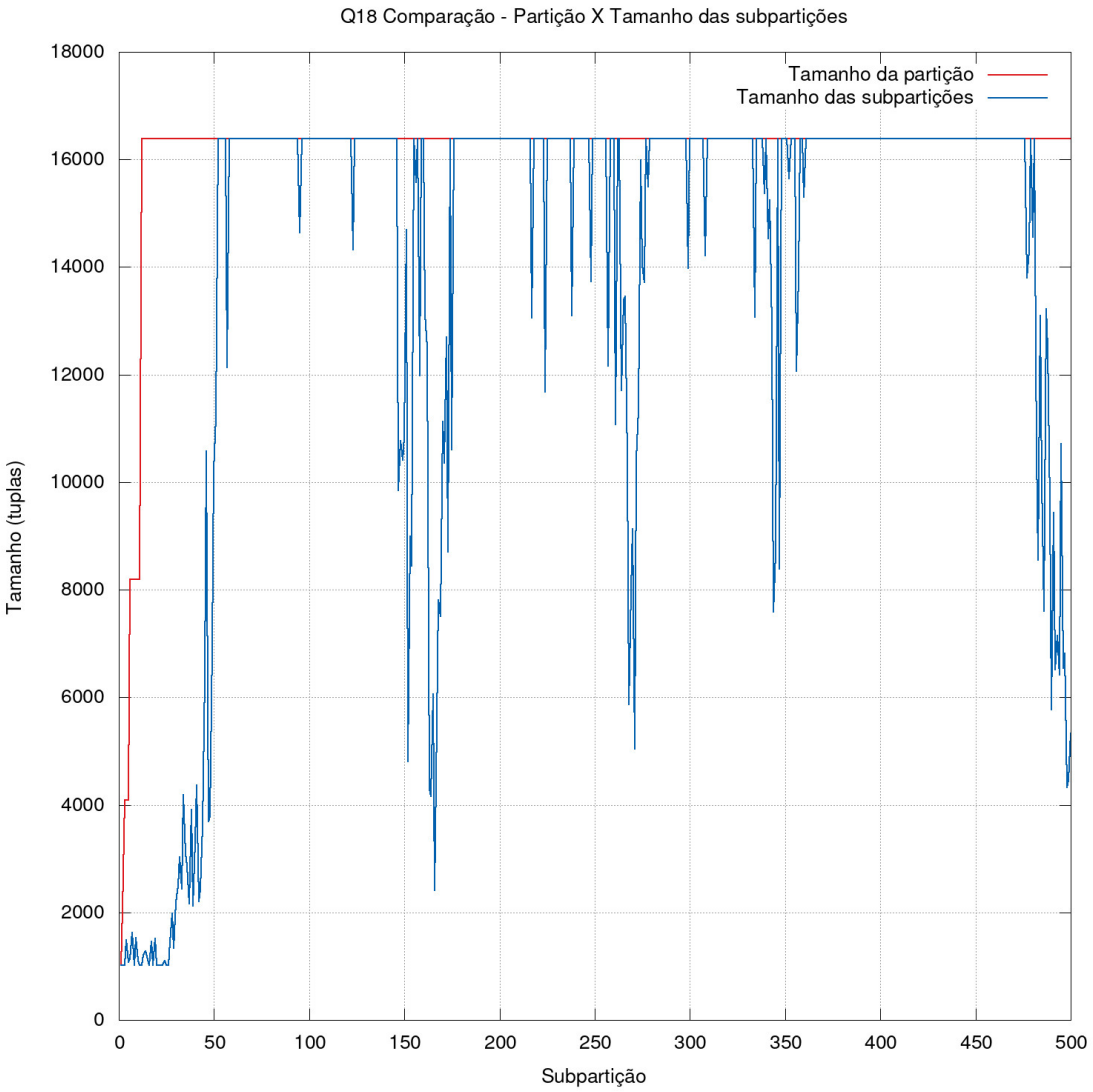

Figura 5.12: Comparação entre o tamanho das subpartições da AVPMSimples (igual ao tamanho da partição) com o da AVPMRandom, evoluindo durante o processamento da consulta Q18.

Abaixo estão os planos de consulta simplificados com análise dos tempos e custos de cada etapa, os planos completos podem ser vistos no apêndice B.5. Os planos são da consulta simulada sobre uma partição e selecionando somente um quarto da partição, respectivamente:

```
-------------------------------------------------------------
GroupAggregate
   ->  Sort
         ->  Nested Loop
               ->  Nested Loop
                     ->  Merge Join
                           ->  Index Scan
```

```
                              ->  GroupAggregate
                                    ->  Index Scan
                        ->  Index Scan
                ->  Index Scan
Planning Time: 115.948 ms
Execution Time: 42.089 ms
(23 rows)
-------------------------------------------------------------

-------------------------------------------------------------
GroupAggregate
   ->  Sort
         ->  Nested Loop
               ->  Nested Loop
                     ->  Merge Join
                           ->  Index Scan
                           ->  GroupAggregate
                                 ->  Index Scan
                     ->  Index Scan
               ->  Index Scan
Planning Time: 23.957 ms
Execution Time: 1.944 ms
(23 rows)
-------------------------------------------------------------
```

Analisando os planos, observa-se que em ambas as consulta, em partição e subpartição, houve aceleração significativa dos tempos de planejamento e execução.

Ao final, a consulta em partição leva 158 ms e a consulta sobre a subpartição leva 25 ms, ocorre uma aceleração significativa, porém os resultados experimentais expressos no gráfico de aceleração (5.5) da consulta Q4 mostram uma aeleração discreta ou nula na maior parde dos casos.

O desempenho ruim na consulta Q18 é atribuído a outros fatores que comprometem o desempenho como o alto custo na composição de resultados, já que a Q18 possui duas tabelas fatos, e a própria complexidade da consulta.

A Q18 é a consulta mais complexa dentre as escolhidas, ela consulta além de duas tabelas fatos, *orders* e *lineitem* também a tabela *customers*. Além disso possui uma subconsulta na tabela *lineitem*, uma agregação, ordenação e agrupamento (o fator de limitação *"LIMIT 100"* foi retirado da consulta neste trabalho). Avaliando os planos de consulta, verificou-se aceleração significativa tanto nos tempos de execução quanto de planejamento, e o único fator constatado que faz com que essa consulta

não tenha acelerado nos testes de aceleração são fatores relacionados à complexidade da consulta como alto custo de composição de resultados.

Capítulo 6

Conclusões

Foram propostos dois algoritmos de execução *multicore* de consultas para complementar a AVP. A abordagem de ambos os algoritmos é bem específica e fortemente inspirada na AVP e seus conceitos já preexistentes de partição virtual, que foi adaptado para subpartição virtual, sua composição de resultados e até mesmo seu espírito adaptativo, de onde se originou o segundo algoritmo AVPMRandom. Em contraposição estão outras abordagens e algoritmos como os baseados em operadores, os de paralelismo intra-operador, os que tentam paralelizar a árvore de execução e outros mais.

O primeiro algoritmo proposto é ingênuo, pois é simples e imediato. Havia a dúvida se o tamanho das subpartições deste algoritmo, a AVPMSimples, que é igual ao tamanho da partição, não seria distante do tamanho que minimizasse o tempo de consulta. Para tentar corrigir essa possível ineficiência foi proposta a AVPMRandom, que é adaptativo e possivelmente poderia encontrar tamanhos das subpartições diferentes e mais próximos do ótimo.

Os resultados mostram no entanto que a AVPMSimples é quase sempre melhor que a AVPMRandom, quando não é melhor possui desempenho próximo. A primeira coisa que se observa são os tempos de execução, onde a AVPMSimples se destaca por uma diferença pequena nas consultas Q1 e Q4, já nas consultas Q6, Q12 e Q18 os resultados são alternados, mas quase sempre muito próximos.

Foram analisados os tamanhos da subpartição da AVPMRandom para entender melhor o seu comportamento, e a conclusão é de que o tamanho das subpartições tende para o tamanho da partição, que é o mesmo tamanho das subpartições da AVPMSimples. Isso significa que a AVPMRandom tende a se comportar como a AVPMSimples, mas não sem alguma perda de desempenho, pois devido ao seu aspecto randômico adaptativo ele busca com frequência soluções possivelmente não boas de tamanhos de subpartição.

Sobre os tempos de consultas, ficou evidente em boa parte das execuções um comportamento de desacelerar entre os número de *cores* 1 e 2, isso é atribuído

a fatores causadores de perda de desempenho citados como estruturas e fatores de programação, composição de resultados e condições de contorno do algoritmo AVPMRandom. Após isso, na maioria dos casos, ocorre uma aceleração local, não necessariamente o tempo de execução para muitos *cores* ficando abaixo do tempo para 1 *core*.

Ficou evidente nos dados obtidos que o comportamento com poucos nós tende a ser mais aleatório e com maiores variações pois o gráfico apresenta sinuosidades e as barras com o desvio padrão são maiores. Já o comportamento com muitos nós é mais estável, porém a aceleração fica visivelmente menor ou até não ocorre. Esse comportamento é atribuído ao tamanho da subpartição em relação à memória. Lembrando que a base utilizada possui 301 GBytes e a memória principal do Lobo Carneiro possui 64 GBytes, logo para uma quantidade elevada de nós cada partição cabe inteira, ou quase inteira na memória.

Os sucesso dos algoritmos apresentados partem de um pressuposto, suponha uma consulta com seletividade N que leva tempo T para ser processada pelo SGBD. Para que os algoritmos propostos sejam efetivos, é necessário que ao diminuir o predicado para uma seletividade $N/4$, por exemplo, o tempo de consulta também diminuía, idealmente para $T/4$. Mas isso não foi verificado, foi simulado o processo de criação de subpartições e os planos de consultas gerados pelo SGBD analisados. A conclusão é que devemos considerar o tempo de execução e de planejamento como tempos constituintes da consulta, enquanto o tempo de processamento acelerou em todos os casos, o tempo de planejamento pode ou não acelerar com a restrição do predicado e redução da consulta. Observa-se ainda que para consultas muito pequenas, o tempo de planejamento tende a dominar o tempo de execução, e se mostra determinante no tempo total.

Nas consultas Q1 e Q4 houve aceleração tanto dos tempos de planejamento quanto de execução, de modo que a consulta como um todo acelerou. Nas consultas Q6 e Q12 houve aceleração significativa apenas nos tempos de execução, porém os tempos de planejamento dominaram o tempo de consulta, de forma que as consultas como um todo não aceleraram significativamente. A Q18 apresentou um resultado atípico, aceleração nos tempos de planejamento e execução porém a consulta não acelerou nos testes de aceleração, isso é atribuído aos fatores que causam perda de desempenho como a composição de resultados e a complexidade da consulta.

Comparando os resultados de aceleração com as complexidades das consultas, conclui-se que a complexidade não possui correlação direta com a aceleração, pois a consulta Q1 que possuía complexidade média obteve boa aceleração, enquanto a Q6, que era a consulta mais simples, não acelerou. O fator determinante na aceleração das consultas parece ser a capacidade do SGBD acelerar tanto os tempos de execução quanto de planejamento. Nota-se ainda que a Q18, a consulta mais complexa,

acelerou nos tempos de execução e planejamento, apesar de não ter acelerado nos testes de aceleração devido à sua complexidade na composição de resultados.

Devido aos fatores citados anteriormente apenas as consultas Q1 e Q4 apresentam aceleração nos testes de aceleração, ainda assim são sublineares. Não existe aceleração em todas as consultas, isso sugere um método adaptativo que identifique quando é vantajoso usar a implementação *multicore* e a use, e quando não for vantajoso pode-se utilizar a AVP.

Na comparação entre os algoritmos propostos e o ParGRES puro, também só foi verificado um desempenho melhor, especialmente para números elevados de *cores*, nas consultas Q1 e Q4. Nas outras consultas o desempenho do ParGRES puro superou os algoritmos *multicore*.

O ParGRES, que de acordo com outros trabalhos já possuía o comportamento de acelerar com o aumento dos nós, não perdeu essa característica devido à implementação dos algoritmos *multicore*. Ainda pode-se observar em quase todos os casos uma clara aceleração com as curvas umas acima das outras ao se variar os nós.

6.1 Trabalhos Futuros

Como trabalhos futuros cita-se o experimento com lotes de consultas, simulando um uso multiusuário com diversos agentes submetendo consultas ao ParGRES de forma concorrente. Estes testes foram feitos no trabalho original que apresentou a AVP e poderiam complementar os resultados obtidos de testes com a AVPMSimples e AVPMRandom.

De acordo com os resultados obtidos, a implementação *multicore* da AVP pode apresentar aceleração ou não dependendo da consulta realizada. Para responder a pergunta de quando e se deve ser usada a versão *multicore* do ParGRES apresentada neste trabalho, uma possível proposta é um algoritmo adaptativo que execute inicialmente apenas uma partição e uma subpartição. Comparando os tempos de execução poderia-se concluir se o processo de criação de subpartições apresenta aceleração significativa ou não, e então decidir sobre o uso do algoritmo multicore. Quando não houver aceleração, mantém-se a AVP sem a implementação *multicore*.

No presente trabalho o algoritmo AVPMRandom busca por um tamanho das subpartições bom sem privilegiar nenhuma direção na busca. Essa característica foi escolhida de forma proposital para produzir um algoritmo que fizesse uma busca que fosse tão abrangente quanto possível. Uma outra possibilidade onde essa escolha de projeto é promissora é em ambientes heterogêneos, pois tal método adaptativo pode encontrar um tamanho das subpartições bom em unidades de processamento com configurações diferentes entre si e desconhecidas. Portanto um trabalho futuro possível seria a implementação e testes do AVPMRandom em ambiente heterogêneos,

podendo ser *clusters* ou outros como a nuvem.

Um dos recursos que o *Java* proporciona através do JDBC é o *Prepared Statement*, que oferece uma forma de pré-compilar a consulta elaborando o plano de consulta uma única vez. Como trabalho futuro poderia ser implementado esse recurso no ParGRES *multicore*, uma vez que foi constatada como principal causa de algumas consultas não terem acelerado o tempo dispendido no planejamento da consulta de cada subpartição. Isso tem potencial de resolver esse problema, e a maioria dos SGBDs atuais têm suporte para esse recurso, apesar de não ser todos.

Referências Bibliográficas

[1] ÖZSU, M. T., VALDURIEZ, P. *Principles of distributed database systems*, v. 3. , Springer, 2011.

[2] GORLA, N. "Features to consider in a data warehousing system", *Communications of the ACM*, v. 46, n. 11, pp. 111–115, 2003.

[3] LIMA, A. D. A. B. *Paralelismo Intra-Consulta em Clusters de Bancos de Dados*. Tese de Doutorado, Tese de Doutorado, Programa de Engenharia de Sistemas e Computação, COPPE ..., 2004.

[4] AKAL, F., BÖHM, K., SCHEK, H.-J. "Olap query evaluation in a database cluster: A performance study on intra-query parallelism". In: *East European Conference on Advances in Databases and Information Systems*, pp. 218–231. Springer, 2002.

[5] DE ASSIS BENTO LIMA, A. "Repositório do ParGRES - Bitbucket". Disponível em: <`https://bitbucket.org/pargres/`>.

[6] HUBER, F., FREYTAG, J. C. "Query Processing on Multi-Core Architectures." In: *Grundlagen von Datenbanken*, pp. 27–31. Citeseer, 2009.

[7] MATTOSO, M., ZIMBRÃO, G., LIMA, A. A., et al. "ParGRES: uma camada de processamento paralelo de consultas sobre o PostgreSQL". In: *WSL2005-6 Workshop Software Livre*, pp. 259–264, 2005.

[8] LIMA, A. A., MATTOSO, M., VALDURIEZ, P. "Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster." In: *SBBD*, v. 4, pp. 92–105, 2004.

[9] LIMA, A. A., MATTOSO, M., VALDURIEZ, P. "OLAP query processing in a database cluster". In: *European Conference on Parallel Processing*, pp. 355–362. Springer, 2004.

[10] LIMA, A. A., FURTADO, C., VALDURIEZ, P., et al. "Parallel OLAP query processing in database clusters with data replication", *Distributed and Parallel Databases*, v. 25, n. 1, pp. 97–123, 2009.

[11] PAES, M., LIMA, A. A., VALDURIEZ, P., et al. "High-performance query processing of a real-world OLAP database with ParGRES". In: *High Performance Computing for Computational Science-VECPAR 2008: 8th International Conference, Toulouse, France, June 24-27, 2008. Revised Selected Papers 8*, pp. 188–200. Springer, 2008.

[12] KOTOWSKI, N., LIMA, A. A., PACITTI, E., et al. "Parallel query processing for OLAP in grids", *Concurrency and Computation: Practice and Experience*, v. 20, n. 17, pp. 2039–2048, 2008.

[13] WM RIBEIRO, M., AB LIMA, A., DE OLIVEIRA, D. "OLAP parallel query processing in clouds with C-ParGRES", *Concurrency and Computation: Practice and Experience*, v. 32, n. 7, pp. e5590, 2020.

[14] GRAEFE, G. "Encapsulation of parallelism in the volcano query processing system", *ACM SIGMOD Record*, v. 19, n. 2, pp. 102–111, 1990.

[15] ACKER, R., ROTH, C., BAYER, R. "Parallel query processing in databases on multicore architectures". In: *Algorithms and Architectures for Parallel Processing: 8th International Conference, ICA3PP 2008, Cyprus, June 9-11, 2008 Proceedings 8*, pp. 2–13. Springer, 2008.

[16] DEEPAK, S., KUMAR, S. U., DURGESH, M., et al. "Query processing and optimization of parallel database system in multi processor environments". In: *2012 Sixth Asia Modelling Symposium*, pp. 191–194. IEEE, 2012.

[17] KRIKELLAS, K., CINTRA, M., VIGLAS, S. "Scheduling threads for intra-query parallelism on multicore processors", *EDBT, University of Edinburgh, Edinburgh, Scotland*, 2010.

[18] VIGLAS, S. D. "A comparative study of implementation techniques for query processing in multicore systems", *IEEE Transactions on Knowledge and Data Engineering*, v. 26, n. 1, pp. 3–15, 2012.

[19] GARCIA, P., KORTH, H. F. "Database hash-join algorithms on multithreaded computer architectures". In: *Proceedings of the 3rd Conference on Computing Frontiers*, pp. 241–252, 2006.

[20] TPC. "TPC-H". . Disponível em: <`https://www.tpc.org/tpch/`>.

[21] TPC. "TPC-H Specs/Source". . Disponível em: <`https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp`>.

[22] KOLEV, B., LEVCHENKO, O., PACITTI, E., et al. "Parallel polyglot query processing on heterogeneous cloud data stores with LeanXcale". In: *2018 IEEE International Conference on Big Data (Big Data)*, pp. 1757–1766. IEEE, 2018.

[23] MÜHLBAUER, T., RÖDIGER, W., REISER, A., et al. "ScyPer: a hybrid OLTP&OLAP distributed main memory database system for scalable real-time analytics", *Datenbanksysteme für Business, Technologie und Web (BTW) 2044*, 2013.

[24] LI, F., ÖZSU, M. T., CHEN, G., et al. "R-store: a scalable distributed system for supporting real-time analytics". In: *2014 IEEE 30th International Conference on Data Engineering*, pp. 40–51. IEEE, 2014.

[25] KEMPER, A., NEUMANN, T. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *2011 IEEE 27th International Conference on Data Engineering*, pp. 195–206. IEEE, 2011.

[26] CROWL, L. A. "How to measure, present, and compare parallel performance", *IEEE Concurrency (out of print)*, v. 2, n. 01, pp. 9–25, 1994.

[27] NACAD. "NACAD". Disponível em: <`http://www.nacad.ufrj.br/pt/informacoes/apresentacao`>.

Apêndice A

Arquivo de Configuração do PostgreSQL

```
# -----------------------------
# PostgreSQL configuration file
# -----------------------------
#
# This file consists of lines of the form:
#
#   name = value
#
# (The "=" is optional.)  Whitespace may be used.  Comments are introduced with
# "#" anywhere on a line.  The complete list of parameter names and allowed
# values can be found in the PostgreSQL documentation.
#
# The commented-out settings shown in this file represent the default values.
# Re-commenting a setting is NOT sufficient to revert it to the default value;
# you need to reload the server.
#
# This file is read on server startup and when the server receives a SIGHUP
# signal.  If you edit the file on a running system, you have to SIGHUP the
# server for the changes to take effect, run "pg_ctl reload", or execute
# "SELECT pg_reload_conf()".  Some parameters, which are marked below,
# require a server shutdown and restart to take effect.
#
# Any parameter can also be given as a command-line option to the server, e.g.,
# "postgres -c log_connections=on".  Some parameters can be changed at run time
# with the "SET" SQL command.
#
```

```
# Memory units:  kB = kilobytes        Time units:  ms  = milliseconds
#                MB = megabytes                     s   = seconds
#                GB = gigabytes                     min = minutes
#                TB = terabytes                     h   = hours
#                                                   d   = days


#------------------------------------------------------------------------------
# FILE LOCATIONS
#------------------------------------------------------------------------------

# The default values of these variables are driven from the -D command-line
# option or PGDATA environment variable, represented here as ConfigDir.

#data_directory = 'ConfigDir'# use data in another directory
# (change requires restart)
#hba_file = 'ConfigDir/pg_hba.conf'# host-based authentication file
# (change requires restart)
#ident_file = 'ConfigDir/pg_ident.conf'# ident configuration file
# (change requires restart)

# If external_pid_file is not explicitly set, no extra PID file is written.
#external_pid_file = ''# write an extra PID file
# (change requires restart)


#------------------------------------------------------------------------------
# CONNECTIONS AND AUTHENTICATION
#------------------------------------------------------------------------------

# - Connection Settings -

#listen_addresses = 'localhost'# what IP address(es) to listen on;
# comma-separated list of addresses;
# defaults to 'localhost'; use '*' for all
# (change requires restart)
#port = 5432 # (change requires restart)
max_connections = 100 # (change requires restart)
#superuser_reserved_connections = 3 # (change requires restart)
```

```
#unix_socket_directories = '/tmp'# comma-separated list of directories
# (change requires restart)
#unix_socket_group = ''# (change requires restart)
#unix_socket_permissions = 0777 # begin with 0 to use octal notation
# (change requires restart)
#bonjour = off # advertise server via Bonjour
# (change requires restart)
#bonjour_name = ''# defaults to the computer name
# (change requires restart)

# - TCP Keepalives -
# see "man 7 tcp" for details

#tcp_keepalives_idle = 0 # TCP_KEEPIDLE, in seconds;
# 0 selects the system default
#tcp_keepalives_interval = 0 # TCP_KEEPINTVL, in seconds;
# 0 selects the system default
#tcp_keepalives_count = 0 # TCP_KEEPCNT;
# 0 selects the system default

# - Authentication -

#authentication_timeout = 1min # 1s-600s
#password_encryption = md5 # md5 or scram-sha-256
#db_user_namespace = off

# GSSAPI using Kerberos
#krb_server_keyfile = ''
#krb_caseins_users = off

# - SSL -

#ssl = off
#ssl_ca_file = ''
#ssl_cert_file = 'server.crt'
#ssl_crl_file = ''
#ssl_key_file = 'server.key'
#ssl_ciphers = 'HIGH:MEDIUM:+3DES:!aNULL' # allowed SSL ciphers
#ssl_prefer_server_ciphers = on
```

```
#ssl_ecdh_curve = 'prime256v1'
#ssl_dh_params_file = ''
#ssl_passphrase_command = ''
#ssl_passphrase_command_supports_reload = off


#------------------------------------------------------------------------------
# RESOURCE USAGE (except WAL)
#------------------------------------------------------------------------------

# - Memory -

shared_buffers = 128MB # min 128kB
# (change requires restart)
#huge_pages = try # on, off, or try
# (change requires restart)
#temp_buffers = 8MB # min 800kB
#max_prepared_transactions = 0 # zero disables the feature
# (change requires restart)
# Caution: it is not advisable to set max_prepared_transactions nonzero unless
# you actively intend to use prepared transactions.
#work_mem = 4MB # min 64kB
#maintenance_work_mem = 64MB # min 1MB
#autovacuum_work_mem = -1 # min 1MB, or -1 to use maintenance_work_mem
#max_stack_depth = 2MB # min 100kB
dynamic_shared_memory_type = posix # the default is the first option
# supported by the operating system:
#   posix
#   sysv
#   windows
#   mmap
# use none to disable dynamic shared memory
# (change requires restart)

# - Disk -

#temp_file_limit = -1 # limits per-process temp file space
# in kB, or -1 for no limit
```

```
# - Kernel Resources -

#max_files_per_process = 1000 # min 25
# (change requires restart)

# - Cost-Based Vacuum Delay -

#vacuum_cost_delay = 0 # 0-100 milliseconds
#vacuum_cost_page_hit = 1 # 0-10000 credits
#vacuum_cost_page_miss = 10 # 0-10000 credits
#vacuum_cost_page_dirty = 20 # 0-10000 credits
#vacuum_cost_limit = 200 # 1-10000 credits

# - Background Writer -

#bgwriter_delay = 200ms # 10-10000ms between rounds
#bgwriter_lru_maxpages = 100 # max buffers written/round, 0 disables
#bgwriter_lru_multiplier = 2.0 # 0-10.0 multiplier on buffers scanned/round
#bgwriter_flush_after = 512kB # measured in pages, 0 disables

# - Asynchronous Behavior -

#effective_io_concurrency = 1 # 1-1000; 0 disables prefetching
#max_worker_processes = 8 # (change requires restart)
#max_parallel_maintenance_workers = 2 # taken from max_parallel_workers
#max_parallel_workers_per_gather = 2 # taken from max_parallel_workers
#parallel_leader_participation = on
#max_parallel_workers = 8 # maximum number of max_worker_processes that
# can be used in parallel operations
#old_snapshot_threshold = -1 # 1min-60d; -1 disables; 0 is immediate
# (change requires restart)
#backend_flush_after = 0 # measured in pages, 0 disables


#------------------------------------------------------------------------------
# WRITE-AHEAD LOG
#------------------------------------------------------------------------------

# - Settings -
```

```
#wal_level = replica # minimal, replica, or logical
# (change requires restart)
#fsync = on # flush data to disk for crash safety
# (turning this off can cause
# unrecoverable data corruption)
#synchronous_commit = on # synchronization level;
# off, local, remote_write, remote_apply, or on
#wal_sync_method = fsync # the default is the first option
# supported by the operating system:
#   open_datasync
#   fdatasync (default on Linux)
#   fsync
#   fsync_writethrough
#   open_sync
#full_page_writes = on # recover from partial page writes
#wal_compression = off # enable compression of full-page writes
#wal_log_hints = off # also do full page writes of non-critical updates
# (change requires restart)
#wal_buffers = -1 # min 32kB, -1 sets based on shared_buffers
# (change requires restart)
#wal_writer_delay = 200ms # 1-10000 milliseconds
#wal_writer_flush_after = 1MB # measured in pages, 0 disables

#commit_delay = 0 # range 0-100000, in microseconds
#commit_siblings = 5 # range 1-1000

# - Checkpoints -

#checkpoint_timeout = 5min # range 30s-1d
max_wal_size = 1GB
min_wal_size = 80MB
#checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0
#checkpoint_flush_after = 256kB # measured in pages, 0 disables
#checkpoint_warning = 30s # 0 disables

# - Archiving -

#archive_mode = off # enables archiving; off, on, or always
```

```
# (change requires restart)
#archive_command = ''# command to use to archive a logfile segment
# placeholders: %p = path of file to archive
#               %f = file name only
# e.g. 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'
#archive_timeout = 0 # force a logfile segment switch after this
# number of seconds; 0 disables


#------------------------------------------------------------------------------
# REPLICATION
#------------------------------------------------------------------------------

# - Sending Servers -

# Set these on the master and on any standby that will send replication data.

#max_wal_senders = 10 # max number of walsender processes
# (change requires restart)
#wal_keep_segments = 0 # in logfile segments; 0 disables
#wal_sender_timeout = 60s # in milliseconds; 0 disables

#max_replication_slots = 10 # max number of replication slots
# (change requires restart)
#track_commit_timestamp = off # collect timestamp of transaction commit
# (change requires restart)

# - Master Server -

# These settings are ignored on a standby server.

#synchronous_standby_names = ''# standby servers that provide sync rep
# method to choose sync standbys, number of sync standbys,
# and comma-separated list of application_name
# from standby(s); '*' = all
#vacuum_defer_cleanup_age = 0 # number of xacts by which cleanup is delayed

# - Standby Servers -
```

```
# These settings are ignored on a master server.

#hot_standby = on # "off" disallows queries during recovery
# (change requires restart)
#max_standby_archive_delay = 30s # max delay before canceling queries
# when reading WAL from archive;
# -1 allows indefinite delay
#max_standby_streaming_delay = 30s # max delay before canceling queries
# when reading streaming WAL;
# -1 allows indefinite delay
#wal_receiver_status_interval = 10s # send replies at least this often
# 0 disables
#hot_standby_feedback = off # send info from standby to prevent
# query conflicts
#wal_receiver_timeout = 60s # time that receiver waits for
# communication from master
# in milliseconds; 0 disables
#wal_retrieve_retry_interval = 5s # time to wait before retrying to
# retrieve WAL after a failed attempt

# - Subscribers -

# These settings are ignored on a publisher.

#max_logical_replication_workers = 4 # taken from max_worker_processes
# (change requires restart)
#max_sync_workers_per_subscription = 2 # taken from max_logical_replication_
                                       #workers


#------------------------------------------------------------------------------
# QUERY TUNING
#------------------------------------------------------------------------------

# - Planner Method Configuration -

#enable_bitmapscan = on
#enable_hashagg = on
#enable_hashjoin = on
```

```
#enable_indexscan = on
#enable_indexonlyscan = on
#enable_material = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_parallel_append = on
#enable_seqscan = on
#enable_sort = on
#enable_tidscan = on
#enable_partitionwise_join = off
#enable_partitionwise_aggregate = off
#enable_parallel_hash = on
#enable_partition_pruning = on

# - Planner Cost Constants -

#seq_page_cost = 1.0 # measured on an arbitrary scale
#random_page_cost = 4.0 # same scale as above
#cpu_tuple_cost = 0.01 # same scale as above
#cpu_index_tuple_cost = 0.005 # same scale as above
#cpu_operator_cost = 0.0025 # same scale as above
#parallel_tuple_cost = 0.1 # same scale as above
#parallel_setup_cost = 1000.0 # same scale as above

#jit_above_cost = 100000 # perform JIT compilation if available
# and query more expensive than this;
# -1 disables
#jit_inline_above_cost = 500000 # inline small functions if query is
# more expensive than this; -1 disables
#jit_optimize_above_cost = 500000 # use expensive JIT optimizations if
# query is more expensive than this;
# -1 disables

#min_parallel_table_scan_size = 8MB
#min_parallel_index_scan_size = 512kB
#effective_cache_size = 4GB

# - Genetic Query Optimizer -
```

```
#geqo = on
#geqo_threshold = 12
#geqo_effort = 5 # range 1-10
#geqo_pool_size = 0 # selects default based on effort
#geqo_generations = 0 # selects default based on effort
#geqo_selection_bias = 2.0 # range 1.5-2.0
#geqo_seed = 0.0 # range 0.0-1.0

# - Other Planner Options -

#default_statistics_target = 100 # range 1-10000
#constraint_exclusion = partition # on, off, or partition
#cursor_tuple_fraction = 0.1 # range 0.0-1.0
#from_collapse_limit = 8
#join_collapse_limit = 8 # 1 disables collapsing of explicit
# JOIN clauses
#force_parallel_mode = off
#jit = off # allow JIT compilation


#------------------------------------------------------------------------------
# REPORTING AND LOGGING
#------------------------------------------------------------------------------

# - Where to Log -

#log_destination = 'stderr'# Valid values are combinations of
# stderr, csvlog, syslog, and eventlog,
# depending on platform.  csvlog
# requires logging_collector to be on.

# This is used when logging to stderr:
#logging_collector = off # Enable capturing of stderr and csvlog
# into log files. Required to be on for
# csvlogs.
# (change requires restart)

# These are only used if logging_collector is on:
#log_directory = 'log'# directory where log files are written,
```

```
# can be absolute or relative to PGDATA
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'# log file name pattern,
# can include strftime() escapes
#log_file_mode = 0600 # creation mode for log files,
# begin with 0 to use octal notation
#log_truncate_on_rotation = off # If on, an existing log file with the
# same name as the new log file will be
# truncated rather than appended to.
# But such truncation only occurs on
# time-driven rotation, not on restarts
# or size-driven rotation.  Default is
# off, meaning append to existing files
# in all cases.
#log_rotation_age = 1d # Automatic rotation of logfiles will
# happen after that time.  0 disables.
#log_rotation_size = 10MB # Automatic rotation of logfiles will
# happen after that much log output.
# 0 disables.

# These are relevant when logging to syslog:
#syslog_facility = 'LOCAL0'
#syslog_ident = 'postgres'
#syslog_sequence_numbers = on
#syslog_split_messages = on

# This is only relevant when logging to eventlog (win32):
# (change requires restart)
#event_source = 'PostgreSQL'

# - When to Log -

#log_min_messages = warning # values in order of decreasing detail:
#   debug5
#   debug4
#   debug3
#   debug2
#   debug1
#   info
#   notice
```

```
#   warning
#   error
#   log
#   fatal
#   panic

#log_min_error_statement = error # values in order of decreasing detail:
#   debug5
#   debug4
#   debug3
#   debug2
#   debug1
#   info
#   notice
#   warning
#   error
#   log
#   fatal
#   panic (effectively off)

#log_min_duration_statement = -1 # -1 is disabled, 0 logs all statements
# and their durations, > 0 logs only
# statements running at least this number
# of milliseconds


# - What to Log -

#debug_print_parse = off
#debug_print_rewritten = off
#debug_print_plan = off
#debug_pretty_print = on
#log_checkpoints = off
#log_connections = off
#log_disconnections = off
#log_duration = off
#log_error_verbosity = default # terse, default, or verbose messages
#log_hostname = off
#log_line_prefix = '%m [%p] '# special values:
```

```
#   %a = application name
#   %u = user name
#   %d = database name
#   %r = remote host and port
#   %h = remote host
#   %p = process ID
#   %t = timestamp without milliseconds
#   %m = timestamp with milliseconds
#   %n = timestamp with milliseconds (as a Unix epoch)
#   %i = command tag
#   %e = SQL state
#   %c = session ID
#   %l = session line number
#   %s = session start timestamp
#   %v = virtual transaction ID
#   %x = transaction ID (0 if none)
#   %q = stop here in non-session
#        processes
#   %% = '%'
# e.g. '<%u%%%d> '
#log_lock_waits = off # log lock waits >= deadlock_timeout
#log_statement = 'none'# none, ddl, mod, all
#log_replication_commands = off
#log_temp_files = -1 # log temporary files equal or larger
# than the specified size in kilobytes;
# -1 disables, 0 logs all temp files
log_timezone = 'Brazil/East'

#------------------------------------------------------------------------------
# PROCESS TITLE
#------------------------------------------------------------------------------

#cluster_name = ''# added to process titles if nonempty
# (change requires restart)
#update_process_title = on


#------------------------------------------------------------------------------
# STATISTICS
```

```
#------------------------------------------------------------------------------

# - Query and Index Statistics Collector -

#track_activities = on
#track_counts = on
#track_io_timing = off
#track_functions = none # none, pl, all
#track_activity_query_size = 1024 # (change requires restart)
#stats_temp_directory = 'pg_stat_tmp'


# - Monitoring -

#log_parser_stats = off
#log_planner_stats = off
#log_executor_stats = off
#log_statement_stats = off


#------------------------------------------------------------------------------
# AUTOVACUUM
#------------------------------------------------------------------------------

#autovacuum = on # Enable autovacuum subprocess?  'on'
# requires track_counts to also be on.
#log_autovacuum_min_duration = -1 # -1 disables, 0 logs all actions and
# their durations, > 0 logs only
# actions running at least this number
# of milliseconds.
#autovacuum_max_workers = 3 # max number of autovacuum subprocesses
# (change requires restart)
#autovacuum_naptime = 1min # time between autovacuum runs
#autovacuum_vacuum_threshold = 50 # min number of row updates before
# vacuum
#autovacuum_analyze_threshold = 50 # min number of row updates before
# analyze
#autovacuum_vacuum_scale_factor = 0.2 # fraction of table size before vacuum
#autovacuum_analyze_scale_factor = 0.1 # fraction of table size before analyze
```

```
#autovacuum_freeze_max_age = 200000000 # maximum XID age before forced vacuum
# (change requires restart)
#autovacuum_multixact_freeze_max_age = 400000000
# maximum multixact age
# before forced vacuum
# (change requires restart)
#autovacuum_vacuum_cost_delay = 20ms # default vacuum cost delay for
# autovacuum, in milliseconds;
# -1 means use vacuum_cost_delay
#autovacuum_vacuum_cost_limit = -1 # default vacuum cost limit for
# autovacuum, -1 means use
# vacuum_cost_limit


#------------------------------------------------------------------------------
# CLIENT CONNECTION DEFAULTS
#------------------------------------------------------------------------------

# - Statement Behavior -

#client_min_messages = notice # values in order of decreasing detail:
#   debug5
#   debug4
#   debug3
#   debug2
#   debug1
#   log
#   notice
#   warning
#   error
#search_path = '"$user", public'# schema names
#row_security = on
#default_tablespace = ''# a tablespace name, '' uses the default
#temp_tablespaces = ''# a list of tablespace names, '' uses
# only default tablespace
#check_function_bodies = on
#default_transaction_isolation = 'read committed'
#default_transaction_read_only = off
#default_transaction_deferrable = off
```

```
#session_replication_role = 'origin'
#statement_timeout = 0 # in milliseconds, 0 is disabled
#lock_timeout = 0 # in milliseconds, 0 is disabled
#idle_in_transaction_session_timeout = 0 # in milliseconds, 0 is disabled
#vacuum_freeze_min_age = 50000000
#vacuum_freeze_table_age = 150000000
#vacuum_multixact_freeze_min_age = 5000000
#vacuum_multixact_freeze_table_age = 150000000
#vacuum_cleanup_index_scale_factor = 0.1 # fraction of total number of tuples
# before index cleanup, 0 always performs
# index cleanup
#bytea_output = 'hex'# hex, escape
#xmlbinary = 'base64'
#xmloption = 'content'
#gin_fuzzy_search_limit = 0
#gin_pending_list_limit = 4MB

# - Locale and Formatting -

datestyle = 'iso, mdy'
#intervalstyle = 'postgres'
timezone = 'Brazil/East'
#timezone_abbreviations = 'Default'     # Select the set of available time zone
# abbreviations.  Currently, there are
#   Default
#   Australia (historical usage)
#   India
# You can create your own file in
# share/timezonesets/.
#extra_float_digits = 0 # min -15, max 3
#client_encoding = sql_ascii # actually, defaults to database
# encoding

# These settings are initialized by initdb, but they can be changed.
lc_messages = 'en_US.UTF-8'# locale for system error message
# strings
lc_monetary = 'en_US.UTF-8'# locale for monetary formatting
lc_numeric = 'en_US.UTF-8'# locale for number formatting
lc_time = 'en_US.UTF-8'# locale for time formatting
```

```
# default configuration for text search
default_text_search_config = 'pg_catalog.english'

# - Shared Library Preloading -

#shared_preload_libraries = ''# (change requires restart)
#local_preload_libraries = ''
#session_preload_libraries = ''
#jit_provider = 'llvmjit'# JIT library to use

# - Other Defaults -

#dynamic_library_path = '$libdir'


#------------------------------------------------------------------------------
# LOCK MANAGEMENT
#------------------------------------------------------------------------------

#deadlock_timeout = 1s
#max_locks_per_transaction = 64 # min 10
# (change requires restart)
#max_pred_locks_per_transaction = 64 # min 10
# (change requires restart)
#max_pred_locks_per_relation = -2 # negative values mean
# (max_pred_locks_per_transaction
#  / -max_pred_locks_per_relation) - 1
#max_pred_locks_per_page = 2            # min 0


#------------------------------------------------------------------------------
# VERSION AND PLATFORM COMPATIBILITY
#------------------------------------------------------------------------------

# - Previous PostgreSQL Versions -

#array_nulls = on
#backslash_quote = safe_encoding # on, off, or safe_encoding
```

```
#default_with_oids = off
#escape_string_warning = on
#lo_compat_privileges = off
#operator_precedence_warning = off
#quote_all_identifiers = off
#standard_conforming_strings = on
#synchronize_seqscans = on

# - Other Platforms and Clients -

#transform_null_equals = off


#------------------------------------------------------------------------------
# ERROR HANDLING
#------------------------------------------------------------------------------

#exit_on_error = off # terminate session on any error?
#restart_after_crash = on # reinitialize after backend crash?
#data_sync_retry = off # retry or panic on failure to fsync
# data?
# (change requires restart)


#------------------------------------------------------------------------------
# CONFIG FILE INCLUDES
#------------------------------------------------------------------------------

# These options allow settings to be loaded from files other than the
# default postgresql.conf.

#include_dir = ''# include files ending in '.conf' from
# a directory, e.g., 'conf.d'
#include_if_exists = ''# include file only if it exists
#include = ''# include file


#------------------------------------------------------------------------------
# CUSTOMIZED OPTIONS
```

```
#------------------------------------------------------------------------------

# Add settings for extensions here
```

A.1 Anexo do Arquivo de Configuração do PostgreSQL - PostgreSQL *Singlecore*

```
#------------------------------------------------------------------
# PARGRES MULTICORE
#------------------------------------------------------------------
max_worker_processes = 0 # (change requires restart)
max_parallel_workers = 0 # maximum number of max_worker_processes that
max_parallel_workers_per_gather = 0 # taken from max_parallel_workers
```

Apêndice B

Planos de Consulta - Análise de Partição e Subpartição

B.1 Consulta Q1

```
------------------------------------------------------------------
HashAggregate  (cost=33661.66..33661.80 rows=6 width=196)
(actual time=112.026..112.031 rows=4 loops=1)
   Group Key: l_returnflag, l_linestatus
   ->  Index Scan using lineitem_pkey on lineitem
   (cost=0.58..32896.41 rows=18006 width=25)
   (actual time=6.008..81.101 rows=15746 loops=1)
         Index Cond: ((l_orderkey >= 0) AND
         (l_orderkey < 16000))
         Filter: (l_shipdate <= '1998-09-02 00:00:00'::
         timestamp without time zone)
         Rows Removed by Filter: 252
 Planning Time: 243.535 ms
 Execution Time: 166.102 ms
(8 rows)
------------------------------------------------------------------

------------------------------------------------------------------
HashAggregate  (cost=8419.97..8420.10 rows=6 width=196)
(actual time=5.965..5.969 rows=4 loops=1)
   Group Key: l_returnflag, l_linestatus
   ->  Index Scan using lineitem_pkey on lineitem
   (cost=0.58..8228.67 rows=4501 width=25)
   (actual time=0.030..1.157 rows=3996 loops=1)
```

```
         Index Cond: ((l_orderkey >= 0) AND
         (l_orderkey < 4000))
         Filter: (l_shipdate <= '1998-09-02 00:00:00'::
         timestamp without time zone)
         Rows Removed by Filter: 50
 Planning Time: 19.156 ms
 Execution Time: 6.074 ms
(8 rows)
-----------------------------------------------------------------
```

B.2 Consulta Q4

```
-----------------------------------------------------------------
GroupAggregate  (cost=1779.67..1779.69 rows=1 width=24)
(actual time=76.019..76.034 rows=5 loops=1)
   Group Key: orders.o_orderpriority
   ->  Sort  (cost=1779.67..1779.68 rows=1 width=16)
   (actual time=76.009..76.014 rows=121 loops=1)
         Sort Key: orders.o_orderpriority
         Sort Method: quicksort  Memory: 30kB
         ->  Nested Loop Semi Join  (cost=1.15..1779.66
         rows=1 width=16) (actual time=0.217..71.354
         rows=121 loops=1)
               ->  Index Scan using orders_pkey on orders
               (cost=0.57..257.60 rows=176 width=24)
               (actual time=0.058..70.515 rows=139 loops=1)
                     Index Cond: ((o_orderkey >= 0) AND
                     (o_orderkey < 16000))
                     Filter: ((o_orderdate >= '1993-07-01'
                     ::date) AND (o_orderdate < '1993-10-01
                     00:00:00'::timestamp without time zone))
                     Rows Removed by Filter: 3860
               ->  Index Scan using lineitem_pkey on lineitem
               (cost=0.58..8.60 rows=1 width=8)
               (actual time=0.005..0.005 rows=1 loops=139)
                     Index Cond: ((l_orderkey =
                     orders.o_orderkey) AND (l_orderkey >= 0)
                     AND (l_orderkey < 16000))
                     Filter: (l_commitdate < l_receiptdate)
```

```
                    Rows Removed by Filter: 1
 Planning Time: 175.823 ms
 Execution Time: 76.379 ms
(16 rows)
```

```
GroupAggregate  (cost=457.12..457.14 rows=1 width=24)
(actual time=0.489..0.495 rows=5 loops=1)
   Group Key: orders.o_orderpriority
   ->  Sort  (cost=457.12..457.12 rows=1 width=16)
   (actual time=0.483..0.485 rows=34 loops=1)
         Sort Key: orders.o_orderpriority
         Sort Method: quicksort  Memory: 26kB
         ->  Nested Loop Semi Join  (cost=1.15..457.11
         rows=1 width=16) (actual time=0.052..0.432
         rows=34 loops=1)
               ->  Index Scan using orders_pkey on orders
               (cost=0.57..70.57 rows=44 width=24)
               (actual time=0.042..0.257 rows=38 loops=1)
                     Index Cond: ((o_orderkey >= 0)
                     AND (o_orderkey < 4000))
                     Filter: ((o_orderdate >= '1993-07-01'
                     ::date) AND (o_orderdate < '1993-10-01
                     00:00:00'::timestamp without time zone))
                     Rows Removed by Filter: 961
               ->  Index Scan using lineitem_pkey on lineitem
               (cost=0.58..8.60 rows=1 width=8)
               (actual time=0.004..0.004 rows=1 loops=38)
                     Index Cond: ((l_orderkey =
                     orders.o_orderkey) AND (l_orderkey >= 0)
                     AND (l_orderkey < 4000))
                     Filter: (l_commitdate < l_receiptdate)
                     Rows Removed by Filter: 1
 Planning Time: 26.343 ms
 Execution Time: 0.585 ms
(16 rows)
```

B.3 Consulta Q6

```
-------------------------------------------------------------
Aggregate  (cost=33081.02..33081.03 rows=1 width=32)
(actual time=4.550..4.550 rows=1 loops=1)
   ->  Index Scan using lineitem_pkey on lineitem
   (cost=0.58..33079.28 rows=348 width=12)
   (actual time=0.051..4.452 rows=300 loops=1)
         Index Cond: ((l_orderkey >= 0) AND
         (l_orderkey < 16000))
         Filter: ((l_shipdate >= '1994-01-01'::date)
         AND (l_shipdate < '1995-01-01 00:00:00'::
         timestamp without time zone) AND
         (l_discount >= 0.05) AND (l_discount <= 0.07)
         AND (l_quantity < '24'::numeric))
         Rows Removed by Filter: 15698
 Planning Time: 18.164 ms
 Execution Time: 4.610 ms
(7 rows)
-------------------------------------------------------------

-------------------------------------------------------------
Aggregate  (cost=8274.82..8274.83 rows=1 width=32)
(actual time=1.198..1.198 rows=1 loops=1)
   ->  Index Scan using lineitem_pkey on lineitem
   (cost=0.58..8274.39 rows=87 width=12)
   (actual time=0.047..1.165 rows=83 loops=1)
         Index Cond: ((l_orderkey >= 0) AND
         (l_orderkey < 4000))
         Filter: ((l_shipdate >= '1994-01-01'::date)
         AND (l_shipdate < '1995-01-01 00:00:00'::
         timestamp without time zone) AND
         (l_discount >= 0.05) AND (l_discount <= 0.07)
         AND (l_quantity < '24'::numeric))
         Rows Removed by Filter: 3963
 Planning Time: 18.897 ms
 Execution Time: 1.250 ms
(7 rows)
-------------------------------------------------------------
```

B.4 Consulta Q12

```
------------------------------------------------------------------
GroupAggregate  (cost=33846.67..33848.52 rows=7 width=27)
(actual time=9.202..9.214 rows=2 loops=1)
   Group Key: lineitem.l_shipmode
   ->  Sort  (cost=33846.67..33846.89 rows=89 width=27)
   (actual time=9.182..9.186 rows=74 loops=1)
         Sort Key: lineitem.l_shipmode
         Sort Method: quicksort  Memory: 30kB
         ->  Nested Loop  (cost=1.15..33843.79 rows=89
         width=27) (actual time=3.198..9.125 rows=74
         loops=1)
               ->  Index Scan using lineitem_pkey on
               lineitem  (cost=0.58..33079.28 rows=89 width=19)
               (actual time=3.188..8.865 rows=74 loops=1)
                     Index Cond: ((l_orderkey >= 0)
                     AND (l_orderkey < 16000))
                     Filter: ((l_shipmode = ANY
                     ('{MAIL,SHIP}'::bpchar[])) AND
                     (l_commitdate < l_receiptdate) AND
                     (l_shipdate < l_commitdate) AND
                     (l_receiptdate >= '1994-01-01'::date)
                     AND (l_receiptdate < '1995-01-01 00:00:00'
                     ::timestamp without time zone))
                     Rows Removed by Filter: 15924
               ->  Index Scan using orders_pkey on orders
               (cost=0.57..8.59 rows=1 width=24)
               (actual time=0.003..0.003 rows=1 loops=74)
                     Index Cond: (o_orderkey = lineitem.l_orderkey)
 Planning Time: 22.834 ms
 Execution Time: 11.928 ms
(14 rows)
------------------------------------------------------------------

------------------------------------------------------------------
GroupAggregate  (cost=8463.86..8464.37 rows=7 width=27)
(actual time=1.487..1.490 rows=2 loops=1)
   Group Key: lineitem.l_shipmode
   ->  Sort  (cost=8463.86..8463.91 rows=22 width=27)
```

```
   (actual time=1.476..1.477 rows=17 loops=1)
         Sort Key: lineitem.l_shipmode
         Sort Method: quicksort  Memory: 26kB
         ->  Nested Loop  (cost=1.15..8463.37 rows=22
         width=27) (actual time=0.130..1.437 rows=17
         loops=1)
               ->  Index Scan using lineitem_pkey on
               lineitem  (cost=0.58..8274.39 rows=22 width=19)
               (actual time=0.122..1.384 rows=17 loops=1)
                     Index Cond: ((l_orderkey >= 0) AND
                     (l_orderkey < 4000))
                     Filter: ((l_shipmode = ANY
                     ('{MAIL,SHIP}'::bpchar[])) AND
                     (l_commitdate < l_receiptdate) AND
                     (l_shipdate < l_commitdate) AND
                     (l_receiptdate >= '1994-01-01'::date)
                     AND (l_receiptdate < '1995-01-01 00:00:00'
                     ::timestamp without time zone))
                     Rows Removed by Filter: 4029
               ->  Index Scan using orders_pkey on orders
               (cost=0.57..8.59 rows=1 width=24)
               (actual time=0.003..0.003 rows=1 loops=17)
                     Index Cond: (o_orderkey = lineitem.l_orderkey)
 Planning Time: 23.247 ms
 Execution Time: 1.574 ms
(14 rows)
-------------------------------------------------------------
```

B.5 Consulta Q18

```
-------------------------------------------------------------
GroupAggregate  (cost=33547.91..33547.93 rows=1 width=75)
(actual time=41.561..41.561 rows=1 loops=1)
   Group Key: customer.c_custkey, orders.o_orderkey
   ->  Sort  (cost=33547.91..33547.91 rows=1 width=48)
   (actual time=41.551..41.552 rows=7 loops=1)
         Sort Key: customer.c_custkey, orders.o_orderkey
         Sort Method: quicksort  Memory: 25kB
         ->  Nested Loop  (cost=2.17..33547.90 rows=1
```

```
width=48) (actual time=37.397..41.525 rows=7
loops=1)
      ->  Nested Loop  (cost=1.59..33539.29
      rows=1 width=51) (actual time=37.380..41.506
      rows=1 loops=1)
            ->  Merge Join  (cost=1.15..33530.90
            rows=1 width=36) (actual time=3.586..7.711
            rows=1 loops=1)
                  Merge Cond: (orders.o_orderkey =
                  lineitem_1.l_orderkey)
                  ->  Index Scan using orders_pkey
                  on orders  (cost=0.57..234.39
                  rows=4641 width=28) (actual
                  time=0.031..0.353 rows=1723 loops=1)
                        Index Cond: ((o_orderkey >= 0)
                        AND (o_orderkey < 16000))
                  ->  GroupAggregate
                  (cost=0.58..33210.39 rows=5961 width=8)
                  (actual time=2.996..7.118 rows=1 loops=1)
                        Group Key: lineitem_1.l_orderkey
                        Filter: (sum(lineitem_1.l_quantity)
                        > '300'::numeric)
                        Rows Removed by Filter: 3998
                        ->  Index Scan using
                        lineitem_pkey on lineitem lineitem_1
                        (cost=0.58..32850.69 rows=18287
                        width=13)
                        (actual time=0.014..3.170 rows=15998
                        loops=1)
                              Index Cond: ((l_orderkey >= 0)
                              AND (l_orderkey < 16000))
            ->  Index Scan using customer_pkey on customer
            (cost=0.44..8.38 rows=1 width=23)
            (actual time=33.791..33.791 rows=1 loops=1)
                  Index Cond: (c_custkey = orders.o_custkey)
      ->  Index Scan using lineitem_pkey on lineitem
      (cost=0.58..8.60 rows=1 width=13)
      (actual time=0.011..0.013 rows=7 loops=1)
            Index Cond: ((l_orderkey = orders.o_orderkey)
```

```
                    AND (l_orderkey >= 0) AND (l_orderkey < 16000))
 Planning Time: 115.948 ms
 Execution Time: 42.089 ms
(23 rows)
-----------------------------------------------------------------

-----------------------------------------------------------------
GroupAggregate  (cost=8411.96..8411.98 rows=1 width=75)
(actual time=1.799..1.799 rows=0 loops=1)
   Group Key: customer.c_custkey, orders.o_orderkey
   ->  Sort  (cost=8411.96..8411.96 rows=1 width=48)
   (actual time=1.797..1.797 rows=0 loops=1)
         Sort Key: customer.c_custkey, orders.o_orderkey
         Sort Method: quicksort  Memory: 25kB
         ->  Nested Loop  (cost=2.17..8411.95 rows=1
         width=48) (actual time=1.781..1.781 rows=0
         loops=1)
               ->  Nested Loop  (cost=1.59..8403.34
               rows=1 width=51) (actual time=1.779..1.779
               rows=0 loops=1)
                     ->  Merge Join  (cost=1.15..8394.91
                     rows=1 width=36) (actual
                     time=1.779..1.779 rows=0 loops=1)
                           Merge Cond: (orders.o_orderkey
                           = lineitem_1.l_orderkey)
                           ->  Index Scan using orders_pkey
                           on orders  (cost=0.57..64.77
                           rows=1160 width=28) (actual
                           time=0.030..0.030 rows=1 loops=1)
                                 Index Cond: ((o_orderkey >= 0)
                                 AND (o_orderkey < 4000))
                           ->  GroupAggregate
                           (cost=0.58..8308.29
                           rows=1515 width=8)
                           (actual time=1.746..1.746
                           rows=0 loops=1)
                                 Group Key: lineitem_1.l_orderkey
                                 Filter: (sum(lineitem_1.l_quantity)
                                 > '300'::numeric)
                                 Rows Removed by Filter: 999
```

```
                                    ->  Index Scan using lineitem_pkey
                                    on lineitem lineitem_1
                                    (cost=0.58..8217.25 rows=4571
                                    width=13) (actual
                                    time=0.011..0.746 rows=4046
                                    loops=1)
                                          Index Cond: ((l_orderkey >= 0)
                                          AND (l_orderkey < 4000))
                              ->  Index Scan using customer_pkey
                              on customer
                              (cost=0.44..8.43 rows=1 width=23)
                              (never executed)
                                    Index Cond: (c_custkey = orders.o_custkey)
                        ->  Index Scan using lineitem_pkey on lineitem
                        (cost=0.58..8.60 rows=1 width=13) (never executed)
                              Index Cond: ((l_orderkey = orders.o_orderkey)
                              AND (l_orderkey >= 0) AND (l_orderkey < 4000))
 Planning Time: 23.957 ms
 Execution Time: 1.944 ms
(23 rows)
-----------------------------------------------------------------
```