Tuning de SQL

Melhore a performance de suas aplicações Oracle

Casa do Código | alura

EDUARDO GONÇALVES

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Vivian Matsui
Sabrina Barbosa

[2021]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

casadocodigo.com.br

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

grupo alura

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código é a editora do Grupo Alura, grupo que nasceu da vontade de criar uma plataforma de ensino com o objetivo de incentivar a transformação pessoal e profissional através da tecnologia.

Juntas, as empresas do Grupo constroem uma verdadeira comunidade colaborativa de aprendizado em programação, negócios, design, marketing e muito mais, oferecendo inovação na evolução dos seus alunos e alunas através de uma verdadeira experiência de encantamento.

Venha conhecer os cursos da Alura e siga-nos em nossas redes sociais.

alura.com.br

@casadocodigo

@casadocodigo

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

ISBN

Impresso: 978-65-86110-94-4

Digital: 978-65-86110-95-1

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

AGRADECIMENTOS

Agradeço a Deus em primeiro lugar, por esta oportunidade de mais uma vez poder compartilhar conhecimento. Que este conhecimento venha a servir às pessoas. Obrigado, Senhor, que tudo seja para Tua honra e glória! Minha gratidão também ao meu amigo de longa data Emanoel Koslosky, por todo o suporte e ajuda na revisão dos conteúdos. Dedico este livro ao meu filho Davi Eduardo, que Deus lhe conceda sabedoria e graça todos os dias de sua vida! Agradecimentos especiais a minha esposa Thais, o amor da minha vida, e as minhas duas princesas, Lisyê e Julia Helena, filhas amadas!

SOBRE O AUTOR

Formado em Tecnologia da Informação, possui mais de 15 anos de experiência em análise e desenvolvimento de sistemas voltados para tecnologias Oracle. Trabalhou para grandes empresas como Lojas Renner, Mundial S.A, Tigre S.A, Pernambucanas, Engie, Portobello Cerâmica, Bematech, Positivo, Ailos entre outras. É instrutor de cursos nas linguagens Oracle SQL e Oracle PL/SQL. Atualmente, desenvolve projetos na Supero Tecnologia.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

PREFÁCIO

Talvez você deva estar se perguntando: por que preciso entender sobre Tuning? Ou talvez: como nunca li ou vi nada sobre isso até hoje, depois de todo esse tempo trabalhando com banco de dados? Será que é realmente interessante que eu conheça esse assunto? Pois bem, durante minha trajetória na área de TI, mais precisamente, trabalhando com análise e desenvolvimento de sistemas, sempre fui muito curioso. Na verdade, eu tenho um desejo forte de entender como as coisas funcionam. Sempre tive essa curiosidade. Falar para mim que isso funciona assim, porque sim, nunca foi do meu agrado, principalmente quando se tratava de questões relacionadas ao meu trabalho. Entendo que, para ser o melhor naquilo que se faz, você tem que entender como as coisas funcionam, como foram idealizadas. Não fazer por fazer e torcer para dar certo. Isso me ajudou muito durante esta trajetória, pois através desse ímpeto de querer saber das coisas fui agregando mais conhecimento e, o melhor de tudo, adquirindo facilidade de entendimento.

A receita na verdade é simples, quanto mais treino, mais resultado. É como exercitar os músculos. Com o cérebro é a mesma coisa. Lembro de como era um aluno mediano em quase todas as matérias no colégio. Meus pesadelos eram Português e Matemática, não gostava de redação nem de fazer cálculos. Quanto ao português, pois bem, hoje, depois de tanto treino, escrevo livros, então acho que alguma coisa mudou e valeu muito a pena o esforço, e a matemática… Bem, ficou mais minha amiga, não tão íntima, mas nos damos bem.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

À medida que fui conhecendo mais sobre programação e banco de dados, fui entrando neste mundo do *Tuning*. Comecei sem saber muita coisa, mas fui lendo materiais e manuais sobre o assunto. Na época, não tinha muito conteúdo, apenas materiais em inglês muito pouco práticos e mais sobre conceitos, sem muitos exemplos e casos do dia a dia. Aprendi muito (quero dizer, apanhei muito) com os problemas que foram surgindo durante minha trajetória na área e fui gostando deste tema.

Durante o tempo em que trabalhei com Tuning, resolvi desde problemas fáceis até problemas mais complicados ou trabalhosos de se resolver. Muitas vezes tive que recorrer a manuais e também a conhecidos da área, colegas de trabalho e da faculdade, para entender mais sobre o assunto ou como os bancos de dados trabalhavam em um nível mais distante da área de desenvolvimento. Mas valeu a pena. Conheci pessoas incríveis, com muito conhecimento sobre o assunto. Com o tempo fui adquirindo mais entendimento e tinha a sensação de ser mais fácil resolver os problemas.

Peguei alguns desafios pela frente e sempre tinha a pressão de ter que resolver em um prazo curto. Um faturamento parado, uma logística atrasada ou uma obrigação fiscal com entrega em cima do laço, foram alguns dos desafios que envolveram problemas de performance com os quais pude atuar. Lembro até da primeira vez em que viajei de avião, fazendo um bate e volta a São Paulo, para resolver um problema de performance referente a um relatório contábil que levava muitas horas para rodar, mas que tinha que funcionar em bem menos tempo, pois, caso contrário, a empresa não poderia fazer as conciliações no prazo em que precisava. E lá fui eu. Enfim, foram várias as experiências que tive. Foi sensacional

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

e valeu muito a pena! Nossa vida é curta e passageira, então o que você puder adquirir de conhecimento certamente vai torná-la mais interessante, isso vale para a vida pessoal, trabalho, família e todo o resto.

Procurei reunir neste livro um pouco de conhecimento sobre monitoramento e ajuste da performance para a linguagem Oracle SQL. O livro aborda conceitos importantes sobre o assunto através de explicações e exemplos. São demonstradas as ferramentas para o trabalho de monitoramento, e como extrair e gerar informações importantes para que o banco de dados possa retornar resultados dentro de uma performance mais adequada. Espero que você aceite este desafio. No mínimo, você conhecerá algo novo ou relembrará o assunto, caso já tenha tido contato. Desejo a você uma ótima leitura!

SOBRE O LIVRO

A CULTURA DA PERFORMANCE

Conforme será explanado durante todo o livro, nosso objetivo maior será demonstrar as técnicas e ferramentas para se analisar instruções SQL, tendo em vista que, quando estamos falando de performance referente a programas, este se torna o ponto principal de análise. A forma como as informações estão sendo acessadas ou processadas em um banco de dados é a maior causa de demoras na execução de programas e muitas vezes são necessários ajustes para corrigir esses problemas.

Além do ajuste de instruções SQL, dependendo do caso,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

podemos propor outros recursos da linguagem ou formas diferentes de lidar com cada situação, sem prejudicar o resultado final. Você vai perceber que às vezes esse trabalho vai envolver a reescrita de um programa e outro, no caso de um ajuste de SQL ainda não surtir o efeito desejado. Por outro lado, vamos explorar os recursos e ferramentas do próprio banco de dados para gerar, extrair e analisar dados estatísticos com o intuito de entender cenários e propor soluções para os mais diversos problemas relacionados à performance.

Antes de mais nada, vamos falar um pouco sobre essa busca pela performance, quando e como fazê-la, e entender as premissas de como executar esse trabalho, tanto de maneira preventiva, como de maneira corretiva. Você verá que muitas vezes poderá resolver questões desse tipo alinhando ou definindo processos, sem a necessidade de codificar. Mas às vezes será preciso colocar a mão na massa – e de forma corretiva. A ideia é mudar esse panorama e criar a cultura de pensar em desempenho desde o início, desde a análise de uma aplicação, até sua concepção e implantação.

O QUE VEREMOS?

Serão demonstradas diversas técnicas para se escrever instruções SQL de uma forma mais adequada, para que o otimizador do banco de dados possa escolher um plano de execução eficaz para executá-las. Veremos algumas dessas técnicas, como a criação de índices, o uso das sugestões (Hints), o ajuste da sintaxe, uso dos métodos de acesso às tabelas, cursores e ponteiros.

Falaremos também sobre as ferramentas de auxílio ao trabalho de Tuning de SQL, mais precisamente as que analisam instruções

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

SQL ou que geram estatísticas para o otimizador do banco de dados. O uso delas trará as informações que auxiliarão você na análise de aplicações problemáticas, na geração de estatísticas para o otimizador, na visualização dos planos de execução e na obtenção de informações sobre o processamento das instruções executadas no banco de dados, ferramentas estas que acompanham o próprio banco de dados Oracle.

Não utilizaremos ferramentas gráficas ou de outros fornecedores que não sejam do próprio banco de dados. A ideia é trabalhar com o que geralmente nós, desenvolvedores, já temos em mãos no ambiente do banco de dados. Na grande maioria das vezes, a pessoa desenvolvedora não tem acesso a ferramentas administrativas ou a ferramentas mais sofisticadas, como gráficos, botões e relatórios estatísticos mais elaborados. A ideia aqui é demonstrar que, com as ferramentas padrões já existentes no próprio banco de dados, já podemos fazer muito, sem inviabilizar este trabalho tendo que adquirir licenças de outros programas, mais instalações ou configurações mais avançadas, bem como despender tempo para aprendê-las.

A busca por um melhor desempenho não envolve apenas questões técnicas, mas também o entendimento de como uma aplicação é utilizada. Entender como um usuário utiliza tal aplicação ou como ela foi projetada pode dar a você indícios do caminho por onde começar. Não necessariamente, como já foi dito, você precisará rescrever um programa ou criar um índice para melhorar um determinado cenário. Às vezes, o problema pode vir de um procedimento mal executado por aqueles que operam parte de um sistema ou até mesmo da falta de conhecimento do sistema como um todo. Você deve ter em mente

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

que não existe exatamente uma receita de bolo de como melhorar a performance de um programa. Temos que garimpar para encontrar o problema e aplicar possíveis soluções até chegar a um resultado aceitável.

PARA QUEM SE DESTINA ESTE LIVRO?

Este livro é voltado para o desenvolvedor e a desenvolvedora de aplicações, analista de sistemas ou para pessoas de outras áreas que querem conhecer essas técnicas. Você verá que a busca pela performance não dependerá, muitas vezes, única e exclusivamente, de um indivíduo, ou seja, somente de você, mas sim do apoio de profissionais de outras áreas e até dos usuários do sistema. As dicas, ferramentas e técnicas apresentadas aqui são direcionadas a quem desenvolve sistemas. A ideia é que todos os recursos apresentados aqui possam ser aplicados por você, dev, no seu dia a dia, exceto para determinados acessos em que você precisará do apoio de um DBA (*Database Administrator*/ Administrador de banco de dados).

DO QUE TRATA ESTE LIVRO?

A ideia central deste livro é mostrar as técnicas e ferramentas de Tuning de SQL para instruções SQL no ambiente Oracle, tanto as que são executadas de forma isolada, quanto aquelas que são executadas de dentro de programas PL/SQL, visando fornecer subsídios para você buscar a performance mais adequada para suas aplicações, e encontrar soluções para os problemas de desempenho. Teremos como ênfase dicas de como melhor escrever essas instruções SQL e de como solucionar problemas utilizando os

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

recursos desta linguagem e do próprio banco de dados. As técnicas e ferramentas aqui demonstradas são específicas para o banco de dados Oracle da empresa Oracle Corporation.

Não trataremos aqui de questões relacionadas ao monitoramento e ajuste da performance referente às áreas de infraestrutura de hardware e redes, sistemas operacionais, parametrizações e configurações específicas do banco de dados. Vamos focar, em primeiro lugar, em ajudar você a pensar em desempenho quando escrever suas aplicações ou necessitar melhorar ou corrigir programas já existentes. Quando me ouvir falar em melhorar seus programas ou aplicações, estarei me referindo, na grande maioria das vezes, a uma instrução SQL, a um conjunto de instruções SQL, ou ainda, a instruções SQL que fazem parte de um ou mais programas PL/SQL, por exemplo.

Depois disso, vamos ver como extrair as informações necessárias das próprias execuções, bem como das diversas estatísticas armazenadas no banco de dados, para as diversas análises de monitoramento e ajuste da performance. Tudo isso, basicamente, para deixar seus programas mais rápidos.

PRÉ-REQUISITOS

Se você desejar acompanhar ou tentar simular os exemplos demonstrados no livro, você pode baixar uma versão gratuita do banco de dados Oracle no site da própria empresa Oracle (https://www.oracle.com/downloads/). Você poderá instalar na sua própria máquina. Trata-se de uma versão mais básica do banco de dados e de simples instalação.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Vale lembrar que, quando falamos sobre questões de performance, parâmetros como ambiente, memória, condições da instalação do banco de dados etc. podem influenciar nos dados apresentados. Desta forma, o exemplo demonstrado neste livro pode não refletir o mesmo resultado estatístico, por exemplo, em outro ambiente. Contudo, não tem problema, pois o mais importante é entender as técnicas e como as ferramentas podem ser utilizadas.

Outro requisito consiste em conhecer a linguagem SQL (e, opcionalmente, um pouco da linguagem PL/SQL), pois as técnicas e ferramentas apresentadas aqui serão para o monitoramento e ajustes das instruções escritas nessa linguagem.

Downloads

Os scripts para montar a base de dados (tabelas e dados) para a execução dos exemplos do livro, e também alguns scripts, estão disponíveis no endereço:

https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo.

Contato

Para falar com o autor, envie e-mail para eduardogoncalves.br@gmail.com.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Sumário

1 Antes da técnica **1**

1.1 Tuning de SQL 1

1.2 Monitoramento da performance 3

1.3 Opte pela prevenção, não pela manutenção 8

1.4 Modelagem é a base de tudo 10

1.5 Desenvolvedores versus DBA: Trabalho em conjunto 11

1.6 Por onde começar? 13

2 Ajuste da SQL **20**

2.1 Otimizador de consultas 22

2.2 O otimizador do Oracle 26

2.3 O Modo baseado em Custos (CBO) 31

2.4 O Modo baseado em Regras (RBO) 45

2.5 SQL x PL/SQL: Como melhorar um cenário? 54

3 Processando instruções SQL **73**

3.1 Passos para a execução de instruções SQL 73

3.2 Shared SQL Areas 77

4 Utilização de índices **88**

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

4.1 Tipos de índices 102

5 Métodos de junção de tabelas 117

5.1 Executando operações de junção no modo CBO 120

5.2 Escolhendo métodos de junção no modo CBO 121

5.3 Escolhendo planos de execução para tipos de junção no modo CBO 122

5.4 NESTED LOOPS join 127

5.5 SORT MERGE join 132

5.6 HASH join 136

5.7 CARTESIAN join 140

5.8 Comparação entre os métodos NESTED LOOPS, SORT MERGE e HASH 149

5.9 Tipos de junção 151

6 Métodos de acesso 179

6.1 Como o armazenamento e a leitura dos registros acontecem? 180

6.2 FULL TABLE SCANS 183

6.3 ROWID SCANS 185

6.4 INDEX SCANS 187

6.5 BITMAP SCANS 199

6.6 CLUSTER SCANS 204

6.7 HASH SCANS 207

6.8 SAMPLE TABLE SCANS 209

6.9 Escolhendo os métodos de acesso no modo RBO 211

6.10 Escolhendo os métodos de acesso no modo CBO 216

7 Tabela Controle (Driving Tables) 228

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

7.1 Ordenando tabelas e filtros 242
7.2 Forçando a Driving Table por hint 245
7.3 Driving Tables e os métodos de junção 250
7.4 Driving Tables e os Planos Cartesianos (CARTESIAN join) 252

8 Otimizando através de hints 266
8.1 hint CHOOSE 273
8.2 hint RULE 274
8.3 hint ALL_ROWS 276
8.4 hint INDEX 277
8.5 hint ORDERED 279
8.6 hint PARALLEL 281
8.7 hint FIRST_ROWS 284
8.8 hint FULL 285
8.9 hint LEADING 287
8.10 hint CACHE 290
8.11 hint USE_NL 291
8.12 hint USE_HASH 292
8.13 hint USE_MERGE 294
8.14 Considerações sobre o uso de hints 295

9 Planos de Execução (Explain Plan) 297
9.1 Interpretando Planos de Execução 299

10 Gerando estatísticas para o otimizador 343
10.1 O pacote dbms_stats 345
10.2 Quando eu devo gerar estatísticas? 350
10.3 Geração automática de estatísticas 351

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

11 Histogramas **360**

11.1 Gerando histogramas 368

11.2 Quando e como utilizar ou não histogramas? 373

12 Gerando e visualizando planos de execução **393**

12.1 SQL*Plus 394

12.2 explain plan 397

12.3 v$sql_plan 407

12.4 autotrace 418

12.5 dbms_xplan 425

12.6 Rastreie e visualize com trace e tkprof 464

13 Conclusão **504**

14 Referências bibliográficas **507**

Versão: 26.3.29

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 1

ANTES DA TÉCNICA

Antes de falarmos sobre as técnicas e ferramentas, propriamente ditas, e ver como isso funciona na prática, vamos entender um pouco sobre os conceitos do que é o *Tuning* de SQL, como é o trabalho de monitoramento das aplicações no banco de dados, por que é tão importante pensar em performance desde a fase de análise de um projeto de aplicação, e quais os primeiros passos para iniciar e conduzir este trabalho.

Saber qual caminho tomar, quando e como melhorar, tanto para novas implementações como para programas já existentes, é o ponto partida para o sucesso da prevenção ou manutenção de todo o sistema. Vamos agora conhecer um pouco desta base.

1.1 TUNING DE SQL

Quando automatizamos processos que antes operavam de forma manual ou por uso de outros programas, como planilhas, por exemplo, o principal intuito é obter rapidez no processamento dos dados e consistência na manipulação e obtenção das informações.

Todavia, essas informações que ficam em grande parte armazenadas em bancos de dados precisam estar disponíveis

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

constantemente e devem ser acessadas de forma rápida. Para isso, uma série de parâmetros e recursos relacionados ao banco de dados devem estar configurados e operando de forma adequada, propiciando um ambiente saudável para todas as aplicações.

É claro que não só de banco de dados vive a informação. Trata-se de um conjunto de coisas, como o sistema operacional, a infraestrutura de rede, os computadores, os servidores etc. Mas quando falamos de performance na busca por informações e na automatização dos processos, devemos nos ater às instruções SQL. Para não termos problemas dessa natureza, elas deverão estar escritas de maneira adequada, usufruindo ao máximo dos recursos que o banco de dados oferece, mas sem causar gargalos para os demais programas que fazem parte do sistema.

Na maioria das vezes, as instruções SQL escritas para buscar informações em um banco de dados possuem o objetivo de apenas trazer os dados, não importando que caminho será percorrido para realizar essa busca. A princípio, essa atitude pode não parecer problemática, mas à medida que o banco de dados for adquirindo volume de informação e mais processamentos forem sendo agregados ao sistema, pode haver concorrência por recurso, e esses problemas relacionados à performance começam a surgir se seus programas não foram bem projetados. Desta forma, o modelo de dados poderá nos dizer muito sobre a saúde das nossas aplicações e, com base nisso, pode ser necessária uma análise mais profunda das situações enfrentadas.

O Tuning de SQL é um conceito que visa, através de suas técnicas, auxiliar na otimização do desempenho na recuperação e manipulação de dados. Ele consiste no monitoramento e no ajuste

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

de programas, sendo uma parte de um conjunto de fatores que deve ser analisado com a devida importância, para que todo o ambiente de banco de dados opere de forma adequada e contribua para o melhoramento do sistema.

Outros ajustes importantes são o Tuning relacionado a rotinas de backup, o de acesso aos discos, os relacionados ao uso de memória, tráfego de redes, hardwares e outros equipamentos, e demais aspectos do operacional do banco de dados, incluindo o Tuning de sistema operacional, que também deve estar ajustado adequadamente para que o gerenciador do banco de dados possa trabalhar sem problemas.

Este livro tem como foco trabalhar essas técnicas e ferramentas sobre instruções SQL, visando melhorar o desempenho dos sistemas de software para o banco de dados Oracle, bem como evitar problemas de baixa performance no processamento das atividades que possam prejudicar um sistema inteiro.

1.2 MONITORAMENTO DA PERFORMANCE

O monitoramento e ajuste da performance, oriundos do Tuning de SQL, fazem parte do ciclo de vida de todo sistema, inclusive os que acessam os bancos de dados através da linguagem SQL ou PL/SQL (no caso do Oracle). Deve-se pensar em buscar a performance adequada já na fase de análise, para viabilizar o uso do sistema com a melhor forma de estruturá-lo.

Ao iniciar esse processo, é de fundamental importância que toda a equipe, projetistas, desenvolvedores e administradores de banco de dados (DBA) que participarão estejam cientes do fator

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

desempenho. Devemos também ter consciência de que, em resumo, desde a fase de análise de um software, passando pelo projeto, codificação, testes e manutenção, até o término de todo esse processo, é preciso manter o mesmo ritmo, sempre pensando na performance do seu uso. Isso permitirá que os requisitos de desempenho sejam lembrados em cada etapa, tendo sempre em mente a principal motivação de se informatizar um sistema, que é a possibilidade de realizar os trabalhos com maior rapidez. Portanto, todas as pessoas que estão envolvidas no processo de construção e avaliação do uso de um sistema também devem estar comprometidas com esse objetivo.

Assim sendo, é possível afirmar que a responsabilidade de um sistema estar operando com performance no mínimo aceitável é de todos os envolvidos, começando com o contato do usuário com o analista de requisitos para descrever o que o sistema precisa fazer, passando pela projetista, que em conjunto com a operação (DBAs, administradores dos Sistemas Operações e de Rede etc.) possa modelar o sistema, tanto na definição dos relacionamentos entre entidades que armazenarão os dados (pensando em crescimento futuro do sistema e base de dados, inclusive) bem como no desenho do fluxo de processo. Como você pode ver, abordar todas as fases da construção e manutenção de um sistema seria algo muito extenso. Aqui neste livro, focaremos somente nas questões relacionadas a instruções SQL e não levaremos em conta deficiências relacionadas ao modelo de dados ou demais processos.

Outro ponto importante a saber é até quando um nível maior de performance é interessante para a aplicação. Quando estamos falando de um sistema de médio a grande porte, falamos da possibilidade de existir um grande número de funcionalidades e

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

um grande volume de dados sendo manipulados por elas. Diante de um cenário assim, não é difícil encontrar dependências entre tais funcionalidades ou dados do sistema. Ao trabalhar nesse contexto, a busca de um maior desempenho em uma parte do sistema pode acarretar na perda de desempenho em outra, não por falta de uma estruturação ou modelagem mais adequada, ou por falta de conhecimento, mas por consequência de tais dependências. Nesse momento, é muito importante considerar o fator **prioridade**, com o qual será preciso avaliar os possíveis cenários, a fim de detectar em qual deles será investido mais ou menos esforço para melhorar ou manter o desempenho.

Um exemplo disso seria a escolha entre melhorar a performance em um programa responsável pela emissão de notas fiscais de um setor de vendas, ou optar pela melhoria em rotinas de busca de informações para o setor de telemarketing. Um minuto ganho em um setor de telemarketing pode ser muito mais valioso. A falta deste poderia significar prejuízo para empresa.

Diante desse contexto, pode-se dizer que o **custo** *x* **benefício** também é algo a se considerar no momento da avaliação do nível de melhoria de performance que se deseja obter. Quanto antes a preocupação com o desempenho for sentida, mais ganho será agregado até o término de um projeto, seja ele um ganho de custo ou de benefício. Veja a seguir um gráfico representando a relação custo x benefício, sobre os aspectos relacionados à performance em cada fase de vida de uma aplicação.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

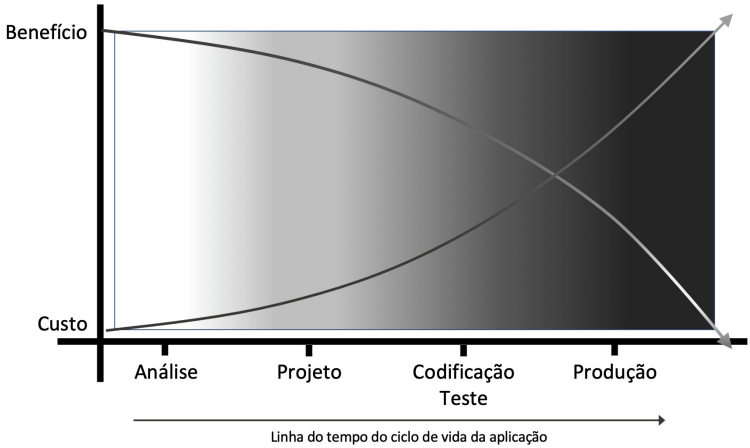

Figura 1.1: Relação de custo e benefício no ciclo de vida da aplicação. Este gráfico ilustra as linhas relacionadas aos índices de benefício e custo, e as variações que sofrem durante toda a fase do ciclo de vida de uma aplicação. O gradiente, partindo da parte clara, mostra o grau de complexidade em relação aos pontos onde os problemas tornam-se mais precários para serem solucionados, à medida que vai ficando mais escuro. Seu principal objetivo é demonstrar que, quanto mais cedo existir a preocupação com a performance, mais benefícios e menos custos serão agregados ao projeto da aplicação. Veja que, na fase de análise, temos um custo baixo e benefícios altos, mas conforme as fases avançam até chegar à implantação em produção, o cenário se inverte, e as ações referentes a melhoras na performance terão um custo mais alto e benefícios mais baixos, pois a aplicação não foi criada pensando nesses aspectos. Desta forma, algumas melhorias nesse sentido poderão ser limitadas.

A grande parte dos problemas de performance enfrentados em aplicações que utilizam linguagem SQL está relacionada a erros encontrados no modelo de dados físicos, ocupando uma parcela de 25% do total. Do restante, 65% fica a cargo de problemas relacionados à aplicação. Dessa forma, da totalidade, 90% estão ligados diretamente à equipe de desenvolvimento. Os outros 10% estão relacionados a outros aspectos.

Diante disso, é necessário que haja um planejamento, que terá como objetivo monitorar e analisar todo o nível de performance para a aplicação. Sua performance e desempenho devem ser avaliados e submetidos a testes, para saber se estão de acordo com

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

o que se espera. Esse plano de monitoramento da performance deve ser bem realizado, pois é um dos componentes de vital importância na implementação de qualquer aplicação. É ele quem vai apontar as falhas e os pontos onde atacar.

Para ser completo, o plano deve ter o auxílio de ferramentas para ajudar no monitoramento da performance e no ajuste do sistema. Um exemplo de planejamento para sistemas existentes seria manter atualizadas as estatísticas dos objetos do banco de dados de tempos em tempos, criando uma rotina de atualização. Monitorar as aplicações que mais consomem recursos do banco de dados também pode ser uma ótima ideia. Através das ferramentas do banco, você pode levantar essas informações.

Quando estamos falando de novos desenvolvimentos, podemos tentar prever questões relacionadas ao volume de dados a serem recuperados ou manipulados nas novas funcionalidades do sistema. Conversar com usuários sobre o uso de determinadas informações e rotinas de processamentos, em horários e dias específicos, mapear qual a importância e a prioridade de cada informação tratada e a rapidez com que isso deve acontecer, esses são exemplos de como você pode se planejar para um monitoramento efetivo e analisar os cenários existentes.

Quanto às atividades de monitoramento, podem ser classificadas em dois tipos: atividades contínuas (proativas), ou sob demanda (reativas). Atividades que são exercidas continuamente buscam a prevenção no sentido de evitar os problemas que estarão por surgir. Já as atividades sob demanda são efetuadas mediante o surgimento de um problema. Na maioria das vezes, acredita-se que o processo de ajuste deve ser feito quando a aplicação em questão

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

não está sendo adequadamente eficaz e sua performance não está de acordo com o que se deseja. Na verdade, neste ponto pode ser muito tarde para se conseguir êxito na efetivação deste processo sem um alto custo com relação ao benefício. Isso porque a manutenção para tentar buscar tais melhoras no desempenho pode exigir mexer em muitas partes do sistema, podendo levar tempo. Sem falar no cuidado que se deve ter para não trazer complicações em processos que estão funcionando normalmente.

1.3 OPTE PELA PREVENÇÃO, NÃO PELA MANUTENÇÃO

É muito comum nós, pessoas desenvolvedoras de sistemas, não pensarmos muito em performance, mas apenas em resolver o problema ou a necessidade em questão – codificar. A primeira ideia que vem na mente é codificar e ver o resultado. Além disso, nem sempre estamos trabalhando com uma massa de dados considerável, o que nos induz ao erro - o de não testar com grandes volumes de dados. Então, quando atingimos o resultado, ficamos orgulhosos e mandamos ver na produção e nos deparamos com programas lentos, programas que consomem muita memória e processamento, e o pior de todos os males - o tempo.

Muitas vezes o resultado dessa lentidão ou consumo excessivo de processamento não aparece na hora, às vezes levam semanas, meses e até anos. Dessa forma, tenha em mente que é possível, sim, melhorar o código já no momento em que ele está sendo escrito e que não se trata somente de enfeitar, ser metódico, fazer bonitinho, e sim, de colher bons frutos lá na frente. Se você não vem de uma cultura de pensar orientado a performance no momento em que

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

está programando, não é vergonha nenhuma utilizar refatoração de código. Escreva-o do jeito que você sabe e depois vá lapidando-o até deixá-lo mais ajustado.

Conhecer a tecnologia que você está utilizando também vai permitir trabalhar com outros recursos e componentes das linguagens que foram pensados exatamente para melhorar a produtividade e que visam custar menos ao banco de dados.

Falando em *custo*, você vai me ver usar diversas vezes esta palavra quando estiver falando de instruções SQL ou programas que deterioram memória e processamento no banco de dados e, principalmente, do tempo que levam para isso. Por exemplo: Quanto esta instrução SQL está "custando" ao otimizador do banco de dados?

Ao pensar em criar um cenário adequado para as aplicações, que comprometa o mínimo possível de desempenho, podemos considerar que teremos menos problemas se:

- O hardware puder atender às exigências do usuário e de todo o ecossistema que abriga tais aplicações.
- O banco de dados for projetado cuidadosamente e visando alto desempenho.
- Os desenvolvedores de aplicações criarem instruções SQL eficientes.

O fato é que, se forem tomadas decisões erradas no início do processo de desenvolvimento do sistema, ou se as expectativas dos usuários forem maiores em relação à performance, você deverá considerar a melhoria do desempenho. Mas nesse caso, você estará trabalhando já no campo da manutenção, e quanto mais você adiar

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

o processo de ajuste, maior será o custo em tempo e recursos. Neste caso, fica óbvio ver que prevenir é melhor que remediar!

1.4 MODELAGEM É A BASE DE TUDO

Quando utilizamos a normalização em um projeto de banco de dados devemos nos concentrar em um objetivo muito importante, o de eliminar redundâncias. Isso para obtermos a melhor forma de manutenção dos dados e a performance mais adequada. Contudo, se seu banco de dados já existe e a estrutura dele não for a mais adequada, se não for possível jogar tudo fora e começar novamente, o jeito é buscar meios para otimizar seu sistema, focando nas situações mais críticas.

Dessa forma, a dica é: se vai construir algo do zero, já construa considerando questões de performance. Como? Aplique os conceitos de modelagem de banco de dados. Modele da forma mais adequada possível utilizando-se dos conceitos e regras desta técnica. Você verá que são ações simples, mas muito eficazes.

Uma vez ouvi de uma colega de trabalho a seguinte frase: "O modelo de dados conta a história". Então, se você for relapso lá no início do projeto, poderá sofrer as consequências no presente ou talvez no futuro. Então, faça esse trabalho de forma adequada! No início, talvez, a falta ou o ganho de performance não serão sentidos, mas com o decorrer do tempo, se você não pensou nela lá atrás, no futuro poderá ter a amarga constatação de que não será possível recomeçar, apenas amenizar a situação.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

1.5 DESENVOLVEDORES VERSUS DBA: TRABALHO EM CONJUNTO

Embora o conteúdo deste livro seja específico para pessoas desenvolvedoras, vale lembrar que o sucesso deste trabalho de monitoramento e ajuste da performance não depende somente de você que desenvolve. Mesmo porque, quando estamos falando de performance como um todo, não somente de aplicação, existem outras peças-chaves, ou melhor dizendo, profissionais-chaves, que poderão nos auxiliar e muito nesse processo.

Um desses profissionais é o DBA. O DBA (*Database Administrator*), que é o Administrador de Banco de Dados, é o responsável por todas as questões técnicas relacionadas especificamente ao funcionamento do banco de dados. É essa pessoa quem monitora a saúde do banco de dados e por isso é quem acaba sinalizando, também, possíveis problemas de desempenho em determinados programas.

Através de ferramentas ou scripts, esse profissional vai poder listar as top 10 instruções SQL que mais consomem recursos no banco de dados, por exemplo, além de ser quem pode nos conseguir alguns dos acessos fundamentais para realizarmos esse trabalho de Tuning (então, tratem bem este colega...), sem contar que o DBA possui conhecimentos que podem nos ajudar a entender melhor o problema e até propor soluções, se for o caso. Se os problemas envolvem ajustes no banco de dados, o DBA poderá conduzir a situação, ajudar na solução, ou melhor direcioná-lo.

Alguns DBAs vêm de uma carreira de desenvolvimento e isso

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

pode ser muito útil, pois possuem conhecimento de codificação e dos recursos da linguagem. Alguns até já fazem este trabalho de `tunar` determinados programas e até já monitoram seu ambiente quanto à performance de aplicações. Outro ponto importante é que eles estão muito próximos da infraestrutura como um todo e poderão, talvez, até nos auxiliar em outros fatores externos que possam ocasionar tais problemas, como tráfego de rede, sistema operacional etc.

Recomendo que, antes de sair colocando a mão na massa para resolver algum problema ou para estruturar o banco de dados de uma aplicação de nível médio a complexo, principalmente em ambientes corporativos, troque uma ideia com o DBA, talvez o da sua empresa, se estiver ao alcance. Muitas vezes, a solução que você tem em mente é de extremo interesse dele, principalmente aquela que envolve a criação ou alteração de índices, criação ou alteração de objetos, ou a manipulação de registros em massa, pois um índice mal projetado ou uma mudança não muito pensada pode afetar milhões de registros e acarretar problemas em outras aplicações, até causar uma parada no banco de dados do sistema, por exemplo.

Outra dica é não pensar que existe conhecimento que deve ser específico e restrito para cada um desses profissionais. Conhecimento nunca é demais. O entendimento de conceitos sobre como o banco de dados armazena ou faz a leitura das informações no detalhe, como leitura e gravação em blocos nos `datafiles`, uso de `tablespaces`, operações de `undo` e `redo`, é de fundamental importância e ao mesmo tempo um diferencial para quem desenvolve. Ou seja, o manual de conceitos do banco de dados no qual se está trabalhando deveria ser lido e entendido por

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

ambos. Isso ajudaria e muito na montagem de melhores cenários visando à performance.

1.6 POR ONDE COMEÇAR?

Algo que é preciso ter em mente é que, para testarmos - ou melhor, simularmos - o nível de performance das aplicações em outras bases de dados, que não sejam no próprio ambiente produtivo (que é geralmente o que apresenta problemas ou que se quer monitorar), é necessário que tanto os aspectos físicos, quanto lógicos, desses ambientes sejam iguais.

Questões relacionadas à memória, processador, rede, discos, sistema operacional e banco de dados podem ser fatores cruciais para a melhora ou piora da performance de uma aplicação. Então, se não for possível realizar testes e simulações em ambientes de produção, o que é muito comum, é necessário termos atenção para que as questões de hardware e softwares sejam semelhantes.

Além disso, a base de dados para as simulações deve estar com a situação mais próxima da base de dados de produção, o mais próximo do cenário onde os problemas ocorrem. É comum as empresas terem clones de suas bases de produção para testes e desenvolvimento de novas funcionalidades, e muitas vezes até mais de um clone. Isso facilita muito as análises de performance em cenários isolados, como a demora na geração de um determinado relatório, ou de uma tela, que ao processar uma quantidade considerável de registros empacou por algum motivo. Dessa forma, é muito importante analisarmos o ambiente onde vamos atuar e fazer possíveis ajustes. Leve sempre isso em consideração.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Quando melhorar?

Você pensará em trabalhar a performance de uma aplicação ou banco de dados em duas ocasiões. A primeira é quando vier uma luz do céu, uma inspiração divina, fazendo com que você tenha uma vontade louca de otimizar. A segunda, e mais provável, é quando o usuário reclamar que algo está lento. Pode apostar que em 99,99% das vezes o motivo vai ser a segunda ocasião. Não porque você não se interessa pelo tema ou não tenha a menor ideia de como fazer este trabalho, mas pela correria do dia a dia, porque muitas vezes você não tem o cenário ideal para realizar esses testes quando está na fase de desenvolvimento, por prazos de entrega apertados ou qualquer outro motivo. Costumamos dizer em TI que estamos sempre em projeto, e que eles nunca terminam. Na verdade, existe até uma máxima que diz que nunca terminamos um projeto, e sim, o abandonamos por ora (risos).

Levando em consideração que na grande maioria das vezes seremos chamados para resolver um problema de performance que foi relatado por algum usuário, devemos ficar atentos às queixas em relação ao que está acontecendo, por exemplo, "o sistema está lento", ou sendo menos específico, "Tá tudo lento". Pois é, então não adianta só ler a descrição de um chamado ou se basear apenas nos relatos desesperados vindos através de e-mail ou chamadas telefônicas. Você terá que correr atrás dos fatos e envolver a vítima no processo. Não aceite apenas o "Tá tudo lento", peça informações, como tempo de demora, qual o volume de informações que ele está processando, o horário em que isso ocorre e se ocorre com frequência, os parâmetros que se está utilizando caso o problema seja em relatório ou uma pesquisa no sistema, por exemplo, ou seja, todo o tipo de informação que possa

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

ajudar nesse processo. Sem falar o mais importante, o nome do programa, tela ou relatório que está apresentando o problema (se não fica difícil, né?).

Depois disso, você já terá informações para iniciar seu trabalho de análise. No caso de programas que dependem da entrada de dados do usuário, como a execução de um relatório, você pode constatar que talvez ele esteja mandando buscar muitos dados no momento da execução, talvez até de forma desnecessária. Se for uma pesquisa em tela, um período de data mal selecionado para a pesquisa, por exemplo, pode gerar um volume muito grande de informação, causando uma demora inesperada. Às vezes, limitar certos tipos de critérios de pesquisa, mesmo que somente exigi-los, pode evitar certos problemas. Em últimos casos, você vai perceber que tais dados e critérios utilizados pelo usuário estão dentro do cenário de que ele precisa, e que realmente será necessário um trabalho de análise e até ajustes para melhorar a situação.

Com mais frequência do que você é capaz de imaginar agora, você descobrirá que a causa da baixa de performance em alguns casos decorre da existência de certos "hábitos" por parte dos usuários que operam a aplicação. Lembre-se de considerar isso! E indo de encontro a essa constatação, um outro ponto importante é identificar quando a melhoria da performance é realmente necessária. Talvez a solicitação de melhoria não possa ser atendida no tempo de resposta que o usuário precisa, pois, dependendo do que se está processando, a demora é inevitável. Nesses casos, a mudança no procedimento de como as coisas estão sendo feitas pode ajudar e até melhorar em muito para se ganhar tempo.

Em algumas situações também é necessário avaliar o escopo da

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

situação. Por exemplo, um relatório que é gerado apenas uma vez por mês, se ele está com o tempo de resposta em torno de três a quatro horas, talvez isso não seja tão problemático, se ele estiver processando dados de um mês inteiro ou até de meses anteriores. Diferente seria se todos os dias esse relatório tivesse que ser executado, mais de uma ou duas vezes por dia. Mas, como eu disse, tudo vai depender do contexto.

Identifique e defina juntamente com a pessoa ou área que está solicitando as melhorias qual o resultado mínimo que ela espera alcançar. Tenha clara a ideia do que se está tentando obter. Tente ser o mais preciso possível ao quantificar as metas em termos práticos. Por exemplo:

- Processar 10.000 pedidos a cada dia;
- Gerar relatório de 250.000 faturamentos de um dia para o outro no fim do mês;
- Gerar relatórios analíticos com informações detalhadas da empresa inteira, em um curto prazo de tempo, para tomadas rápidas de decisão.

Um fator relevante neste processo será conhecer o histórico de como o sistema foi concebido para saber se é possível implementar as melhorias solicitadas. Muitas vezes, projetos que nasceram com uma data de início e término determinadas acabam se estendendo pelo fato de muitas solicitações, antes não conhecidas, surgirem depois que a fase de modelagem e desenho de processo já tiverem sido concluídas. Essas solicitações podem impactar, de forma negativa, o funcionamento, comparado àquilo que havia sido idealizado nas outras fases. Fique atento a projetos que nunca acabam, embora, inicialmente, tivessem um escopo bem definido.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Se esse sistema for de uso de uma empresa, além do histórico técnico, tente resgatar um histórico de utilização e crescimento futuro. Um sistema processando 1.000 notas fiscais por dia pode vir a ter um comportamento muito diferente quando este número passar a resultar em 10 vezes mais. Isso vai ajudar a entender se a estrutura anteriormente montada vai corresponder às solicitações do presente e futuras.

Depois que você analisar os cenários e resolver os problemas, é muito importante acompanhar tais melhorias para comprovar a eficácia da solução. Não deixe apenas para ter o resultado no momento em que o usuário precisar reutilizá-la. Muito provavelmente, ele só vai testar novamente quando precisar utilizar tal programa e, quando isso acontecer, pode não vir apenas o descontentamento de não estar de acordo com o tempo de resposta esperado, mas também a pressão de que aquilo tem que estar funcionando o mais rápido possível.

Outro fator muito importante é quando você for colocar essa solução em ambiente de produção. Sabemos que, por mais que um ambiente de teste ou de homologação possa estar idêntico ao de produção, alguma coisa ruim pode ocorrer ao colocarmos algo a mais lá dentro. Dessa forma, eis duas dicas importantíssimas:

A primeira é, se na sua solução você está contemplando a criação, alteração ou até a exclusão de índices, garanta que esta ação não vá prejudicar outros programas que estão em produção. Faça uma varredura na base de dados antes, verificando possíveis programas que porventura possam estar usando o mesmo conjunto de atributos que faça parte de tal índice. Você não vai querer consertar algo e estragar outro, correto?

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Segundo, se a sua solução for constituída por uma série de melhorias distintas, tente verificar antes se uma ou outra, não as duas, já não resolve o problema. Às vezes, no ambiente de produção, onde geralmente há mais recursos como processamento e memória, uma simples atualização de estatística do banco de dados ou de determinadas tabelas específicas pode resolver o problema. O que eu quero dizer com isso? Aplique tais melhorias, uma de cada vez, e entre uma e outra vá observando se o último passo aplicado já não surte o resultado esperado. Você pode economizar tempo e espaço e resolver o problema sem muitos impactos.

Como pode ser visto, não há um tempo ou receita exatos de quando melhorar determinados aspectos em um sistema, tudo vai depender de avaliar cenários, necessidades e possibilidades. Um sistema pode ser delimitado em um conjunto de funcionalidades e objetivos, contudo, o crescimento exponencial dos dados, a estagnação dos recursos de hardware e rede, sistema operacional e banco de dados podem fazer com que o desempenho mude com o tempo, exigindo melhorias.

Para finalizar a parte teórica, vai aí uma dica de como conduzir o trabalho de monitoramento e ajuste da performance. Seja metódico em seus testes e crie padrões e requisitos para testar diferentes cenários e situações. O trabalho de analisar problemas de performance é quase um garimpo. Na maioria das vezes, você tem que fazer um trabalho de investigação para detectar onde exatamente vai atacar. Assim, você acaba tendo que realizar várias verificações até detectar onde e por que o problema acontece. Nem sempre o problema vai estar visível e de fácil acesso, e até você chegar no ponto onde atacar, muitos testes podem ter sido feitos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Cria índice, exclui índice, coloca `hint`, tira `hint`, substitui tabelas e assim por diante, se você não cuidar e anotar os cenários que já testou, tanto para detectar o problema quanto para testar uma solução, você pode correr o risco de andar em círculos.

Desta forma, monte e se baseie em cenários, mapeando situações onde os problemas acontecem, e quais soluções são melhores que outras. Isso vai economizar tempo e vai ajudar você a se organizar com mais rapidez e eficiência. Com a experiência, você vai aprimorando esses cenários e tornando este trabalho bem menos moroso.

Bem, era isso que eu queria comentar antes de iniciarmos o estudo das técnicas e ferramentas para este aprendizado. Você vai ver que esse trabalho vai lhe trazer um conhecimento de como as coisas funcionam quando se trata de processamento e recuperação de informações em um banco de dados. Embora estejamos falando sobre o banco de dados Oracle, em específico, você vai notar que haverá conceitos universais que poderão ser usados em qualquer banco de dados relacional existente. Eu espero muito que você goste dos assuntos aqui expostos, e que fique instigado a conhecer mais deste tema. Então, vamos para os próximos passos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 2

AJUSTE DA SQL

Podemos utilizar técnicas de Tuning em programas ou instruções SQL com o propósito de otimizar o acesso ou a manipulação dessas informações. Desta forma, o Tuning - também conhecido como *ajuste* ou *afinação*, mais precisamente ao que se refere à linguagem SQL, incluindo aqui também as instruções contidas dentro de programas escritos na linguagem PL/SQL - tem como objetivo refinar as aplicações, ou programas, como podemos comumente chamar, visando à melhoria de sua performance.

Podemos dizer que Tuning seria o ajuste de alguma coisa para que funcione melhor, de forma mais adequada e dentro das expectativas esperadas, construindo ou ajustando instruções SQL, dentro ou fora de programas PL/SQL, da forma mais adequada possível. Com isso, pode-se dizer que o principal objetivo do Tuning de SQL é buscar a escolha do modo mais eficiente para a execução de uma instrução SQL (`select` , `insert` , `update` ou `delete`), visando a uma performance satisfatória para uma aplicação que busque ou processe qualquer informação em um banco de dados.

Sistemas aparentemente bem projetados podem sofrer problemas com relação a desempenho em sua operação se as instruções SQL não forem construídas de forma adequada. Isso

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

acontece pelo fato de que o banco de dados, ao responder as requisições enviadas a ele, depende do SQL para que as informações sejam retornadas. Se tais instruções SQL não estiverem escritas de maneira otimizada, o desempenho do banco de dados pode ser influenciado. A performance deve ser construída com o intuito principal de minimizar o esforço na busca para localizar os dados.

Quando falamos em programas PL/SQL, também devemos ter cuidado com relação ao seu desempenho, pois dentro deles teremos muito provavelmente uma ou mais instruções SQL que, se não forem escritas da forma adequada, podem comprometer todo o funcionamento do programa. Além disso, teremos alguns recursos da própria linguagem PL/SQL que poderão ocasionar ou não a melhoria da performance, como o uso de cursores, variáveis `bind` e estrutura de dados em memória.

Logo, fica claro que a otimização da performance é um dos grandes desafios na construção de uma aplicação. A integração de todas as áreas envolvidas neste processo é muito importante, pois, como foi visto, a análise, o monitoramento e o ajuste da performance devem ser realizados desde o início do projeto. Uma aplicação que tenha um desempenho adequado e trabalhe dentro da performance esperada trará aos seus usuários benefícios importantes e satisfatórios.

Abordaremos as técnicas de monitoramento da performance e ajuste da SQL, tendo como ênfase principal a linguagem SQL. Mas antes disso, vamos falar do orquestrador, o responsável por analisar e processar todas as solicitações de manipulação e recuperação de dados no banco de dados – o otimizador.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

2.1 OTIMIZADOR DE CONSULTAS

Os Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDRs) possuem, em sua estrutura, um programa otimizador responsável por otimizar os comandos DML (*Data Manipulation Language*), a fim de que sejam executados seguindo planos de execução baseados em regras definidas e informações contidas em seu sistema. Ele é o componente encarregado de escolher um plano de execução, uma sequência exata de passos que permitirá obter o resultado da forma mais eficiente e rápida. Ou seja, quando o usuário executa uma instrução SQL (`select` , `update` , `delete` ou `insert`), o otimizador do banco de dados analisa essas instruções e as ajusta, caso seja preciso, escolhendo qual o melhor caminho para a execução. Este trabalho é realizado internamente pelo otimizador, sem que seja explícito ao usuário. O otimizador utiliza de métricas, cálculos estatísticos e até aspectos de inteligência artificial (AI) que o ajudam neste trabalho.

Essa otimização realizada de forma automática traz benefícios para quem não tem o conhecimento (ou tempo) para ajustar suas instruções SQL. Todavia, esse não é o único benefício que a otimização automática proporciona, e sim, o fato de que há uma real possibilidade de o otimizador se sair melhor do que qualquer um neste trabalho, mediante as seguintes razões:

- Um bom otimizador possui uma série de informações disponíveis. Estas informações, que muitas vezes quem está escrevendo instruções SQL não possui, são relacionadas às estatísticas empregadas aos objetos do banco de dados;
- Outro ponto é a mudança que esses dados estatísticos sofrem com o tempo. Nesse caso, uma reotimização

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

poderia ser necessária. Em uma otimização automática isso seria resolvido com um reprocessamento da requisição. Quanto à forma manual ou em um sistema não relacional, a reotimização poderia implicar em reescrever o programa;

- O otimizador possui mais paciência que o ser humano. Ele é capaz de testar centenas de cenários, procurando o melhor caminho para a execução das requisições. Quanto a quem está escrevendo um programa, seria pouco provável tal disposição ou tempo para testar esses inúmeros cenários.

O otimizador realiza uma série de tarefas com o intuito de que as instruções SQL sejam executadas de forma adequada. Essas tarefas resumem-se em avaliar expressões e condições, modificando-as caso seja preciso, decidindo qual melhor acesso aos dados armazenados e, em caso da utilização de várias tabelas, quais ligações (`joins`) serão necessárias. O processamento da instrução envolve quatro grandes estágios os quais demonstram como podem ser explicadas e compreendidas as estratégias e técnicas de otimização:

Estágio 01 - Converter a instrução para algum formato interno: converte a instrução original em um formato mais adequado para o processamento da máquina, elimina considerações puramente externas como detalhes da sintaxe da linguagem SQL, prepara o caminho para os estágios subsequentes;

Estágio 02 - Converter para a forma canônica: aqui uma série de otimizações consideradas adequadas é realizada, independente dos valores de dados reais e dos caminhos de acesso físico que possam estar armazenados no banco de dados;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Estágio 03 - Escolher procedimentos candidatos de baixo nível: depois de convertida a consulta, chegou o momento de decidir como executar a consulta transformada. Neste ponto são consultados a existência de índices ou outros caminhos de acesso, distribuição de valores, `cluster` físico de dados armazenados etc.;

Estágio 04 - Gerar planos de execução e escolher o mais "barato": o último estágio envolve a montagem de um conjunto de planos de execução que serão avaliados. A escolha será pelo melhor, quer dizer, o mais econômico entre os planos. Isso é feito utilizando-se fórmulas de custo e os conhecimentos das estatísticas dos objetos do banco de dados. Cada plano é constituído por um conjunto de procedimentos que serão responsáveis pela implementação das operações.

A imagem a seguir demonstra esses estágios dentro de um fluxo que vai desde a execução de uma instrução SQL, até o retorno do seu resultado.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

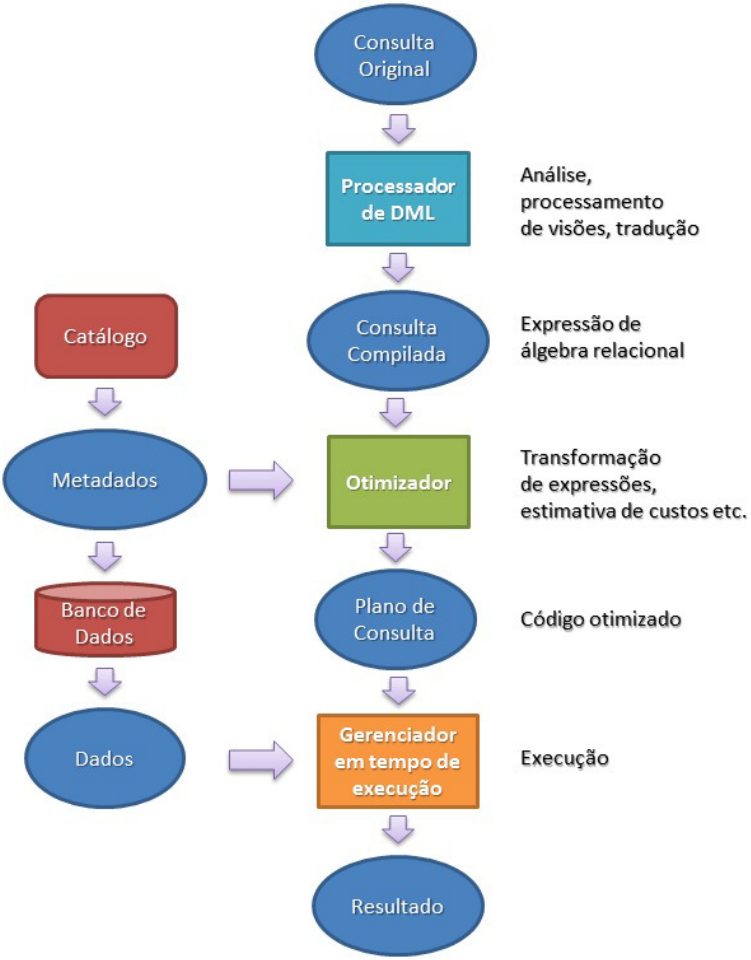

Figura 2.1: Visão geral do processamento de consultas. Adaptado de: Oracle Database Online Documentation 12c, Release 1 (12.1) / Database SQL Tuning Guide https://docs.oracle.com/database/121/TGSQL/tgsql_sqlproc.htm#TGSQL175

O otimizador tem papel importante na otimização das instruções SQL. Ele possui a capacidade de ler as expressões inseridas e encaminhá-las de forma a serem executadas, a seu ver,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

da forma mais adequada possível. Entretanto, pode haver momentos em que a escolha do otimizador não será tão eficaz a ponto de suprir as expectativas, sendo preciso introduzir outros conceitos que o auxiliem nesse sentido. Não porque ele é falho, mas sim, por nossas solicitações serem até confusas para ele. Isso tem a ver até com a forma como os sistemas foram estruturados e com quão complexas são as instruções SQL que fazem parte dele. Haverá situações em que, apesar dos grandes esforços do otimizador, teremos que ajudá-lo a determinar o caminho mais eficiente para a obtenção ou manipulação dos dados. Diante disso, é conveniente utilizar muito bem os benefícios do otimizador, mas também empregar de forma adequada os conceitos de otimização, buscando melhorar o cenário para atender todas as expectativas esperadas com relação à performance.

2.2 O OTIMIZADOR DO ORACLE

Como foi dito anteriormente, existem várias formas de se executar instruções SQL. O Oracle, através do seu otimizador, chamado de Optimizer, escolhe a forma mais eficaz, o plano mais adequado para executar essas instruções. Então seu objetivo é, através de um conjunto de informações, traçar a melhor rota e o melhor plano de execução. O otimizador da Oracle trabalha baseando-se em planos de execução, que serão mencionados durante todo o livro. Para montar esse plano, ele poderá utilizar:

- Informações do dicionário de dados referente aos objetos do banco de dados;
- Estatísticas dos objetos que fazem parte da instrução SQL;
- `hints` que possam ter sido mencionados de forma explícita na instrução SQL;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Cláusulas `where` , agrupamentos, ordenação e índices.

Antes de nos aprofundarmos nas características do otimizador, é preciso entender os passos de como uma instrução SQL é executada. Quando se executa uma instrução SQL em um banco de dados, três etapas são executadas, sendo elas o *Parse*, *Execute* e *Fetch*. Em cada uma, o Oracle define tarefas que devem ser seguidas para que o resultado seja obtido.

Etapa: `Parse`

- A primeira tarefa desta etapa é verificar a sintaxe da instrução e validar a semântica;
- Depois é a vez de pesquisar se essa instrução já foi executada anteriormente e encontra-se na memória. Se existir, o plano de execução já estará traçado, tendo em vista que a instrução já foi analisada e executada antes. Então será passado direto para a fase `Execute` , sem repetir as outras ações;
- Caso não seja encontrado na memória, o Oracle verifica no dicionário de dados a existência das tabelas e colunas envolvidas na instrução, bem como as permissões de acesso a esses objetos;
- Após a pesquisa, o otimizador poderá traçar o caminho de acesso para cada tabela presente na instrução, montando um plano de execução para obter e/ou atualizar os dados referentes à requisição feita através desta instrução;
- Tendo o plano montado, o Oracle o inclui em memória para que a execução seja realizada e para que este plano seja utilizado por outra execução idêntica à atual, ou seja, se a mesma instrução for executada novamente.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Etapa: `Execute`

- A primeira tarefa desta etapa é aplicar o plano de execução. Com isso, a leitura dos dados será feita de forma física ou lógica, dependendo da sua localização em disco ou em memória. Se estiver em memória, não necessitará fazer qualquer acesso a disco. Caso contrário, a leitura é feita, e o retorno é armazenado na área de memória do banco de dados;
- Já com os dados em memória, será realizada a análise das restrições que consistem em verificar as regras de integridade (*constraints*), por exemplo. Esta tarefa é aplicada a processos de atualização;
- Como a tarefa anterior, esta tarefa também só é aplicada em processos de atualização, sendo que é preciso bloquear as linhas de dados (*locked*) e alocar o segmento de *rollback* para que a atualização das linhas seja realizada.

Etapa: `Fetch`

- Se for o caso em que a instrução SQL executada for um `select` , o Oracle retorna o resultado.

Através dessas etapas, o Optimizer avalia as expressões e condições, transforma uma instrução em uma instrução equivalente aos seus critérios, decide como será o acesso aos dados armazenados e quais conexões serão necessárias para isso.

O otimizador pode trabalhar de acordo com duas premissas, baseando-se no que ele chama de RBO (Otimização Baseada em Regra) ou em CBO (Otimização Baseada em Custo). Ele pode trabalhar ainda em diferentes modos, os quais são definidos

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

através de parâmetros que são configurados no banco como padrão, ou para sessões específicas. Pense que, em cada modo, o otimizador trabalhará de uma forma diferente, utilizando estratégias diferentes de otimização. Os possíveis modos para o otimizador estão listados a seguir.

- `CHOOSE` : usa a otimização baseada em custo (visto mais adiante) para todas as tabelas analisadas. Seu papel é estimar o tempo de execução para diferentes métodos de execução e escolher o mais rápido. Esse é um bom modo para sistemas bem montados e bem ajustados. Pode ser visto como `cost` em algumas bibliografias;
- `RULE` : sempre usa a otimização baseada em regras. Trata-se de um conjunto de regras a respeito de como Oracle processa instruções;
- `FIRST_ROWS` : retorna o primeiro registro de um resultado mais rapidamente (em geral, força o uso de índice). Isso é bom para sistemas não ajustados, que processam muitas transações isoladas;
- `FIRST_ROWS_(1|10|100|1000)` : retorna as primeiras *n* linhas mais rapidamente. Isso é bom para aplicações que exibem rotineiramente resultados parciais aos usuários, como dados de paginação em uma aplicação Web;
- `ALL_ROWS` : retorna todas as linhas mais rapidamente (em geral, força a supressão do índice). Isso é bom para sistemas em lote não ajustados e com grande volume (normalmente, não usado).

Neste livro, veremos os modos `CHOOSE` e `RULE` do otimizador, baseando-se em custos e em regras respectivamente.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

A forma como o otimizador vai trabalhar pode ser definida dos seguintes modos:

Modo em nível de instância:

Esta configuração é geralmente efetuada pelo DBA e, nela, o seguinte parâmetro de banco de dados `optimizer_mode` é configuração com os possíveis valores: `CHOOSE` , `RULE` , `FIRST_ROWS` , `ALL_ROWS` . Veja a sintaxe a seguir:

```
OPTIMIZER_MODE = {CHOOSE|RULE|FIRST_ROWS|ALL_ROWS}
```

Modo em nível de sessão:

Já a configuração em nível de sessão se limita apenas à sessão em que estamos conectados. Utilizamos para isso o comando `alter table` . Os mesmos modos utilizados para configurar no nível de instância também estão em nível de sessão. Veja a sintaxe a seguir:

```
SQL> alter session set optimizer_mode = {choose|rule|first_rows|a
ll_rows}
```

Modo em nível de comando:

Outra forma de influenciar o modo de otimização é através de `hints` , diretamente no comando `select` , por exemplo. Nesse caso, nem a instância e nem a sessão são afetadas, apenas o comando em si é afetado pelo modo. Veja o exemplo.

```
select /*+ RULE */ employee_id, first_name, department_id
from employees
where department_id = 10
```

Para visualizar com qual modo de otimizador o banco de dados ou sessão está configurado, use a seguinte instrução:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select name, value from v$parameter where name = 'optimizer_
mode';
```

NOTA: O modo `CHOOSE` equivale ao modo `ALL_ROWS` quando existirem estatísticas. Em algumas documentações, você pode encontrar ainda o nome de parâmetro `optimizer_goal`. Ele funciona como um sinônimo para `optimizer_mode` e está disponível por questões de compatibilidade para alteração do modo em nível de sessão com `alter session`. Contudo, em versões mais novas do banco, uma mensagem de erro é gerada quando tentamos utilizá-lo, nos informando de que ele está obsoleto.

Vamos conhecer agora um pouco das características da otimização baseada em custos.

2.3 O MODO BASEADO EM CUSTOS (CBO)

O modo padrão do otimizador do Oracle, pelo menos para as versões mais novas do banco (partindo do Oracle 9i), é o `choose` (baseado em custos). Esse modo foi introduzido na versão 7 do banco de dados Oracle e seu foco principal é a escolha do plano de execução com base em cálculos de custo em cima de dados estatísticos como número de leituras lógicas e físicas, utilização de CPU e tráfego de rede.

Entendendo estatísticas

As estatísticas do otimizador são uma coleção de dados que

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

descrevem mais detalhes sobre o banco de dados e seus objetos. São usadas pelo otimizador para escolher o melhor plano de execução para cada instrução SQL. Estatísticas do otimizador incluem:

- Estatísticas de tabela:
  - Número de linhas (registros de uma tabela);
  - Número de blocos de dados;
  - Comprimento médio das linhas.
- Estatísticas de colunas:
  - Número de valores distintos na coluna (NDV);
  - Número de colunas nulas;
  - Distribuição dos dados (histogramas).
- Estatísticas dos índices:
  - Número de *Leaf Blocks*;
  - Níveis;
  - Fator de agrupamento (*Clustering*).
- Estatísticas de sistema:
  - Desempenho e utilização de I/O;
  - Desempenho e utilização de CPU.

Trabalhando nesse modo, o otimizador levará em conta a natureza dos dados, considerando o número de registros da tabela e a quantidade de colunas, ou seja, informações dos dados, dos objetos e de estatísticas geradas e armazenadas no dicionário de dados do banco, para calcular o custo de cada plano de execução, escolhendo o mais apropriado para executar a instrução SQL. Além disso, ao contrário do modo baseado em regra, o modo baseado em custo é continuamente melhorado pela Oracle a cada nova versão.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Devemos ter atenção, entretanto, ao fato de que o otimizador utilizará o modo baseado em regras enquanto as tabelas não forem analisadas, mesmo que o parâmetro relacionado ao modo, esteja configurado como `CHOOSE` (baseado em custo). Contudo, se pelo menos um dos objetos referenciados na instrução SQL estiver com as estatísticas geradas, então ele considerará o modo baseado em custo. Nesse caso, para os objetos cujas estatísticas não existirem, o otimizador se utilizará de valores internos predefinidos (para esses casos, veremos mais adiante no modo baseado em regras).

O otimizador usa as estatísticas para calcular os custos em busca de melhores resultados, pois elas melhoram o grau de precisão na escolha do plano de execução. Na fase de otimização, o otimizador utiliza-se das seguintes métricas: seletividade, cardinalidade e custo. Antes de vermos como gerar estatísticas para o otimizador, vamos entender um pouco sobre cada uma dessas métricas.

Seletividade

A seletividade representa uma fração de linhas de um conjunto de linhas. O conjunto de linhas pode ser referente a uma tabela, uma *view*, ou o resultado de um *join* ou de um operador `group by` . A seletividade está associada a um predicado de consulta, como `last_name = 'SMITH'` , ou uma combinação de predicados, tais como `last_name = 'SMITH' and job_type = 'CLERK'` . Um predicado funciona como um filtro que restringe um determinado número de linhas a partir de um conjunto de várias linhas. Portanto, a seletividade de um predicado indica quantas linhas de um determinado conjunto de linhas passam no teste do predicado.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

A seletividade é medida por uma faixa de valores que varia de 0.0 a 1.0. Uma seletividade de 0.0 significa que não há linhas a serem selecionadas a partir de um conjunto de linhas, e uma seletividade de 1.0 significa que todas as linhas do conjunto serão selecionadas.

Cardinalidade

Cardinalidade representa o número de linhas em um conjunto de linhas, podendo ser uma tabela, uma `view` , ou o resultado de um `join` ou de um operador `group by` . A cardinalidade mais básica (*Base cardinality*) de uma fonte de dados, por exemplo, uma tabela, seria a própria quantidade de linhas que ela possui. Contudo, existem outros quatro tipos:

- *Effective cardinality*: refere-se à quantidade de registros realmente selecionados.
- *Join cardinality*: refere-se à quantidade de registros gerados a partir do relacionamento entre duas fontes de dados (tabelas).
- *Distinct cardinality*: refere-se à quantidade de valores distintos. A cardinalidade distinta de um conjunto de linhas é calculada com base nos dados da coluna.
- *Group cardinality*: refere-se à quantidade de registros após aplicada à cláusula `group by` . A cardinalidade de grupo depende da cardinalidade distinta de cada uma das colunas do agrupamento e do número de linhas do conjunto de linhas.

Desses tipos de cardinalidade, ouve-se falar bastante do *Join cardinality*, principalmente quando estamos descrevendo o MER

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

(Modelo Entidade Relacionamento) ou desenhando o DER (Diagrama Entidade Relacionamento). É quando definimos a forma de relacionamento entre as entidades existentes em nosso modelo que aplicamos este tipo de cardinalidade, classificando-os como: um para um (1..1), um para muitos (1..N) ou muitos para muitos (N..N).

Um exemplo clássico seria a cardinalidade entre duas tabelas, EMPREGADOS e DEPARTAMENTOS, onde temos, de um lado, uma cardinalidade 1..N no caso de departamentos para empregados (lê-se 1 departamento para muitos empregados), e no inverso, N..1 (lê-se muitos empregados para 1 departamento) no caso de empregados para departamentos. Vale lembrar que podemos ainda definir as cardinalidades mínima e máxima para cada um destes dois casos. No entanto, na maioria das vezes, apenas a cardinalidade máxima é a mais relevante. As figuras a seguir exemplificam o uso da notação de cardinalidade nos diagramas.

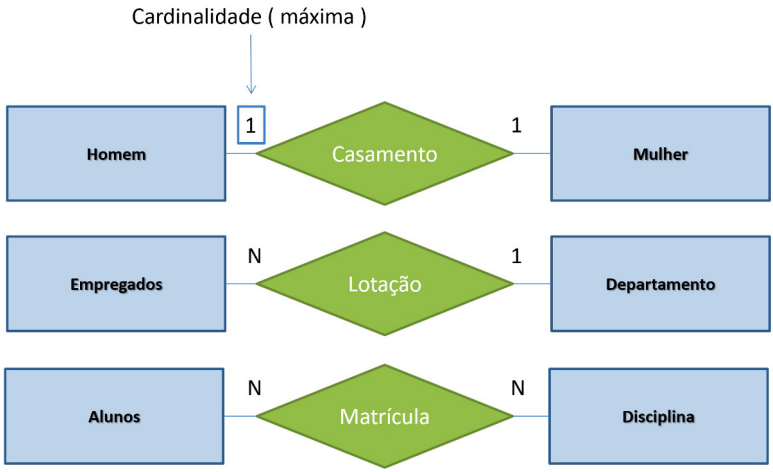

Figura 2.2: Visão da cardinalidade máxima.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

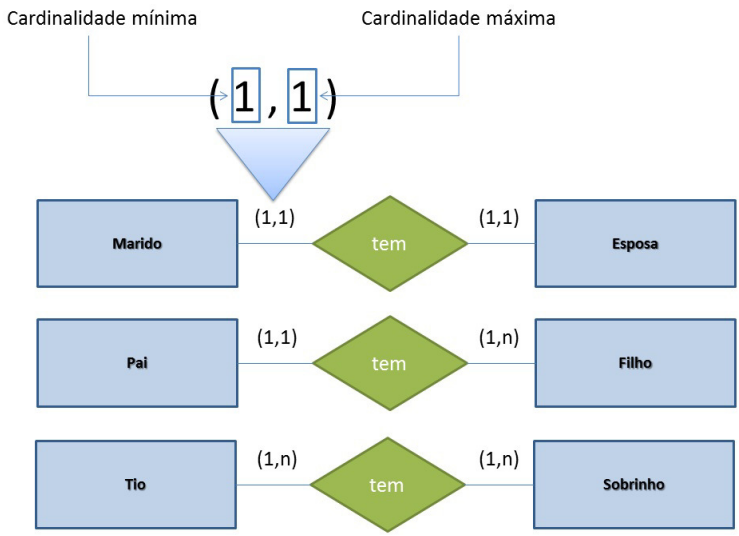

Figura 2.3: Visão das cardinalidades mínima e máxima.

Custo

O custo é o resultado do cálculo dos recursos utilizados. A CBO (Otimização Baseado em Custos) usa o I/O de disco, CPU e memória como recursos. Assim, o custo utilizado pela CBO representa uma estimativa do número de I/Os de disco, quantidade de CPU e memória utilizada para executar uma operação, que pode ser uma varredura da tabela, o acesso a linhas de uma tabela usando um índice, junção de duas tabelas, ou ordenação de um conjunto de linhas. O custo de um plano de execução é o número de recursos que se espera que sejam incorridos quando a consulta é executada e o resultado é produzido.

O caminho de acesso determina o número de recursos necessários para obter os dados de uma tabela. Este caminho pode

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

ser uma varredura de tabela (`TABLE SCAN`), uma varredura rápida completa do índice (`FAST FULL INDEX SCAN`), ou uma varredura simples do índice (`INDEX SCAN`). Durante a varredura de tabela ou uma rápida varredura completa de um índice, vários blocos são lidos a partir do disco em uma única operação de I/O. Portanto, o custo de uma varredura de tabela ou uma rápida varredura completa de índice depende do número de blocos a serem varridos e a contagem de multiblocos a serem lidos. O custo de uma varredura de índice depende dos níveis da estrutura da árvore do índice (`B-tree`), do número de `Leaf Blocks` a ser varrido, bem como o número de linhas a serem buscadas usando o `rowid` nas chaves de índice. O custo de buscar as linhas usando `rowids` dependerá, ainda, do fator de agrupamento do índice.

O custo de `join` representa a combinação dos custos de acesso individuais de dois conjuntos de linhas a serem unidas (por exemplo, duas tabelas). Em um `join` , um conjunto de linhas é chamado de interno, e o outro é chamado de externo. É muito comum vê-los aparecendo no plano de execução.

Em um `NESTED LOOP JOIN` (junções aninhadas), por exemplo, para cada linha do conjunto de linhas externo, o conjunto de linhas interno é acessado para encontrar todas as linhas correspondentes que satisfaçam a junção entre estes dois conjuntos. Portanto, em um `NESTED LOOP` (loops aninhados), o conjunto de linhas interno é acessado tantas vezes quanto o número de linhas no conjunto externo.

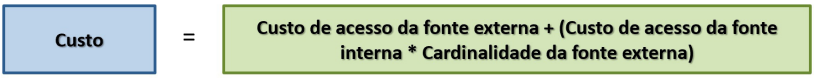

Figura 2.4: Custo em nested loop.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Em um `sort merge join`, os dois conjuntos de linhas são ordenados pelas chaves de junção, caso elas já não estejam na ordem da chave.

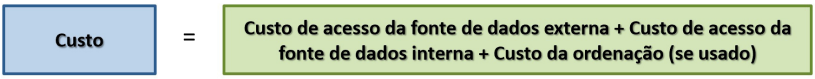

Figura 2.5: Custo em sort merge join.

Em uma `HASH JOIN`, o conjunto de linhas da fonte interna é colocado em uma `hash table` (tabela em memória) usando a chave de `join`. Em seguida, cada linha do conjunto de linhas externo, também é colocada na `hash table`, onde as linhas correspondam à junção. Se o conjunto de linhas interno for muito grande, então apenas uma parte dele é colocado na memória da `hash table`. Esta parte é chamada de `hash partition`.

Cada linha do conjunto de linhas externo da `hash table` é confrontada com as linhas correspondentes na `hash partition`. A próxima parte do conjunto de linhas interno é colocada na `hash table`, seguida por uma verificação junto ao conjunto de linhas da fonte externa. Esse processo é repetido até que todos os `hash partition` do conjunto de linhas interno estejam esgotados.

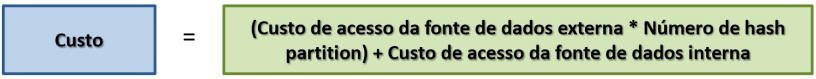

Figura 2.6: Composição do valor do custo.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

NOTA: Não se preocupe em saber agora o que é um `NESTED LOOP JOIN` ou um `HASH JOIN`. Mais adiante teremos um capítulo específico sobre esses métodos de junção de tabelas.

O comando analyze

É através do comando `analyze` que geramos as estatísticas para o modo baseado em custo. É preciso que as tabelas usadas nas consultas estejam analisadas e suas estatísticas estejam armazenadas nas tabelas do dicionário de dados do Oracle. Essas estatísticas podem ser consultadas através das tabelas `user_tables`, `user_tab_columns`, `user_indexes`, `user_clusters`, `dba_part_col_statistics`, entre outras. O comando `analyze` gera ou elimina estatísticas para que, em modo custo, as instruções SQL sejam executadas dentro de um plano de execução mais adequado, visando a uma melhor performance. Esse recurso é usado para:

- Aplicar performance Tuning (Ajuste de Desempenho);
- Validar a estrutura de Objetos;
- Gerenciar o espaço utilizado pelo objeto.

O uso do comando `analyze` não é mais recomendado, porque um novo pacote de recursos, chamado `dbms_stats`, está disponível e deve ser utilizado para este fim. Este novo pacote está disponível para uso nas versões mais novas do banco de dados Oracle (a partir da 9i). Contudo, por questões de compatibilidade, a Oracle mantém o `analyze` em versões atuais. Veja a seguir as vantagens e desvantagens entre `dbms_stats` e `analyze`:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Você pode importar e exportar um conjunto de estatísticas com `dbms_stats` ;
- É mais fácil para automatizar com `dbms_stats` (pois ele funciona de forma procedural, enquanto `analize` é apenas um comando);
- `dbms_stats` pode analisar tabelas externas, `analize` não.
- `dbms_stats` gera estatísticas apenas para a otimização baseada no custo; não gera outras estatísticas. Por exemplo, as estatísticas de tabela geradas pelo `dbms_stats` incluem o número de linhas, número de blocos contendo dados e comprimento médio de linha, mas não o número de linhas encadeadas, espaço livre, média ou o número de blocos de dados não utilizados;
- `analize` calcula as estatísticas globais de tabelas e índices em vez de recolhê-las diretamente. Isso pode conduzir a imprecisões para algumas estatísticas, tais como o número de valores distintos.
- `dbms_stats` pode ser executado em paralelo;
- No futuro, `analize` não coletará estatísticas necessárias para o otimizador baseado em custo de forma eficiente.

O `dbms_stats` é o método indicado pela Oracle para a geração de estatísticas a partir de agora.

Para você ter conhecimento do comando `analize` , demonstrarei alguns exemplos do seu funcionamento. Mais adiante, teremos um capítulo que abordará a geração de estatísticas do pacote `dbms_stats` .

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Sintaxe

```
analyze { index index_name | table table_name | cluster
        cluster_name }
    [ partition ( partition_name ) ]
    [ [ compute | estimate | delete ] statistic ]
    [ validate structure [ cascade ] ]
    [ list chained rows into table_name ]
```

Em que:

- `compute` : examina todas as linhas do objeto.
- `estimate` : examina parte das linhas do objeto.
- `delete` : remove as estatísticas do objeto.

Nota: Ao utilizar a cláusula `estimate` , é gerada apenas uma estimativa aproximada, utilizando um percentual ou número de registros da tabela como amostragem. Quanto ao uso da cláusula `compute` , duas estatísticas são coletadas e armazenadas no dicionário de dados. Ambas as estatísticas, tanto a calculada, quanto a estimada, são utilizadas pelo otimizador do banco de dados Oracle para escolher o plano de execução mais adequado para as instruções SQL. Na cláusula `estimate` , se você não especificar a quantidade de dados ou percentual analisado, será utilizada como padrão uma quantidade de linhas igual a 1064. Caso a tabela analisada tenha alguns milhares de registros, talvez seja interessante informar um número ou percentual maior. No caso de informar uma estimativa que ultrapasse a metade dos dados, poderá ser calculada a estatística para todas as linhas.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Exemplos

```
SQL> analyze table employees estimate statistics;

Tabela analisada.


SQL> analyze table employees estimate statistics sample 100 rows;

Tabela analisada.


SQL> analyze table employees estimate statistics sample 15 percen
t;

Tabela analisada.


SQL> analyze table departments compute statistics;

Tabela analisada.


SQL> analyze table employees delete statistics;

Tabela analisada.

SQL>
```

As informações geradas pelo `analyze` podem estar classificadas em três níveis:

- **Nível de Tabela**: número de linhas, número de blocos e blocos nunca usados, média do espaço livre por bloco e tamanho médio da linha;
- **Nível de Coluna**: número de valores distintos por coluna;
- **Nível de Índice**: nível do índice, número de *Leaf Blocks*, número de chaves distintas, média do número de *Leaf Blocks* por chave, média do número de bloco de dados por chave, menor valor de chave e maior valor de chave.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

NOTA: As informações contidas nas linhas das tabelas são gravadas nos blocos. A informação referente aos Blocos significa que estamos falando de um ou mais blocos do sistema operacional. Já os *Leaf Blocks* são os blocos de índice. Eles são os blocos inferiores em uma árvore de índice (`B-tree`). É neste bloco que constam os dados com seus valores indexados por `rowid` , que será a base para localizar as linhas correspondentes numa tabela, por exemplo. Além dos *Leaf Blocks*, temos os *Branch Blocks*, que são os blocos superiores na árvore `B-tree` . Neles estão contidos dados de índice que apontam para um nível abaixo de blocos de índice. Veja a imagem a seguir:

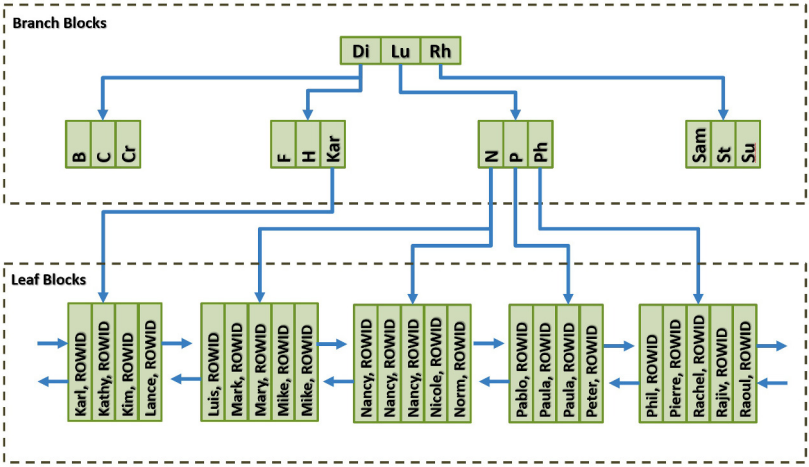

Figura 2.7: Adaptado de: https://ittutorial.org/index-types-and-index-create-in-oracle-database/

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

No modo baseado em custo, devemos ter atenção pois, à medida que os dados em uma tabela mudarem, ela precisará ser regularmente reanalisada, ou seja, à medida que o banco de dados for crescendo ou diminuindo, as estatísticas deverão ser atualizadas.

> **NOTA:** Geralmente, quem gera as estatísticas é o DBA.

Vamos visualizar o plano de execução de uma instrução SQL na tabela `departments`, considerando que temos estatísticas geradas para esta tabela.

```
SQL> analyze table departments compute statistics;

Tabela analisada.
```

Instrução SQL e Plano de Execução:

```
SQL> select * from departments

Plano de Execução
----------------------------------------------------------
  0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=27 Bytes=486)
  1    0   TABLE ACCESS (FULL) OF 'DEPARTMENTS' (TABLE) (Cost=2 Card=27 Bytes=486)
```

Figura 2.8: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap02img01code.txt

As informações de Cost, Card(inalidade) e Bytes são inerentes ao modo baseado em custo utilizado pelo otimizador. Atenção para o valor `cost`: quanto menor este valor, mais chance de uma execução mais rápida.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Vimos como customização baseada em custo funciona. Vamos ver agora as características e detalhes da otimização baseada em regras.

2.4 O MODO BASEADO EM REGRAS (RBO)

O modo `RULE` (baseado em regras) não é mais recomendado pela Oracle nas versões mais atuais do banco de dados, contudo, por questões de compatibilidade, ele ainda permanece lá. Mesmo assim, é estritamente aconselhável o uso da otimização baseada em custo.

Vamos falar sobre ele pois, além do conhecimento histórico, já que antes, no Oracle 6 e versões anteriores, este era o único modo existente, ainda pode acontecer de você ter que lidar com o modo baseado em regras para resolver algum problema específico de desempenho, ou de você ter a necessidade de trabalhar em um ambiente com versões antigas do Oracle.

No modo baseado em regras, o otimizador não leva em conta a natureza do dado, não considerando o número de linhas da tabela, por exemplo. Ele não se utiliza de estatísticas e não calcula custos, em vez disso, ele leva em consideração a sintaxe de instruções SQL, principalmente, no que está nas cláusulas `from` e `where`. Isso quer dizer que, modificando a sintaxe de uma instrução SQL, podemos influenciar na performance de sua execução.

Outro fator que influencia, e muito, no modo baseado em regras é a existência de índices nas tabelas. Mesmo às vezes não sendo uma boa ideia a utilização de um determinado índice, dependendo da situação, o otimizador acaba utilizando-o. Índices

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

são bem-vindos quando estamos trabalhando com resultados onde a seletividade é baixa, contudo, se estivermos esperando um retorno maior que 20% a 30% dos registros da tabela, por exemplo, optar pelo uso do índice pode ser um problema. Neste caso, um acesso completo a tabela poderia ser mais interessante.

Influenciando o otimizador através da sintaxe

O otimizador baseado em regras leva em consideração a sintaxe da instrução SQL. Dessa forma, questões como a ordem das tabelas na cláusula `from`, a ordem e como estão escritas as cláusulas `where` poderão influenciar na forma como o otimizador vai iniciar a leitura da sua instrução SQL e, portanto, a busca por caminhos de acesso para a montagem do plano de execução.

No caso da cláusula `from`, é indicado que você sempre coloque a tabela que contém mais restrições no final, seja pela quantidade de dados ou pelas cláusulas condicionais que venham a fazer parte da instrução SQL, principalmente as que restringirão a quantidade de dados. O motivo disso é que o otimizador inicia a leitura da cláusula `from` da direita para a esquerda. Dessa forma, ler tabelas com mais restrições primeiro evitaria de levar registros não condizentes para a próxima etapa de junção (no caso de `loops` aninhados) com outras tabelas da instrução, por exemplo.

No caso da cláusula `where`, podemos seguir a mesma lógica. Sempre é recomendando colocar as cláusulas mais restritivas por último na descrição da instrução SQL, pois o otimizador iniciará sua leitura pela última cláusula de restrição informada. Via de regra, sempre que possível, procure colocar as restrições de cláusula `where` na mesma ordem em que as tabelas estão na

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

cláusula `from` . Isso poderá ajudar o otimizador a poupar trabalho e escolher o caminho mais adequado.

Outro ponto a ser observado é que o otimizador quase sempre pode optar por utilizar índices quando estiverem disponíveis. Nem sempre a escolha pelo índice vai ser uma boa escolha. Nesses casos, para evitar que ele não escolhesse tais índices, seria necessária alguma ação, como excluí-los.

> **NOTA:** Vamos aprender mais tarde que para um índice não ser usado como caminho de acesso, seria a utilização do `hint FULL` . Contudo, um `hint` (exceto a `hint RULE`) tenta invocar o otimizador baseado em custo (CBO) e talvez não seja essa a intenção.

Excluir índices poderia ser uma tarefa incomum e não muito adequada, tendo em vista que podemos não conhecer o impacto que esta exclusão ocasionará nas demais aplicações, ou por ele ser produto de uma `primary key` ou de alguma regra de integridade. Assim, podemos utilizar de um subterfúgio para invalidar seu uso pelo otimizador. Um índice, para ser usado pelo otimizador, precisa que sua coluna esteja informada na instrução SQL de forma que não esteja sofrendo qualquer alteração no seu dado. Se estivermos aplicando uma função a uma coluna indexada, por exemplo a função `to_number(coluna_indexada)` , ou até mesmo concatenando um caractere junto a coluna como `||''` , o índice associado a esta coluna é desconsiderado pelo otimizador, invalidando seu uso como caminho de acesso.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

No entanto, vale ressaltar que essas práticas não são recomendadas em versões onde o otimizador trabalhe em modo de custo ou em que recursos mais avançados existam. É recomendado ter estatísticas atualizadas e utilizar o modo baseado em custo para se obter maiores ganhos de performance.

Ordem de preferência: o rank do RBO

No modo baseado em regras, o otimizador escolhe uma rota de acesso baseada nos caminhos (regras) disponíveis e na pontuação de tais caminhos que são predefinidos. Ele utiliza-se da chamada ordem de preferência para acessar os dados, da mais rápida para a mais lenta, sendo o acesso por `rowid` (rank 1) o mais rápido, e o acesso `FULL TABLE SCAN` (rank 15) o mais lento, conforme demonstrado a seguir:

Rank	Caminho de acesso
1	Acesso de uma única linha via `rowid`
2	Acesso de uma única linha via `cluster join`
3	Acesso de uma única linha via `hash cluster key` com `unique` ou `primary key`
4	Acesso de uma única linha via índice `unique` ou `primary key`
5	`cluster join`
6	`hash cluster key`
7	`cluster key` Indexada
8	Índice composto
9	Índices de uma coluna
10	Intervalo definido de pesquisa, em colunas indexadas
11	Intervalo indefinido de pesquisa, em colunas indexadas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

12	`sort merge join`
13	`max` ou `min` de colunas indexadas
14	`order by` em colunas indexadas
15	`full table scan`

Ao executar uma instrução SQL, o otimizador analisa e verifica sua sintaxe, isto é, que tabelas e cláusulas `where` você utilizou. Então, ele tenta ranquear a instrução com base nestes 15 caminhos de acesso. A seguir, temos alguns exemplos.

Antes de iniciar, foi habilitado o recurso de *autotrace* na sessão do `SQL*Plus` para podermos analisar qual o caminho que o otimizador escolherá para executar a instrução SQL. Também é preciso garantir que o modo de otimização esteja baseado por regras (`RULE`). Mais adiante veremos todos estes recursos, inclusive o *autotrace*.

```
SQL> set autotrace on

SQL> alter session set optimizer_mode = 'RULE'
```

Agora vamos executar a seguinte instrução SQL:

```
SQL> select * from departments

Plano de Execução
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (FULL) OF 'DEPARTMENTS' (TABLE)
```

Figura 2.9: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap02img02code.txt

Como não foi utilizada a cláusula `where` , o resultado foi o

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

retorno de todos os registros da tabela. O otimizador, como não havia muito a avaliar, optou por um acesso via `TABLE ACCESS (FULL)`, escolhendo o caminho `FULL TABLE SCAN`, rank 15, na tabela mostrada anteriormente.

Se executarmos a instrução SQL colocando algumas cláusulas `where`, veremos que o otimizador se comportará diferente.

```
SQL> select * from departments where department_id = 40;

Plano de Execução
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'DEPARTMENTS' (TABLE)
   2    1     INDEX (UNIQUE SCAN) OF 'DEPT_ID_PK' (INDEX (UNIQUE))
```

Figura 2.10: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap02img03code.txt

Note que ele utilizou um caminho diferente. Ele pode verificar que existe um índice chamado `dept_id_pk` (chave primária da tabela criada para o campo `department_id`). Veja que ele classificou e utilizou o rank 4, que é o acesso, entre outras opções, por `primary key`. Vejamos mais um exemplo:

```
SQL> select * from departments where location_id between 1000 and
 1700;

Plano de Execução
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'DEPARTMENTS' (TABLE)
   2    1     INDEX (RANGE SCAN) OF 'DEPT_LOCATION_IX' (INDEX)
```

Figura 2.11: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap02img04code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Neste caso, o caminho escolhido foi o rank 10, pois, conforme podemos ver, ele utilizou o índice existente para a coluna `location_id` (`dept_location_ix`), dentro de um range limitado (`RANGE SCAN`) por conta da expressão `between` .

Agora vamos reescrever a instrução, juntando as duas cláusulas `where` dos dois exemplos anteriores e ver o que acontece:

```
SQL> select *
  2  from   departments
  3  where  department_id = 40
  4  and    location_id between 1000 and 1700;

  Plano de Execução
  ----------------------------------------------------------
     0      SELECT STATEMENT Optimizer=RULE
     1    0   TABLE ACCESS (BY INDEX ROWID) OF 'DEPARTMENTS' (TABLE)
     2    1     INDEX (UNIQUE SCAN) OF 'DEPT_ID_PK' (INDEX (UNIQUE))
```

Figura 2.12: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap02img05code.txt

Veja que o otimizador voltou a utilizar o índice `dept_id_pk` para a busca dos dados, ou seja, uma busca por colunas indexadas, com valores únicos ou que tenha `primary key` . Desta forma, optou novamente pelo rank 4, já que a outra opção o levaria para um `RANGE SCAN` , caracterizando um rank 10. Ao olhar a tabela de ordem de preferência, podemos ver claramente o rank 4 sendo melhor que o rank 10.

Tendo em mente a forma como o otimizador escolhe o melhor caminho de acesso quando está trabalhando baseado em regras, temos o seguinte cenário após a execução dessa última instrução.

```
select * from departments
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Para começar, o otimizador já considera o acesso `FULL` na tabela `departments`. Assim, o primeiro caminho de acesso seria o rank 15. Se ele não achar nenhum caminho melhor, é este que ele tomará como o melhor caminho.

```
where  department_id = 40
```

Logo após, ele encontra uma coluna indexada com chave primária (índice único) e com o predicado de igualdade. Olhando na tabela, isso nos traz uma classificação melhor, agora obtendo um rank 4. Este então, poderia ser o caminho de acesso escolhido pelo otimizador, pois temos um rank 15 e um rank 4, sendo este último o melhor na classificação.

```
and location_id between 1000 and 1700;
```

E depois ele encontra outra condição que se refere a uma coluna, também indexada, onde o predicado constitui-se de um intervalo de valores definidos. Considerando esse cenário e aplicando a ordem de preferência, vemos que essa condição emplaca um rank 10. Em resumo, para a instrução `select` temos a seguinte situação:

Rank	Caminho de acesso	Linha de instrução
15	`FULL TABLE SCAN`	`select * from departments`
4	Acesso de uma única linha via índice `unique` ou `primary key`	`where department_id = 40`
10	Intervalo definido de pesquisa, em colunas indexadas	`and location_id between 1000 and 1700;`

Sabendo que, quanto mais alto o rank na tabela, melhor o desempenho, fica mais fácil detectar o porquê da escolha do otimizador por utilizar uma varredura pelo índice `dept_id_pk`

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

referente à coluna `department_id` , tendo em vista que o rank 4 está acima na tabela em comparação ao rank 10 e 15.

CBO x RBO

O modo `CHOOSE` (CBO) também utiliza a mesma tabela de ordem de preferência utilizada pelo modo `RULE` . Entretanto, leva em consideração a seletividade do dado, número de blocos necessários para se fazer um `FULL TABLE SCAN` *versus* o número de blocos obtidos em uma operação de leitura, número de linhas da tabela, cardinalidade, seletividade e outras informações estatisticamente geradas, consistindo nos seguintes passos:

- O otimizador gera um conjunto de planos de execução potenciais baseados nos caminhos disponíveis e `hints` ;
- Estima o custo de cada plano de execução baseado na distribuição dos dados e informações estatísticas armazenadas no dicionário de dados;
- Calcula o custo baseado nos recursos computacionais, incluindo I/O, tempo de CPU e memória requerida. O custo é um valor estimado proporcional ao tempo necessário para a execução da instrução;
- Compara os custos dos planos de execução e escolhe aquele que tiver o menor custo.

Como foi visto, o otimizador tem a função de avaliar as expressões e condições de pesquisa contidas nas instruções SQL. Ele poderá transformar essas condições de pesquisa em expressões equivalentes, para que sejam executadas dentro de um nível de performance mais adequado possível. O desenvolvedor estará ajudando o otimizador a encontrar o caminho mais adequado se

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

escrever instruções SQL menores e mais performáticas. Mais adiante serão mostradas algumas das técnicas para melhorar essa escrita. Vale ressaltar, também, que o bom uso do otimizador é no modo baseado em custo, mas para isso, as estatísticas deverão estar geradas e, principalmente, atualizadas no banco de dados.

2.5 SQL X PL/SQL: COMO MELHORAR UM CENÁRIO?

Quando estamos lidando com problemas de performance, tudo vale a pena ser considerado. Às vezes o problema não está na forma como você escreve seus programas, mas, sim, de onde ou como você os executa. Observe a imagem a seguir, onde é mostrado o papel de cada recurso que pode estar sendo utilizado dentro do contexto de suas aplicações.

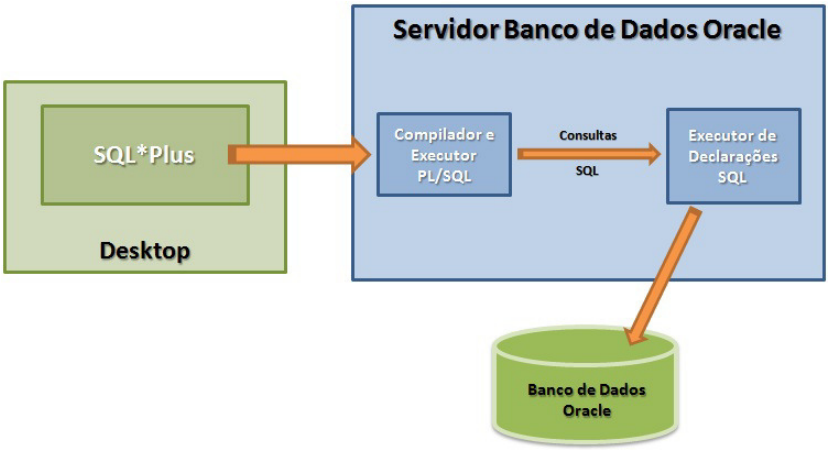

Figura 2.13: Processamento cliente/servidor.

Através do SQL*Plus, podemos entrar com instruções SQL ou

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

blocos PL/SQL. Observando a imagem, podemos ver que esses códigos são encaminhados ao servidor do banco de dados onde ele pode direcionar para o motor PL/SQL, isto é, o processador que vai validar e executar o bloco PL/SQL, e/ou para o executor de declarações de instrução SQL. Através desta visualização é possível entender a finalidade e importância de cada um desses recursos e ferramentas.

Um ponto importante a entender é como são processadas as instruções SQL e blocos PL/SQL dentro de aplicações desenvolvidas com Oracle Forms, Reports ou APEX, por exemplo. Para entendimento, o Oracle Forms, Reports e APEX são ferramentas RAD para o desenvolvimento de formulários e relatórios. Elas se conectam nativamente ao banco de dados Oracle. O termo "nativamente" faz referência a como é realizada a conexão com o banco de dados. Na maioria das vezes, uma aplicação se conecta a um banco de dados através de *drivers* disponíveis pela ferramenta ou por algum gerenciador de conexão, por exemplo, via ODBC (encontrado no Windows). Pois bem, dentro das ferramentas Oracle é possível inserir tanto instruções SQL, quanto blocos PL/SQL. Essas instruções, ou blocos, podem ainda ser executadas de formas diferentes.

Quando temos instruções SQL distintas dentro da aplicação, elas são enviadas uma a uma para o servidor do banco de dados. Dessa forma, a aplicação envia uma instrução para o servidor, espera a resposta e depois envia outra.

Quando temos blocos PL/SQL, eles são enviados por completo ao servidor do banco de dados, não importando o seu teor. Dentro deste bloco, podemos ter várias instruções SQL e demais estruturas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

em PL/SQL. Assim, economizamos tempo, pois a aplicação envia de uma só vez todas as solicitações, e o número de respostas esperadas também reduz muito. Reduzir o número de respostas e tráfego de informações entre aplicação e o servidor do banco de dados aumenta a chance de ganho de desempenho ou evitam problemas dessa natureza. Sabemos que o tráfego de rede também pode ser uma das causas de lentidão na operação de sistemas, principalmente se esta comunicação depender de uma rede cliente *x* servidor. Veja a seguir a ilustração desse conceito.

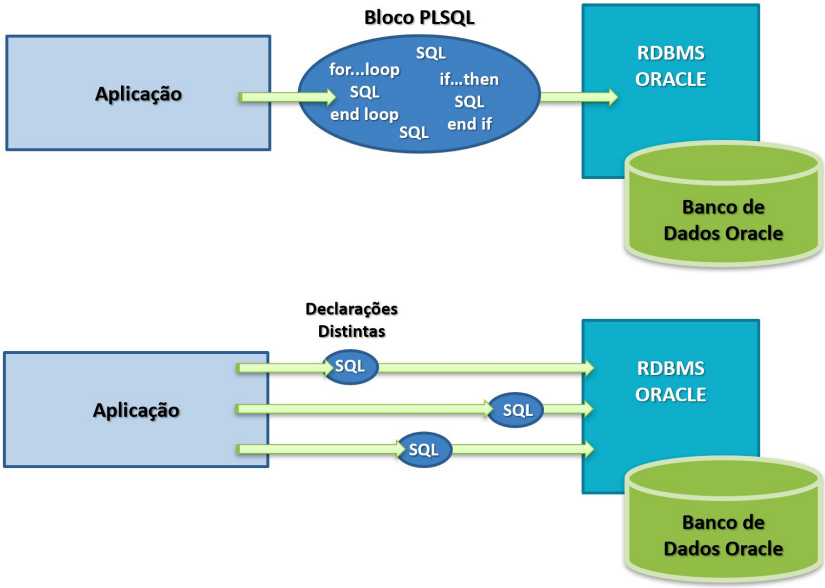

Figura 2.14: Processamento Cliente/Servidor: SQL x PL/SQL. Adaptado de: PL/SQL User's Guide and Reference Release 2 (9.2) https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/01_oview.htm

Nessa imagem, podemos ver que, quando utilizamos blocos PL/SQL dentro de uma aplicação (primeira ilustração), todo o seu conteúdo é enviado para o servidor do banco de dados para

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

processamento interno dele. Podemos, nesse caso, reduzir chamadas ao servidor, encapsulando tudo dentro de um bloco PL/SQL, em vez de ficar enviando várias solicitações distintas (segunda ilustração). Com isso, reduzimos idas e vindas da aplicação ao servidor e ganhamos rapidez nas operações do sistema.

Considere reavaliar a forma como estão sendo executados seus programas ou instruções SQL. Se há instruções SQL sendo executadas de formas distintas, diversas vezes ou trazendo muitos dados, elas podem comprometer a performance de um sistema. Isso também vale para instruções SQL sendo executadas de programas construídos em outras linguagens, como Java, dotNet, PHP, que estejam acessando um banco de dados.

Performance através da sintaxe SQL

Quando escrevemos uma instrução SQL, é pouco provável que ela esteja escrita da forma esperada pelo otimizador. Então, o otimizador de consultas a avalia e, se possível, aplica transformações, sem perder a equivalência, a fim de torná-la mais eficiente para execução. No entanto, essa expressão pode sofrer várias outras transformações conforme vai sendo modificada. À medida que uma regra é aplicada e a transformação é feita, esta nova expressão pode sofrer outras transformações até chegar o momento em que o otimizador julgue aquela expressão como sendo a mais adequada para a execução.

Esse ajuste baseado na sintaxe da SQL tem o intuito de minimizar o trabalho do otimizador de consultas e educar o desenvolvedor no sentido de conhecer como o sistema de

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

otimização do banco de dados com o qual se está trabalhando trata as instruções. Inclusive, instruções mal escritas podem induzir o otimizador a fazer escolhas não muito adequadas dependendo do caso, o que seria o pior dos problemas em relação à má escrita. Vale lembrar que este tipo de otimização não leva em conta os fatores não sintáticos, como índices, tamanhos de tabela, armazenamento etc., e também que os exemplos aqui mostrados são realizados utilizando otimização baseada em custo.

As melhores condições de pesquisas são aquelas constituídas por poucas linhas e de fácil comparação. As tabelas a seguir mostram como são avaliadas as condições, exibindo os possíveis tipos, suas respectivas pontuações, ordenadas da melhor para a pior forma. Quanto mais pontos somar a condição de pesquisa, mais otimizada ela estará.

Segue a tabela de pontos da condição de pesquisa para operadores:

Operador	Pontos
=	10
>	5
>=	5
<	5
<=	5
LIKE	3
<>	0

Segue a tabela de pontos da condição de pesquisa para operandos:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Operando	Pontos
Literal sozinho	10
Coluna sozinha	5
Parâmetro sozinho	5
Expressão com vários operandos	3
Tipo de dado numérico exato	2
Outro tipo de dado numérico	1
Tipo de dado temporal	1
Tipo de dado caractere	0
NULO	0

O otimizador, não exclusivamente o Optimizer da Oracle, já conhece essas regras de pontuação e as aplicará em suas condições de pesquisa. Isso deixa claro que existirão situações em que a otimização manual não será necessária, pois o otimizador fará esse trabalho. A contagem de pontos é uma delas, lembrando que o número de pontos mostrados nas tabelas é apenas representativo para entendimento e não deve ser julgado como sendo a métrica real que o otimizador realiza. Embora esse trabalho já seja realizado pelo otimizador, é importante conhecê-lo a fim de que se possa entender melhor este processo e, ressaltando, procurar tomá-lo como padrão na escrita de instruções SQL. A seguir são mostrados dois exemplos desse tipo de otimização, utilizando consultas simples, sem junções ou subconsultas, ou seja, com uma única tabela:

Exemplo:

```
where smallint_column = 12345
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Esse exemplo possui uma contagem de 27 pontos, que foram calculados conforme os dados das tabelas de pontuação.

- 5 pontos para a coluna (`smallint_column`) sozinha à esquerda;
- 2 pontos pelo fato do tipo de dados operando (`smallint_column`) ser numérico exato;
- 10 pontos para o operador de igualdade;
- 10 pontos para o literal (12345) sozinho à direita.

Exemplo:

```
where char_column >= varchar_column || 'x'
```

Nesse exemplo, a quantidade para a condição de pesquisa foi muito inferior, chegando a somente 13 pontos.

- 5 pontos para a coluna (`char_column`) sozinha à esquerda;
- 0 ponto para o tipo de dados operando CHAR (`char_column`);
- 5 pontos para o operador "maior que" ou "igual a";
- 3 pontos para a expressão com vários operandos (`varchar_column || 'x'` à direita);
- 0 ponto para o tipo de dado operando `varchar` (`varchar_column`).

Outra maneira de otimizar uma condição de pesquisa seria colocar as expressões na ordem adequada, conforme mostra o exemplo a seguir:

```
select *
from   table1
where  column1 = 5
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
and    column2 = 77.3
and    column3 = 'Smith'
and    column4 < 117
and    column4 > column5;
```

As outras transformações que o otimizador pode vir a realizar estão relacionadas aos operadores `like`, `in`, `any` ou `some`, `all`, `between` e `not` em uma condição de pesquisa. Entendendo esses conceitos é possível fazer os ajustes manualmente, poupando tempo do otimizador.

Operador `like`: quando o otimizador encontra, em uma instrução SQL, o operador `like`, tendo como operando valores sem o caractere curinga (`%`, `_`), ele é trocado pelo operador de igualdade. Exemplo:

```
first_name like 'Lex'
```

Transforma-se em:

```
first_name = 'Lex'
```

Operador `in`: o otimizador amplia a condição que utiliza o operador `in` e a transforma em várias condições utilizando o operador de igualdade (`=`) e o operador lógico `or`. Exemplo:

```
first_name in ('Diana', 'John', 'Steven')
```

Transforma-se em:

```
first_name = 'Diana' or first_name = 'John' or first_name = 'Stev
en'
```

Operadores `any` ou `some`: o mesmo procedimento realizado para o operador `in` é aplicado quando operadores `any` ou `some` são encontrados. Entretanto, se o operando for fruto de uma subquery, o otimizador utilizará `exists` como operador.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Exemplo:

```
salary > any (:first_salary, :second_salary)
```

Transforma-se em:

```
salary > :first_salary or salary > :second_salary
```

Utilizando subquery:

```
x > any (select salary
           from   employees
            where  job_id = 'ST_MAN')
```

Transforma-se em:

```
exists (select salary
          from   employees
        where  job_id = 'ST_MAN'
          and    x > salary)
```

Operador `all` : o otimizador amplia a condição que utiliza o operador `all` e a transforma em várias condições utilizando o operador de igualdade e o operador lógico `and` . Exemplo:

```
exists (select salary
          from   employees
          where  job_id = 'ST_MAN'
          and    x > salary)
```

Transforma-se em:

```
sal > :first_salary and salary > :second_salary
```

Entretanto, se o operando for fruto de uma subquery, o otimizador poderá utilizar outros operadores. No exemplo a seguir, o otimizador transformará utilizando `not` e `any` . O curioso é que, como os operandos utilizados para o operador `any` são valores vindos de uma `subquery` , o otimizador transformará a condição novamente a fim de simplificá-la ainda mais.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
x > all (select salary
           from   employees
           where  department_id = 10)
```

Primeira transformação:

```
not (x <= any (select salary
                 from   employees
                 where  department_id = 10))
```

Segunda transformação:

```
not exists (select salary
              from   employees
              where  department_id = 10
              and    x <= salary)
```

Operador `between` : sempre que o otimizador encontrar o operador `between` em uma condição de pesquisa, ele o transforma utilizando os operadores maior ou igual que (`>=`) e menor ou igual que (`<=`). Exemplo:

```
salary between 8000 and 10000
```

Transforma em:

```
salary >= 8000 and salary <= 10000
```

Operador `not` : quando o otimizador encontra o operador `not` em uma condição de pesquisa, ele transforma esta condição, retirando este operador e invertendo o operador de comparação da condição. Exemplo:

```
not department_id = (select department_id
                     from   employees
                     where  first_name = 'David')
```

Transforma-se em:

```
department_id <> (select department_id
                  from   employees
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
where  first_name = 'David')
```

Existem várias formas de escrever uma condição de pesquisa que contém o operador `not`. O otimizador fará essas transformações, procurando encontrar um meio de tornar possível a simplicidade da condição, até mesmo se for preciso que a condição resultante contenha mais de um `not`. Exemplo:

```
not (salary < 1000 or commission_pct is null)
```

Primeira transformação:

```
not salary < 1000 and commission_pct is not null
```

Segunda transformação:

```
salary >= 1000 and commission_pct is not null
```

O otimizador também realiza otimizações com base na **Lei da Transitividade**, que diz: se existirem duas condições que envolvem a cláusula `where` em colunas comuns, então o otimizador poderá, às vezes, deduzir uma terceira condição. Através dessa terceira coluna, pode-se fazer com que seja disponibilizado um caminho de acesso de índice, o qual na consulta original não era possível, por exemplo.

Sintaxe:

```
where column1 operador_de_comparação constante
and column1 = column2
```

Então, pode-se dizer que:

```
column2 operador_de_comparação constante
```

Onde: `operador_de_comparação` pode ser: `=`, `!=`, `^=`, `<`, `<>`, `>`, `<=`, ou `>=`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Exemplo: este exemplo envolve duas condições referentes à coluna `employees.department_id` .

```
select *
from   employees
      ,departments
where  employees.department_id = 20
and    employees.department_id = departments.department_id;
```

Usando o princípio da transitividade:

```
select *
from   employees
      ,departments
where  departments.department_id = 20
and    employees.department_id = departments.department_id;
```

Nesse caso, se existisse um índice para a coluna `departments.department_id` , por ele ser uma chave primária ou única, seria usado o caminho de acesso através deste índice.

O otimizador só deduzirá condições que forem relacionadas a colunas e expressões constantes. Portanto, não aplicará o princípio da transitividade onde houver relacionamento de colunas para outras colunas.

Sintaxe:

```
where column1 operador_de_comparação column3
and   column1 = column2
```

Nesse caso, o otimizador não deduzirá a condição `column3 operador_de_comparação column2` . Exemplo:

```
1  select e.first_name,
2         d.department_name,
3         count(j.job_id)
4  from   employees e, departments d, job_history j
5  where  e.department_id = d.department_id
6  and    e.department_id = j.department_id
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
7  and    e.job_id = j.job_id
8  group  by e.first_name, d.department_name
9  group  by e.first_name
```

Nas linhas 5 e 6, é possível observar que o campo `e.department_id` da tabela `employees` se relaciona com o campo `d.department_id` da tabela `departments` e também da tabela `job_history`, pelo mesmo campo. Nesse caso, o otimizador não vai deduzir que, se `e.department_id` é igual a `d.department_id` e `e.department_id` é igual a `j.department_id`, então isso é o mesmo que dizer que `d.department_id` é igual a `j.department_id`.

Isso se deve ao fato de o otimizador não conhecer os valores de ambas as colunas. O contrário aconteceria se esse exemplo fosse escrito desta forma:

```
1  select e.first_name,
2         d.department_name,
3         count(j.job_id)
4  from   employees e, departments d, job_history j
5  where  e.department_id = d.department_id
6  and    e.department_id = j.department_id
7  and    d.department_id = 50
8  and    e.job_id        = j.job_id
9  group by e.first_name, d.department_name
10 order by e.first_name
```

Como foi utilizada uma constante na condição de pesquisa, o otimizador vai procurar entre as outras condições que possam ser simplificadas e que causem uma carga menor de trabalho. Na transformação do otimizador, ficaria assim:

```
1  select e.first_name,
2         d.department_name,
3         count(j.job_id)
4  from   employees e, departments d, job_history j
5  where  e.department_id = 50
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
6  and    e.department_id = j.department_id
7  and    d.department_id = 50
8  and    e.job_id        = j.job_id
9  group  by e.first_name, d.department_name
10 order  by e.first_name
```

Pelas linhas 5 e 7, fica evidente a transformação que o otimizador faria nesse exemplo. Pois, se `e.department_id` é igual a `d.department_id` e `d.department_id` é igual `50`, então pode-se deduzir que `e.department_id1` é igual a `50`, também.

Quando existir uma expressão com múltiplas condições de pesquisa e acontecer de uma das colunas ser substituída por um valor constante, este processo é chamado de **Propagação Constante**.

No caso da otimização de subexpressões comuns, o otimizador identifica, através da heurística, formas para reduzir as condições de pesquisa em caso de redundância de expressões, a fim de separar expressões em comum, para as partes da consulta. Para que o otimizador trabalhe com a otimização de subexpressão, é preciso habilitar o parâmetro de inicialização chamado `optimizer_features_enable`.

O otimizador só aplicará a otimização de subexpressões comuns se as partes forem separadas pelo operador `or` e se a cláusula `where` estiver respeitando as seguintes regras:

- Se a separação através do operador `or` estiver em um nível acima em caso de haver várias condições;
- Se cada separação for um predicado simples (um predicado é considerado simples se não contiver `and` ou `or`) ou uma união;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Se cada união for um predicado simples ou uma separação de predicados simples;
- Se a expressão aparecer em todas as partes da instrução em questão.

Exemplo: **Otimização de subexpressões comuns contendo OR entre duas partes (reduz 2 em 1)**

A instrução a seguir tem o objetivo de selecionar os nomes dos empregados que trabalham no departamento `'Sales'`, cujo salário seja maior que 40.000 ou que a data de admissão seja 01/01/2004.

```
select e.first_name
from employees e, departments d
where (d.department_id = e.department_id
       and e.hire_date = to_date('01/01/2004','dd/mm/yyyy')
       and d.department_name ='Sales')
or  e.department_id = d.department_id
and e.salary > 40000 and d.department_name = 'Sales'
```

O otimizador eliminará a redundância gerando uma expressão isolada e juntará as outras expressões que fazem parte da condição e que são diferentes, em outra.

```
select e.first_name
from   employees e, departments d
where  (d.department_id = e.department_id and d.department_name =
'Sales')
and    (e.hire_date = to_date('01/01/2004','dd/mm/yyyy') or e.sal
ary > 40000);
```

Existe também uma regra para condições que envolvem colunas que utilizam comparações com constantes. O exemplo a seguir mostra as várias formas de se obter os salários maiores que 2000.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
salary > 24000 / 12

salary > 2000

salary * 12 > 24000
```

Se a condição de pesquisa estiver sendo representada da primeira forma `salary > 24000 / 12`, então o otimizador a transformará na segunda `salary > 2000`. Se estiver na terceira forma, `salary * 12 > 24000`, o otimizador não fará a transformação. Portanto, sempre que possível, use expressões que envolvam constantes e não expressões que envolvam colunas.

Quando há uma consulta que contém a cláusula `where` com múltiplas condições combinadas com `or`, o otimizador as transforma em uma expressão equivalente utilizando `union all`, se isso for tornar a execução mais eficaz e estiver dentro das regras impostas para esta transformação:

- Se cada condição faz um acesso ao índice disponível, o otimizador faz a transformação;
- Se alguma condição requer um `FULL TABLE SCAN` (porque o índice não está disponível) o otimizador não transforma a declaração.

Exemplo:

```
select *
from   employees
where  job_id = 'FI_MGR'
or     department_id = 10;
```

Transforma-se em:

```
select *
from   employees
where  job_id = 'FI_MGR'
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
union all
select *
from   employees
where  department_id = 10
and   job_id <> 'FI_MGR';
```

Em caso de subconsultas, o otimizador transforma esta declaração complexa em uma declaração utilizando relacionamento (`join`) entre as tabelas, sempre que o resultado obtido através do relacionamento garantir que o resultado, isto é, o número de linhas, seja exatamente igual à declaração complexa.

Exemplo:

```
select *
from employees
where job_id in
        (select job_id from jobs);
```

Se a coluna de `job_id` da tabela de `jobs` for uma chave primária ou tiver uma `constraint unique` agregada a ela, então o otimizador poderá transformar a declaração complexa em uma declaração equivalente sem alterar o resultado:

```
select employees.*
from   employees, jobs
where  employees.job_id = jobs.job_id;
```

Em caso de declarações complexas utilizando funções de agregação, como `avg` e `sum` , o otimizador não fará a transformação. Exemplo:

```
select *
from   employees
where  employees.commission_pct >
        (select avg(commission_pct) from employees);
```

Outra transformação que o otimizador pode aplicar é com relação às `views` . No caso de haver alguma tabela na cláusula

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`from` da instrução SQL que seja uma `view` , o otimizador tentará unificar a instrução SQL com o comando implícito da `view` . Se por um acaso o otimizador não conseguir fazer essa unificação, ele executará a `view` , e as linhas que retornarem serão utilizadas como se fossem uma tabela. A unificação da instrução SQL pode ser feita utilizando as seguintes técnicas:

- Juntar a consulta da `view` dentro da instrução SQL ou
- Juntar a instrução SQL dentro da consulta da `view` .

Pode-se obter a performance também através da maneira como você expõe suas instruções SQL. O otimizador da Oracle, após avaliar e executar, guarda as declarações a fim de que, caso elas sejam executadas novamente, ele possa usar os mesmos planos de execução. No momento da verificação e armazenagem da instrução, são avaliados espaços em branco, letras maiúsculas e minúsculas, acentos e disposição de colunas e cláusulas, ou seja, as declarações devem ser igualmente escritas para que o otimizador possa aproveitá-las. Desta forma, o melhor a fazer é criar um estilo para a escrita de instruções SQL com o intuito de padronizar essa escrita, criando instruções com maior legibilidade e assim auxiliar o otimizador. Algumas regras podem ser estabelecidas para que se crie um estilo de escrita:

- Coloque as palavras-chaves em letras maiúsculas e os nomes de colunas em letras minúsculas;
- Use os nomes de tabelas com a inicial maiúscula;
- Use espaços simples ao redor de cada palavra e de cada operador aritmético;
- Use variáveis de substituição (`binds`) quando for utilizar expressões variáveis.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Vimos que através do ajuste da sintaxe é possível obter performance na execução das instruções SQL. Por meio dessas técnicas, também foi possível conhecer como o otimizador faz para executar essas instruções e, assim, melhorá-las de forma a auxiliar você nesse processo. No entanto, vale ressaltar, que não só da sintaxe é garantida a performance, pois existem outros fatores que devem ser considerados, que veremos mais à frente nos próximos capítulos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 3

PROCESSANDO INSTRUÇÕES SQL

Para conhecer melhor o processo de otimização é necessário entender como funcionam as fases do processamento de uma instrução SQL, as áreas compartilhadas de SQL (*Shared SQL Areas*) e os cursores compartilhados (*Shared Cursors*). Dessa forma, você vai entender como padrões de codificação podem minimizar a quantidade de vezes que suas instruções SQL serão analisadas e otimizadas.

Além disso, quanto mais você escrever instruções SQL de forma padronizada, menor será a frequência com que você precisará se dedicar a ajustes de performance, tanto você, quanto o próprio otimizador do banco de dados.

3.1 PASSOS PARA A EXECUÇÃO DE INSTRUÇÕES SQL

Vimos anteriormente que os passos para a execução de uma instrução SQL são basicamente três: `Parse`, `Execute` e `Fetch`. Esses três passos são os principais, contudo, entre o `Parse` e o `Execute`, pode ainda acontecer outro, chamado `Bind`. Esse

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

passo acontece quando nas instruções SQL são encontradas referências de variáveis que são chamadas de `bind` ou variável de ligação. A variável `bind` funciona como se fosse um parâmetro na instrução SQL, sendo ela uma forma de passagem dinâmica de valores para a instrução. Desta forma, não se utilizam valores fixos para serem passados para instrução e sim, uma variável que dentro dela contém o valor a ser utilizado.

No Oracle, as variáveis `bind` são precedidas por dois pontos, como `:nome_funcionario`. Na fase do `Bind`, o Oracle não sabe os valores das variáveis `bind` ao otimizar a declaração. Isso permite um rápido `rebind-execute` sem a necessidade de recompilação, o que poupa tempo e memória. Uma desvantagem é que é impossível para o otimizador estimar a seletividade, pois ele não conhece os dados que estão dentro da variável. Dentro da ordem de execução, vamos revisar cada passo:

`Parse`:

- Procura pela instrução na `SHARED POOL`.
- Verifica a sintaxe da instrução, determinando a gramática e especificações da linguagem SQL.
- Verifica a semântica, garantindo que os objetos referenciados na instrução SQL são válidos e satisfazem as regras de integridade e segurança.
- Pode fazer transformações na instrução SQL quando se tratar de uma view, ou transformações em uma view quando se tratar de uma instrução SQL, e tenta simplificar a instrução eliminando subconsultas (subqueries) se for o caso, substituindo-as por condições `join` na consulta principal.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Determina e armazena o plano de execução.

Bind :

- O servidor Oracle faz uma verificação na instrução, buscando por referências de variáveis `bind` .
- O servidor Oracle atribui ou reatribui valores para cada variável `bind` encontrada.

Execute :

- O servidor Oracle aplica a `parse tree` aos buffers de dados.
- Múltiplos usuários podem compartilhar a mesma `parse tree` .
- O servidor Oracle efetua leituras físicas ou leituras/escritas lógicas para instruções DML, e também ordena os dados quando necessário.

Fetch :

- O servidor Oracle recupera as linhas para um comando `select` durante a fase de `Fetch` . Cada `Fetch` normalmente recupera múltiplas linhas, utilizando um `array fetch` (obedecendo ao `arraysize` configurado), onde constam os dados.

NOTAS: O tamanho do `arraysize` (tamanho do `array`) pode ser configurado no SQL*Plus através do parâmetro `arraysize` , com o comando `set array[size] array_size` , onde `array_size` é o número de linhas que

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

você deseja configurar, podendo variar de 1 a 5000. O padrão é 15 linhas.

Outros passos não foram mencionados aqui, pois são muito específicos de determinadas situações, são eles: `Define` , `Describe` e `Parallelize` . Para saber mais, consulte a documentação do banco de dados. Confira o diagrama mostrando os passos para a execução de instruções SQL, incluindo o passo `Bind` :

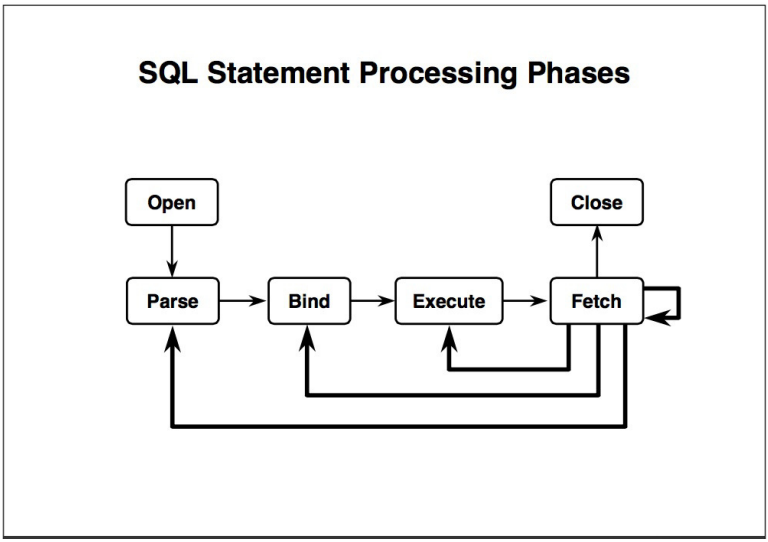

Figura 3.1: Fonte: Oracle9i: SQL Tuning Workshop, Instructor Guide Vol. 1.

Agora que vimos como uma instrução SQL é compilada e processada, chegou a hora de ver onde isso acontece e como podemos obter benefícios deste processo.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

3.2 SHARED SQL AREAS

A *Shared Pool* é uma parte da *System Global Area* (SGA), que é uma área de memória usada para armazenar informações do banco de dados Oracle e que é compartilhada por processos do banco de dados. Ela é alocada na memória virtual do servidor Oracle e contém várias subáreas que são usadas para organizar e gerar diferentes tipos de informação. O servidor Oracle usa a memória para armazenar vários tipos de informações, tais como código de programas que estão sendo executados, informações sobre as sessões ativas, instruções SQL, e assim por diante. Vários processos podem gerir essa área de memória. Os principais componentes da Shared Pool são a *Library Cache* e o *Data Dictionary Cache*. Sendo assim, a `Shared Pool` é constituída por:

- Cache do Dicionário de Dados (*Data Dictionary Cache*).
  - Guarda informações de usuários, informações de segmentos, informações de `tablespaces` e `sequence numbers`.
- Library Cache.
  - Instruções SQL (*SQL statements*).
  - Códigos analisados e compilados de blocos PL/SQL (*Parsed or compiled PL/SQL blocks*).
  - Classes Java.

A *Shared SQL Area*, por sua vez, é parte da `Library Cache`. É nela que vamos nos concentrar. A figura a seguir mostra a SGA como um todo e também a localização da `Shared Pool`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

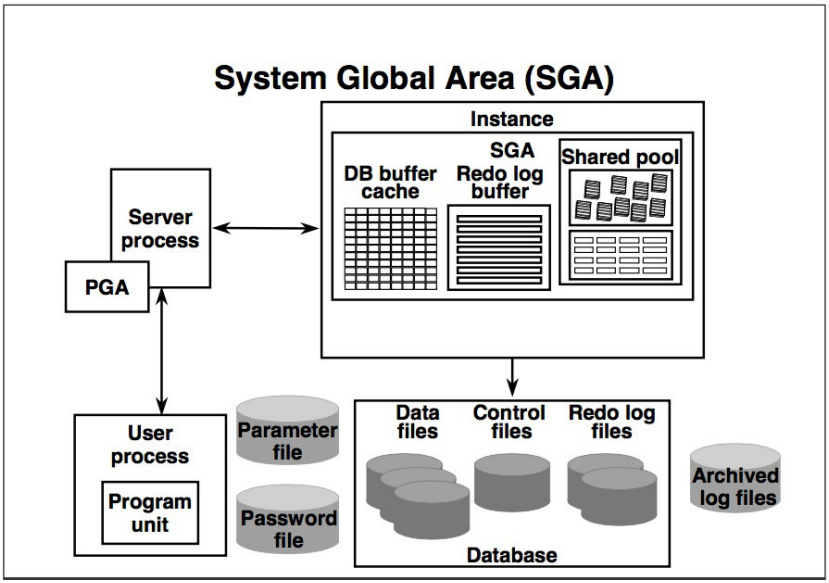

Figura 3.2: Fonte: Oracle9i: SQL Tuning Workshop, Instructor Guide Vol. 1.

Quando você executa uma instrução SQL, o motor do banco de dados Oracle tenta verificar se ela já existe na `Library Cache` por uma execução realizada anteriormente. Se existir a instrução na `Library Cache` , ele a compartilha para a execução atual. Nesse caso, ele reutiliza o código já existente. Isso é algo muito inteligente, pois ele parte do princípio de que uma compilação e uma análise foram realizadas anteriormente, e, se a instrução for exatamente a mesma, não será necessário realizar todo esse trabalho novamente.

Contudo, essa área de memória é delimitada e, com isso, algumas instruções podem não estar mais nessa área. Quando a `Shared Pool` enche, o servidor Oracle libera espaço para programas recém-executados. Para controlar isso, ele utiliza um

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

algoritmo (chamado LRU) que analisa as instruções menos utilizadas e vai dando lugar para as mais novas. Instruções que não são usualmente executadas permanecem no `Shared Pool` por um curto período de tempo. Veja a seguir um diagrama mostrando o funcionamento das `Shared SQL Areas` .

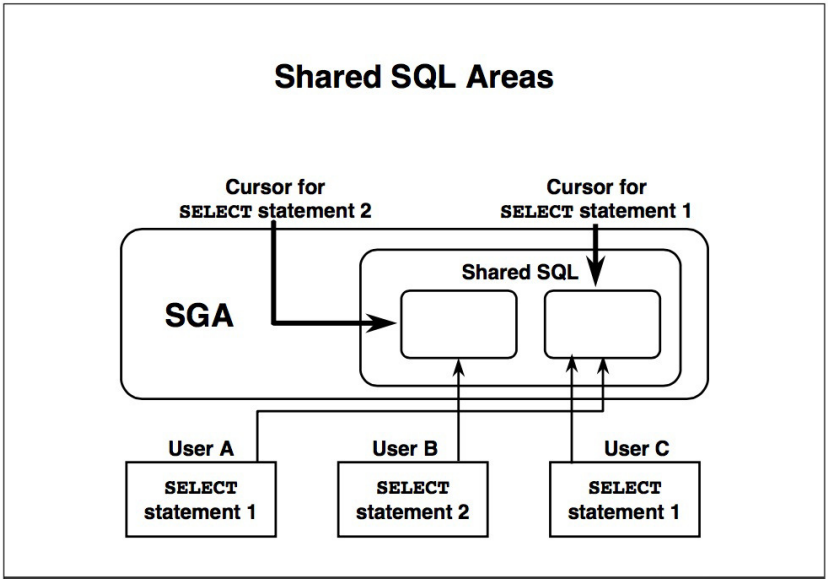

Figura 3.3: Fonte: Oracle9i: SQL Tuning Workshop, Instructor Guide Vol. 1.

Nota: O DBA é quem pode alterar o tamanho da `Shared SQL Areas` modificando, para isso, parâmetros de banco.

Cursores

Dentro da `Shared SQL Area` , cada instrução SQL é analisada

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

(Parse) na sua totalidade. Esta área pode ser chamada de área de contexto (Context Area) ou cursor. Dentro de cada cursor analisado, temos as seguintes informações:

- A instrução analisada: instruções SQL estáticas, dinâmicas e recursivas, além de program units , tais como procedures e triggers de banco de dados;
- O plano de execução;
- Uma lista de objetos referenciados.

Um ponto interessante dos cursores é que se dois ou mais usuários, por exemplo, executarem a mesma instrução SQL, então eles compartilharão do mesmo cursor. Só lembrando, isso se a instrução em questão ainda existir na Shared Pool . Caso contrário, todo o trabalho de Parse e compilação da instrução será realizado, novamente. Existirão situações em que a instrução existirá na Shared Pool mas seu cursor poderá estar inválido. Isso acontece, por exemplo, se uma instrução DDL, como alter table , foi executada para um dos objetos da instrução SQL, ou se uma tabela dependente foi analisada (no caso de uma atualização de estatística). Nesse caso, não ocorrerá nenhum erro, mas a instrução SQL será analisada, passando pelo Parse e sendo compilada como uma nova instrução, gerando um novo cursor que será guardado na Shared Pool para futuras execuções.

Shared Cursor

Quando uma instrução SQL é encontrada na Shared SQL Area , tal instrução não precisa passar pela fase do Parse , pois todas as informações inerentes a ela, como o plano de execução, já foram armazenadas antes e estão prontas para a próxima fase.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Com isso, podemos listar os principais benefícios deste compartilhamento:

- O compartilhamento de cursores (`Shared Cursor`) reduz o `Parse` e economiza tempo;
- A memória ajusta-se dinamicamente a instrução SQL executada;
- O uso de memória pode melhorar dramaticamente, mesmo para ferramentas que armazenam instruções SQL dentro do sistema.

Para se beneficiar dos recursos de compartilhamento, a pessoa desenvolvedora precisa fazer sua parte investindo na padronização de códigos e boas práticas de escrita. Para que o uso compartilhado de cursores seja melhor aproveitado, as instruções SQL precisam ser idênticas. Eis os principais requisitos:

- Os textos das instruções SQL devem ser exatamente os mesmos, incluindo as distinções entre letras maiúsculas e minúsculas (*case sensitive*), espaços, tabulações, retornos de dados e comentários;
- Os objetos referenciados nas instruções SQL devem apontar para os mesmos objetos no banco de dados;
- Os tipos de variáveis de `bind` utilizados nas instruções SQL devem ser os mesmos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Antes de enviar instruções SQL para o servidor Oracle, a maioria das ferramentas Oracle (como PL/SQL, pré-compiladores e ferramentas de desenvolvimento Oracle) processam essas instruções com o intuito de torná-las o mais idênticas possível. Esse trabalho é feito através da remoção de comentários, subtração de espaços em branco e conversão de textos para maiúsculas ou minúsculas. Mas preste atenção, a ferramenta Oracle SQL*Plus, por exemplo, envia as instruções SQL para o servidor Oracle no mesmo formato em que são escritas pelo desenvolvedor.

Escrevendo instruções SQL para cursores compartilhados

Se houver distinção entre textos com letras maiúsculas e minúsculas ou se a quantidade de espaço em branco for diferente, então as instruções SQL não serão consideradas idênticas.

```
select * from customers where cust_id = 180;
```

```
select * from CUSTOMERS where CUST_ID = 180;
```

Nesses dois primeiros exemplos, podemos ver que, embora os resultados sejam os mesmos, para a `Shared SQL Area` se tratam de instruções diferentes. Isso acontece porque tanto o nome da tabela quanto o nome da coluna, entre os `selects`, estão escritos com cases diferentes, um com nomes em maiúsculo e outro com nomes em minúsculo.

Se os objetos pertencerem a diferentes usuários, então as

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

instruções SQL não serão consideradas idênticas.

```
select * from customers where cust_id = 180;

select * from customers where cust_id = 180;
```

Nessas duas instruções SQL do exemplo, as escritas são idênticas. No entanto, se a tabela `customers`, utilizada nas duas instruções, existisse em `schemas` diferentes, ou seja, caracterizando objetos diferentes, para a `Shared SQL Area`, essas instruções seriam distintas. Logo, o servidor Oracle tratará essas instruções como duas referências de cursor.

Reaproveitamento de código e padrões de formatação

O reaproveitamento de código é muito útil para manter a `Shared SQL Area` saudável. Através desse processo é possível criar rotinas que sejam reutilizadas em vários sistemas por vários desenvolvedores. Por exemplo, uma rotina para validar o número de CPF é algo que pode ser utilizado em várias partes de um sistema. Nesse caso, você poderá criar uma rotina no banco de dados contendo essa função, com a qual todas essas partes podem acessar essa mesma rotina. Veja como reaproveitar seus códigos:

- Escrever e armazenar `packages`, `procedures` e `functions` que possam ser compartilhadas entre aplicações;
- Utilizar `triggers` de banco de dados;
- Utilizar variáveis `bind` quando possível;
- Escrever bibliotecas para serem utilizadas em vários programas ou ambientes.

A imagem a seguir exemplifica o conceito. Nela, temos vários

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

programas que necessitam de uma função para validar o documento de CPF e todos estão chamando uma mesma função criada no banco de dados:

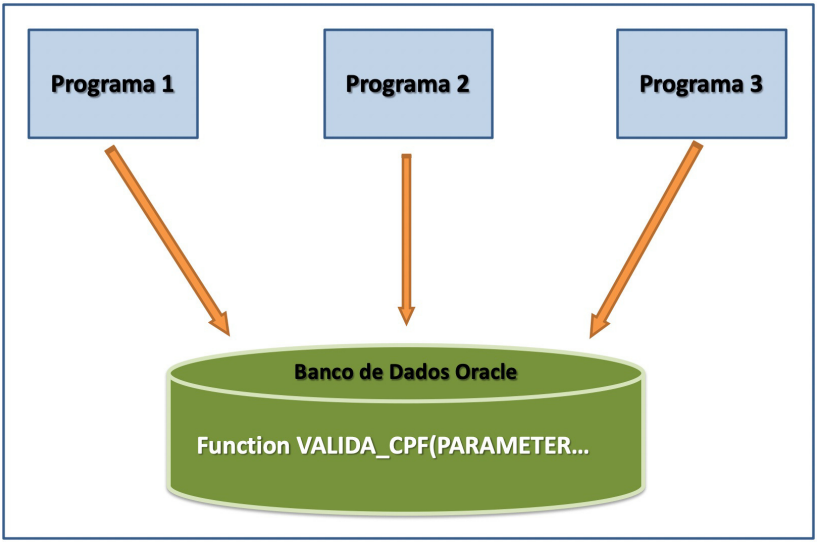

Figura 3.4: Visão de código centralizado no banco de dados.

Já os padrões de formatação ou convenções de codificação, além de ser o ponto-chave para contribuir para a `Shared SQL Area` e, consequentemente, para o uso de cursores compartilhados, vai tornar mais limpa e unificada toda e qualquer codificação de programas. Além disso, qualquer programa codificado dentro desses padrões ou convenções, será familiar a todos os desenvolvedores da equipe. Seguem alguns exemplos de padrões que podem ser aplicados:

- Definir padrões de formatação para todas as instruções, incluindo as codificadas em PL/SQL;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Definir regras para a utilização de textos em maiúsculas e minúsculas;
- Definir regras para a utilização de espaços em branco e tabulações (indentação de código);
- Definir nomenclatura para nomes de variáveis e parâmetros e também o uso de variáveis `bind` ;
- Definir regras para comentários, de preferência mantendo-os fora das instruções SQL;
- Utilizar os mesmos nomes para referenciar objetos idênticos do banco de dados.

A seguir, uma imagem que exemplifica como escritas não padronizadas e a falta de convenções são vistas e interpretadas pelo motor do banco de dados:

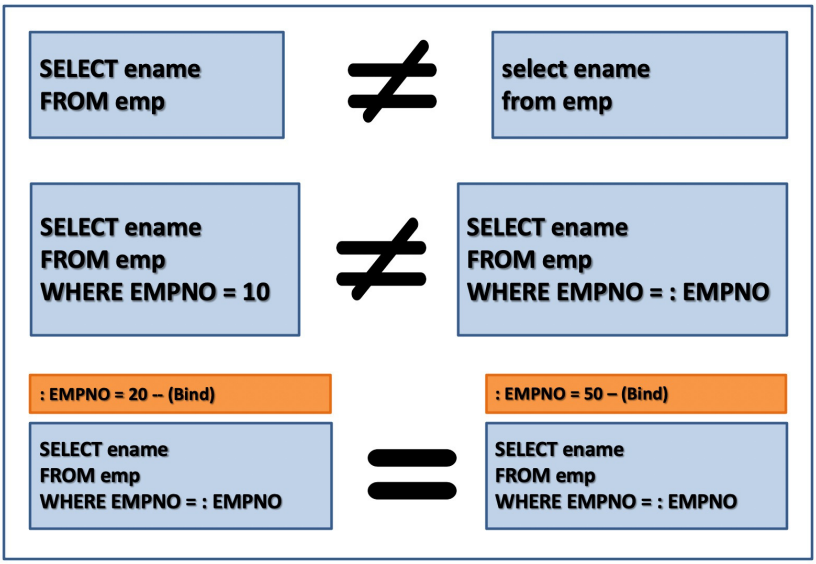

Figura 3.5: Exemplos de uso e não uso de padrões e regras.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Cursores compartilhados: variáveis bind

Variáveis `bind` são muito úteis quando estamos falando de cursores compartilhados, pois, neste caso, em vez de trabalharmos com a constante em si, que pode ser diferente dependendo da execução, podemos utilizá-las para evitar a duplicidade de instruções SQL.

Se, em duas instruções relativamente idênticas, duas variáveis `bind` têm diferentes tipos de dados, então as instruções são consideradas diferentes para a `Shared SQL Area`. Contudo, se os tipos dos dados das variáveis `binds` forem iguais, não importa que seus nomes sejam diferentes, pois o servidor Oracle as renomeia internamente e, para ele, serão idênticas. Este é um processo interno do Oracle, onde a primeira variável `bind` é sempre chamada de :B1, a seguinte variável `bind` é chamada de :B2 e assim por diante. Veja o exemplo a seguir.

As instruções SQL a seguir:

```
select * from customers where cust_id = :c

select * from customers where cust_id = :d
```

Serão transformadas nesta:

```
select * from customers where cust_id = :b1
```

Nesse caso, para o Oracle, as duas instruções anteriores são consideradas idênticas, embora tenham nomes de variáveis `bind` diferentes.

Monitorando Shared Cursors

Para o monitoramento e levante de informações sobre os

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

cursores compartilhados, podemos utilizar as views `v$librarycache` e `v$sqlarea`. Nelas, ficam as informações pertinentes aos cursores e também os seus dados de uso e definição. Através dessas informações, você poderá detectar, por exemplo, que um determinado cursor, embora bastante utilizado, tem que ser reanalisado por sempre estar inválido, ou porque parece nunca ser armazenado, mesmo que seu conteúdo seja sempre idêntico a instruções já executadas anteriormente.

É possível também verificar quantas vezes o servidor não encontrou uma instrução em memória. Estas e outras informações poderão dar orientações e explicações sobre problemas de performance que podem acontecer em um ambiente de banco de dados. Peça apoio ao DBA para analisar tais informações dessas views para ver se essas áreas estão saudáveis.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 4

UTILIZAÇÃO DE ÍNDICES

Neste capítulo, falaremos dos índices com relação à otimização de sistemas que utilizam banco de dados. A utilização de índices é um dos principais temas quando o assunto é performance.

Índices são estruturas de dados mantidas internamente pelo gerenciador de banco de dados para otimizar o acesso aos dados. Dessa forma, seu principal propósito é aumentar a performance na busca pelos dados nas tabelas do banco de dados. Entretanto, ao contrário do que muitos pensam, esse propósito só será alcançado quando o retorno das linhas selecionadas for pequeno, comparado à quantidade de linhas das tabelas acessadas. Enfatizando isso, podemos dizer que índices podem aumentar a performance das instruções, sim, mas provavelmente se estiverem selecionando apenas um percentual pequeno de linhas. A seguir, algumas considerações sobre o uso dos índices:

- Deve-se criar índices em tabelas que sejam frequentemente utilizadas e cujo retorno das linhas seja de 2% a 4%;
- Essa porcentagem pode ser um pouco maior nos casos em que a busca pelos dados seja feita apenas pelas colunas indexadas, não acessando a tabela, ou em casos em que as colunas indexadas possam ser utilizadas em junções de tabelas;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Um índice no Oracle sempre existirá, mesmo que não esteja sendo usado;
- A manutenção em índices gera demandas de CPU e I/O e sua criação deve ser analisada.

Uma das razões do ganho de performance através dos índices é o fato de que eles guardam somente os valores que pertencem à chave de índices. Dessa forma, se a busca pelos dados for referente aos campos que compõem determinado índice, o otimizador não precisará acessar a tabela para buscar esses dados, em vez disso, os encontrará na própria chave do índice. Mesmo que os dados requisitados em uma instrução SQL não façam parte da chave de índice, a busca será mais rápida se os critérios utilizados na cláusula `where` estiverem usando com eficiência as chaves de índice. Quando não há existência de índices ou sua utilização não está sendo adequada, o otimizador poderá fazer uma varredura completa nas tabelas contidas na cláusula `from`, podendo tornar o processamento lento.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

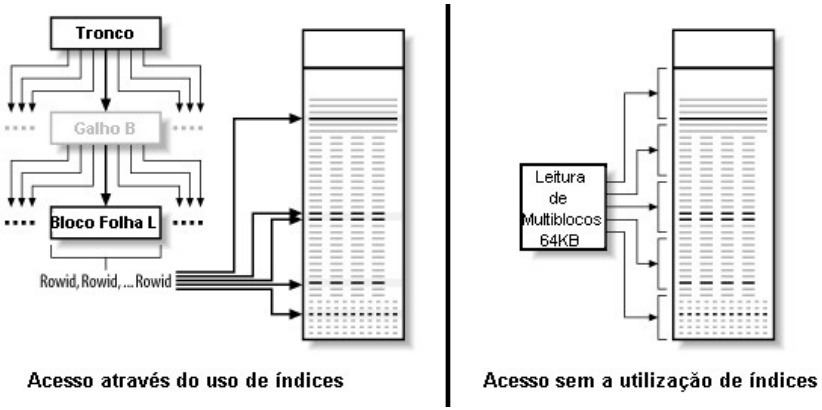

Figura 4.1: A figura mostra o acesso aos dados através de índices e por acesso direto à tabela. Do lado esquerdo do desenho, é apresentado o acesso utilizando índices. Com base nos critérios informados na cláusula `where`, são procuradas as `rowids` no índice e, através destas, é realizado o acesso direto aos registros da tabela. Já na busca realizada através da varredura completa na tabela, isto é, sem a utilização de índices, todos os blocos correspondentes aos dados terão que ser lidos. A consequência disso é a leitura de todos os registros da tabela, buscando aqueles que correspondem aos critérios informados. Adaptado de: SQL Tuning: Generating Optimal Execution Plans.

Os ganhos de performance através do uso de índices refletem em instruções `select`, `update` e `delete`. Entretanto, quando se trata de instruções `insert`, o desempenho diminui. Isso acontece porque, cada vez que é inserida uma linha em uma tabela que possui colunas indexadas, é inserida também uma linha no índice. Portanto, a inserção em tabelas com colunas indexadas tende a ser mais lenta. Contudo, uma operação `delete` excluindo a metade dos registros de uma tabela também deverá excluir metade das linhas do índice. No caso da atualização de uma coluna indexada, acontecerá a exclusão e a inserção de uma linha do índice. A linha contendo o valor antigo é excluída e uma com o novo valor é inserida, o que pode deixar o processo lento. Nesse caso, devem ser analisados os benefícios que o índice proporcionará, tendo em vista que haverá ganhos de desempenho

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

na recuperação de linhas, mas poderá ter efeitos negativos quando forem usadas instruções `insert` , `delete` e `update` com grande frequência, ou o retorno de linhas for muito grande.

Em muitos casos, pode ser vantajoso perder performance em uma dessas instruções, mas ganhar na recuperação das linhas - isso dependerá do objetivo em questão. Geralmente, não aparecem muitos problemas com relação à manipulação de dados em instruções `insert` , `delete` e `update` , a não ser que essas operações estejam trabalhando com uma massa de dados muito grande ao mesmo tempo, via uma exclusão ou atualização de muitos registros de uma vez só.

Um índice também pode ser considerado seletivo ou não seletivo pelo otimizador. A seletividade de um índice no Oracle é determinada pelo seu valor de uso, que é baseado na instrução e nos dados. Este valor é determinado pelos vários métodos que o Oracle disponibiliza. Um desses métodos é determinar o número de chaves exclusivas ou distintas no índice através da análise da tabela ou do índice. Isso pode ser feito por meio de uma consulta à tabela `user_indexes` pela quantidade de chaves distintas (atributo `distinct_keys`) de uma determinada tabela.

Fazendo a comparação entre esse número de chaves distintas e o número de linhas da tabela, pode-se determinar a seletividade de um índice. Quanto maior a seletividade, melhor seria o índice para retornar pequenas quantidades de dados, ou seja, quanto mais seletivo for o índice, menor a quantidade de linhas que serão retornadas para cada valor distinto. A seletividade de um índice pode ser melhorada por intermédio da criação de índices compostos ou concatenados, como também são chamados. O

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

exemplo a seguir mostra a análise feita na tabela `'EMPLOYEES'`, com o intuito de verificar a seletividade dos índices definidos para ela. Esta operação foi realizada através da ferramenta SQL*PLUS.

```
SQL> analyze table employees estimate statistics;

Tabela analisada.

SQL> select table_name, index_name, distinct_keys
  2  from   user_indexes
  3  where  table_name = 'EMPLOYEES';

TABLE_NAME              INDEX_NAME                DISTINCT_KEYS
----------------------- ------------------------- -------------
EMPLOYEES               EMP_EMAIL_UK                       1075
EMPLOYEES               EMP_EMP_ID_PK                      1075
EMPLOYEES               EMP_DEPARTMENT_IX                    11
EMPLOYEES               EMP_JOB_IX                           19
EMPLOYEES               EMP_MANAGER_IX                       21
EMPLOYEES               EMP_NAME_IX                        1075

6 linhas selecionadas.

SQL> select count(*) from employees;

  COUNT(*)
----------
      1075

SQL>
```

Esse exemplo demonstra que a seletividade do índice `emp_emp_id_pk`, chave primária da tabela `employees`, é de 100%, pois, para 1075 registros contidos na tabela, há 1075 chaves distintas para este índice. Já o índice `emp_department_ix` possui uma seletividade baixa com apenas 11 chaves distintas.

Para ficar mais claro o conceito de seletividade, serão demonstrados exemplos de instruções SQL que foram submetidas a uma grande quantidade de linhas. Para esses exemplos, a sessão

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

foi configurada com modo baseado em regras. Foi escolhido dessa forma para que outros fatores, referentes às informações estatísticas armazenadas no banco, não influenciem o otimizador. O intuito deste exemplo é demonstrar que nem sempre o uso do índice é o melhor caminho.

A instrução SQL a seguir tem o objetivo de buscar o primeiro e o último nome de um determinado empregado cujo id seja igual a 800 e também os locais dos últimos 5 dias trabalhados. Neste exemplo, a tabela `timesheet` possui 1 milhão de registros. Os atributos `employee_id` das tabelas `employees` e `timesheet` são chaves primárias das tabelas, respectivamente. A chave primária da tabela `timesheet` é uma chave primária composta, juntamente com o campo `start_date`. Portanto, sabendo que, ao criar uma chave primária no Oracle, automaticamente é criado um índice para o mesmo campo, fica fácil prever que, muito provavelmente, o otimizador utilizará índices para tentar acessar tais tabelas. A instrução a seguir foi executada duas vezes, uma vez utilizando os índices atribuídos às colunas `employee_id`, acessando tais tabelas através do índice, e outra vez acessando as tabelas por completo. Para esta última, os índices foram invalidados para que o otimizador não os utilizasse.

Passos realizados antes da execução da instrução SQL para garantir o uso baseado em regras:

```
SQL> alter session set optimizer_mode = 'RULE';

Sessão alterada.

SQL> analyze table employees delete statistics;

Tabela analisada.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> analyze table locations delete statistics;

Tabela analisada.

SQL> analyze table timesheet delete statistics;

Tabela analisada.

SQL> alter session set sql_trace = TRUE;

Sessão alterada.

SQL> select name, value from v$parameter where name = 'optimizer_
mode';

NAME                      VALUE
------------------------- ------------------------------------
optimizer_mode            RULE

SQL>
```

Vamos ver o resultado da instrução SQL executada com acesso através dos índices. Tenha atenção ao resultado das colunas `elapsed` e `query`:

```
SQL> select first_name, last_name, l.city
  2  from  (
  3         select *
  4         from  (
  5                select e.first_name
  6                      ,e.last_name
  7                      ,e.employee_id
  8                      ,t.location_id
  9                      ,t.start_date
 10                from   employees e, timesheet t
 11                where  e.employee_id = 800
 12                and    e.employee_id = t.employee_id
 13                order  by t.start_date desc
 14               )
 15         where rownum < 6
 16        ) emp_loc
 17       ,locations l
 18  where emp_loc.location_id = l.location_id
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
19  /
```

Resultado gerado pelo `TKPROF` (utilitário para formatação de arquivos de Trace, contendo estatísticas, que será visto mais adiante):

```
Resultado gerado pelo TKPROF:

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0        290          0           5
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          0        290          0           5

Misses in library cache during parse: 0
Optimizer mode: RULE
Parsing user id: 41  (TTUN1)
```

Figura 4.2: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img01code.txt

Plano de execução:

```
Plano de execução:

Rows     Execution Plan
-------  -------------------------------------------------------
      0  SELECT STATEMENT   MODE: RULE
      5   NESTED LOOPS
      5    VIEW
      5     COUNT (STOPKEY)
      5      VIEW
      5       SORT (ORDER BY STOPKEY)
    284        NESTED LOOPS
      1         TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES' (TABLE)
      1          INDEX (UNIQUE SCAN) OF 'EMP_EMP_ID_PK' (INDEX (UNIQUE))
    284         TABLE ACCESS (BY INDEX ROWID) OF 'TIMESHEET' (TABLE)
    284          INDEX (RANGE SCAN) OF 'TIMESHEET_EMPID_STRDATE_PK' (INDEX (UNIQUE))
      5    TABLE ACCESS (BY INDEX ROWID) OF 'LOCATIONS' (TABLE)
      5     INDEX (UNIQUE SCAN) OF 'LOC_ID_PK' (INDEX (UNIQUE))
```

Figura 4.3: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img02code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Agora vamos ver o resultado da instrução SQL executada com acesso completo à tabela, não utilizando índices:

```
SQL> select first_name, last_name, l.city
  2  from  (
  3         select *
  4         from  (
  5                select e.first_name
  6                      ,e.last_name
  7                      ,e.employee_id
  8                      ,t.location_id
  9                      ,t.start_date
 10                from   employees e, timesheet t
 11                where  e.employee_id+0 = 800
 12                and    e.employee_id = t.employee_id+0
 13                order  by t.start_date desc
 14               )
 15         where rownum < 6
 16        ) emp_loc
 17       ,locations l
 18  where emp_loc.location_id = l.location_id
```

Segue o resultado gerado pelo `TKPROF` :

```
Segue o resultado gerado pelo TKPROF:

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      4.53       4.59          0    1004973          0           5
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      4.53       4.59          0    1004973          0           5

Misses in library cache during parse: 0
Optimizer mode: RULE
Parsing user id: 41  (TTUN1)
```

Figura 4.4: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img03code.txt

Plano de execução:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plano de Execução:

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: RULE
      5   NESTED LOOPS
      5    VIEW
      5     COUNT (STOPKEY)
      5      VIEW
      5       SORT (ORDER BY STOPKEY)
    284        NESTED LOOPS
1000000         TABLE ACCESS (FULL) OF 'TIMESHEET' (TABLE)
    284         TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES' (TABLE)
    284          INDEX (UNIQUE SCAN) OF 'EMP_EMP_ID_PK' (INDEX (UNIQUE))
      5    TABLE ACCESS (BY INDEX ROWID) OF 'LOCATIONS' (TABLE)
      5     INDEX (UNIQUE SCAN) OF 'LOC_ID_PK' (INDEX (UNIQUE))
```

Figura 4.5: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img04code.txt

Podemos ver que, através dos valores da coluna `elapsed` (tempo gasto para a execução da consulta), gerada pelo `TKPROF` , o acesso aos dados por meio do índice, nesse caso, foi muito mais rápido do que por uma leitura completa na tabela, pois o tempo sem uso do índice foi de um pouco mais de 4 segundos. Já com o uso do índice, o tempo foi menos de 0.00 (o resultado aqui é mostrado apenas com duas casas decimais). Veja a comparação dos tempos, com e sem o uso dos índices.

Pelo acesso através de índices:

```
elapsed
----------
0.00
0.00
0.00
----------
0.00      (total da duração da consulta em segundos)
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Pelo acesso sem a utilização dos índices:

```
elapsed
----------
 0.00
 0.00
 4.59
----------
 4.59     (total da duração da consulta em segundos)
```

Você pode estar pensando que o tempo gasto pela primeira instrução foi de zero, ou seja, não gastou tempo algum para realizar. Entretanto, embora o tempo tenha aparecido zerado, a instrução teve um tempo de execução. O tempo que é demonstrado no `TKPROF` está demonstrado em segundos e centésimos de segundos, portanto, se uma instrução levou menos de um centésimo de segundo para executar, os valores aparecerão zerados.

O próximo exemplo de execução busca 100 mil registros (10% dos registros) utilizando o acesso com e sem o índice.

Veja a instrução SQL executada com acesso através do índice:

```
select /*+ RULE */ 'Boa Tarde ' || first_name || ' ' || last_name
from   employees
where  employee_id <= 100000;
```

O resultado gerado pelo `TKPROF` :

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
O resultado gerado pelo TKPROF:

call     count       cpu    elapsed       disk      query    current        rows
-------  ------ -------- ---------- ---------- ---------- ----------  --------
Parse        1     0.01       0.00          0          0          0           0
Execute      1     0.00       0.00          0          0          0           0
Fetch     6668     5.40       5.84         28      14430          0      100000
-------  ------ -------- ---------- ---------- ---------- ----------  --------
total     6670     5.41       5.85         28      14430          0      100000

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 61
```

Figura 4.6: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img05code.txt

Agora a instrução SQL executada com acesso completo à tabela, não utilizando o índice:

```
select /*+ RULE */ 'Boa Tarde ' || first_name || ' ' || last_name
from   employees
where  employee_id <= 100000;
```

O resultado gerado pelo `TKPROF` foi:

```
O resultado gerado pelo TKPROF foi:

Call       count       cpu    elapsed       disk      query    current        rows
-------  ------ -------- --------- ---------- ---------- ---------  --------
Parse        1      0.07      0.07          0          1          0           0
Execute      1      0.00      0.00          0          0          0           0
Fetch     6668     10.01     32.49       8759      16489          0      100000
-------  ------ -------- --------- ---------- ---------- ---------  --------
total     6670     10.08     32.56       8759      16490          0      100000

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 61
```

Figura 4.7: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img06code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Ainda que a busca pelos dados tenha sido de 10% dos registros da tabela, o acesso através do índice foi muito mais rápido. Uma das causas para isso é a alta seletividade deste índice. Já quando o retorno é de 50%, a situação se reverte.

Segue a instrução SQL executada com acesso através do índice buscando 50% dos registros:

```
select /*+ RULE */ 'Boa Tarde ' || first_name || ' ' || last_name
from   employees
where  employees_id <= 500000;
```

O resultado gerado pelo `TKPROF` :

```
O resultado gerado pelo TKPROF:

call     count    cpu     elapsed       disk      query     current    rows
-------  ------  -------- ---------- ---------- ----------  --------   ----
Parse        1     0.10     0.12          0          1          0         0
Execute      1     0.00     0.00          0          0          0         0
Fetch    33335    38.73   116.44       5143      72482          0    500000
-------  ------ --------- ---------- ---------- ----------  -------- ------
total    33337    38.83   116.56       5143      72483          0    500000

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 61
```

Figura 4.8: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img07code.txt

Agora veja a instrução SQL executada com acesso completo à tabela, não utilizando o índice:

```
select /*+ RULE */ 'Boa Tarde ' || first_name || ' ' || last_name
from   employees
where  employees_id <= 500000;
```

Veja o resultado gerado pelo `TKPROF` :

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Veja o resultado gerado pelo TKPROF:

call     count       cpu    elapsed       disk      query    current        rows
-------  ------ -------- ---------- ---------- ---------- --------- --------
Parse        1     0.00       0.00          0          0         0           0
Execute      1     0.00       0.00          0          0         0           0
Fetch    33335    23.46      46.39       8620      43145         0      500000
-------  ------ -------- ---------- ---------- ---------- --------- --------
total    33337    23.46      46.39       8620      43145         0      500000

Misses in library cache during parse: 1
Optimizer goal: RULE
Parsing user id: 61
```

Figura 4.9: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap04img08code.txt

Embora um índice tenha um alto grau de seletividade, a busca por muitos registros pode tornar seu uso inviável. Por outro lado, se a busca for pequena, o retorno dos dados pode ser feito de forma mais rápida. É preciso lembrar que outros fatores, como CPU, leituras em disco e em memória, devem ser considerados e avaliados caso a caso. O intuito desses exemplos foi apenas demonstrar, com relação a tempo, quanto o otimizador poderia gastar para retornar tais linhas, considerando cenários diferentes com o uso de índices, sem levar em conta outros fatores.

É de fundamental importância que as tabelas e índices estejam sempre com suas estatísticas atualizadas. No entanto, mais importante ainda é que os programas que as acessam estejam escritos da forma mais adequada possível. Pouco adiantará ter informações atualizadas no banco de dados caso o programa não tenha sido escrito da melhor forma para utilizá-las. Vale lembrar que, dependendo do modo de otimização com o qual o banco de dados está configurado, por exemplo, o modo baseado em regras, a forma como o programa foi escrito terá um peso muito grande na

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

maneira como o otimizador o executará. Agora vamos conhecer um pouco sobre os tipos de índices no Oracle.

4.1 TIPOS DE ÍNDICES

No Oracle, os índices são classificados por tipos. Falaremos dos tipos `B-tree` e `Bitmap`. O principal objetivo dessa abordagem é demonstrar os conceitos por trás deles e características quanto aos seus funcionamentos. A escolha por esses dois tipos de índices se deu porque o índice `B-tree` é o mais utilizado, e o Bitmap é uma tecnologia mais atual que traz recursos bastante interessantes.

Índices B-tree

Os índices `B-tree` são os índices de uso geral no Oracle, ou seja, eles são os tipos de índices padrões utilizados quando se fala em criação de índices. Os índices `B-tree` podem ser índices de única coluna (simples) ou índices compostos/concatenados (multicolunas). Índices `B-tree` podem ter até 32 colunas em sua definição. A seguir, algumas características principais deste tipo de índice:

- O índice `B-tree` tem uma estrutura de árvore hierárquica;
- Árvores balanceadas rapidamente localizam os valores e chaves;
- Os endereços fornecidos pelos `rowid` provêm acesso direto às linhas das tabelas;
- Consultas que usam somente colunas indexadas podem ser resolvidas no índice;
- Pesquisas em índices são uma alternativa para pesquisas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

sequenciais em tabelas, já que podem reduzir o I/O de disco;

- Os índices são usados e mantidos, automaticamente, pelo Oracle.

A figura a seguir mostra a representação da estrutura de um índice `B-tree` . Como podemos ver, esse índice possui uma altura binária de três níveis. O Oracle precisará percorrer dois níveis de bloco, chamados de *Branch Blocks*, para chegar até o último nível de bloco, chamado *Leaf Blocks*, contendo a `rowid` . Dentro de cada `Branch Block` , existem linhas contendo o ID que indica o próximo ID de bloco dentro da cadeia. Para o nível dois, cada *Branch Block* conterá o ID de bloco indicando como chegar ao nível 3 (*Leaf Blocks*).

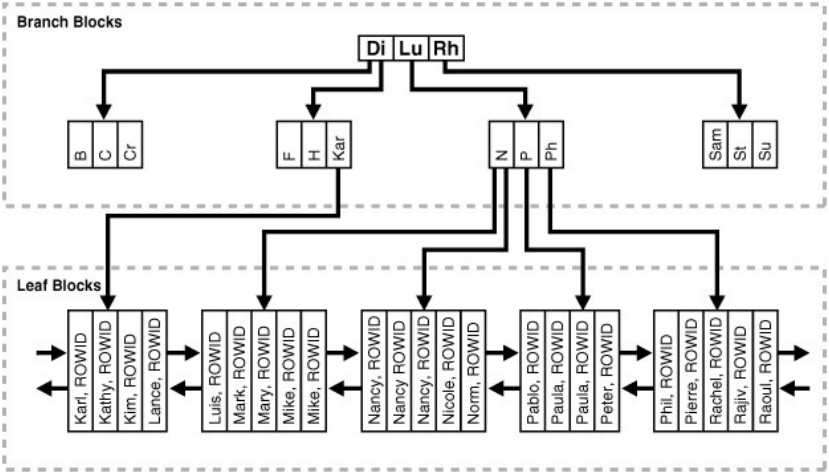

Figura 4.10: Estrutura do índice B-tree. Fonte: Getting a Handle on Indexes By Steve Callan http://www.dba-oracle.com/t_callan_handle_on_indexes.htm

Algumas vantagens relacionadas ao uso do índice `B-tree` :

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Todos os `Leaf Blocks` (Blocos Folha) da árvore estão na mesma profundidade; assim, na recuperação de todo o registro em qualquer lugar dentro das chaves do índice, levará aproximadamente a mesma quantidade de tempo;
- Os índices `B-tree` permanecem automaticamente balanceados;
- Os índices `B-tree` fornecem um desempenho excelente para buscas exatas;
- Inserções, atualizações e exclusões são eficientes, mantendo uma recuperação rápida com relação a ordenação das chaves de índice;
- O desempenho do índice `B-tree` se adequa a tamanhos diferentes de tabelas e, se bem estruturados, não degradam a performance à medida que o tamanho dessas tabelas cresce.

Desvantagens relacionadas ao uso do índice `B-tree` :

- Não é recomendado quando o retorno das linhas de uma tabela for maior que 30% do total de registros;
- Em caso de inserções em sequência, pode ser necessário fazer o balanceamento da árvore do índice para evitar a demora na busca pelos dados;
- Seu uso pode não ser indicado quando instruções SQL possuírem funções de grupo (`group by`) em sua declaração.

A criação de um índice `B-tree` é uma tarefa muito simples. Veja a seguir exemplos da criação de alguns índices do tipo `B-tree` :

Sintaxe:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
create index nome_index on tabela (coluna1, colunan, ....);
```

Exemplo de índice simples:

```
create index job_idx1 on emp (job);
```

Nesse exemplo, está sendo mostrada a criação de um índice para a coluna `job` da tabela `emp`. Trata-se de um índice simples, pois possui em sua definição apenas uma coluna.

Exemplo de índice único:

```
create unique index empno_un1 on emp (empno);
```

Já nesse exemplo, está sendo mostrada a criação de um índice único para a coluna `empno` da tabela `emp`. Trata-se de um índice único, indicando que os valores desta coluna e, consequentemente, do índice, deverão ser únicos.

Exemplo de índice composto, com mais de uma coluna:

```
create index job_deptno_in1 on emp (job, deptno);
```

Nesse exemplo, está sendo mostrada a criação de um índice composto para as colunas `job` e `deptno` da tabela `emp`. Trata-se de um índice composto, pois possui em sua definição mais de uma coluna.

Garantindo a melhor performance de um índice B-tree

Dentre as desvantagens do uso do índice `B-tree`, citadas anteriormente, existe uma que pode requerer um pouco mais de atenção, pois foge da área de desenvolvimento. Ao contrário dos cenários relacionados ao percentual de registros retornados em uma instrução SQL e a questão do uso de funções agregadas, que

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

são facilmente identificados e podem ser trabalhados pelo desenvolvedor de aplicações, o balanceamento da árvore de índice requer, na grande maioria das vezes, o apoio de um DBA para analisar esses índices e verificar se eles necessitam de um `rebuild`, por exemplo, para que se tornem mais eficazes. Caso um índice tenha todos os requisitos para ser um bom índice, mas mesmo assim não estiver satisfazendo as condições de performance, converse com o DBA responsável pelo banco de dados sobre o balanceamento de índices. Quem sabe não há algo a ser melhorado neste sentido?

Índices Bitmap

Em um índice `Bitmap`, o Oracle cria um `bitmap` para cada chave distinta e depois armazena as `rowids` associadas ao valor de chave como um `bitmap`. No caso de um índice comum, esse processo é realizado por meio do armazenamento de uma lista de `rowids` para cada chave de índice que corresponde às linhas da tabela com aquele valor de chave, pois, no Oracle, cada chave é armazenada juntamente com o `rowid` correspondente. Dessa forma, no índice `B-tree` é armazenado uma lista de `rowids`, enquanto no `Bitmap`, ele converte isso em um mapa de `bits`.

Um índice `Bitmap` também é organizado como uma estrutura de `B-tree`. Contudo, os *Leaf Blocks* de um índice `Bitmap` armazenam um `bitmap` para cada valor de chave em vez do `rowid`. Em vez disso, na prática, cada `bit` corresponde a um possível `rowid`. Se o `bit` estiver ligado (sinalizado com valor 1), ou marcado, como também podemos nos referir, isso indica que a linha com o `rowid` correspondente contém o valor da chave. A diferença está na forma como ele guarda a informação sobre a

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

chave do índice, que é diferente para cada tipo. Basicamente, o funcionamento do índice `Bitmap` acontece da seguinte forma:

- Um mapa de `bits` para cada valor de chave é usado em vez de uma lista de `rowids` . Cada `bit` corresponde a uma possível `rowid` , e se o `bit` estiver marcado, isso significa que a linha com o `rowid` correspondente contém o valor-chave.
- Uma função de mapeamento é usada para converter a posição do `bit` no valor correspondente do `rowid` , para que o índice forneça a mesma funcionalidade que um índice comum mesmo usando uma representação diferente.

A figura a seguir nos demonstra como seria a estrutura de um índice `Bitmap` . Como pode ser visto, a chave de índice é formada por apenas uma coluna (chamada color, no exemplo), e a primeira entrada na árvore do índice tem como valor: amarelo, a segunda: verde, a terceira: vermelho, e por último, a quarta: cinza (representando valores nulos). Cada entrada está associada a um mapa de `bits` indicando quais `rowids` correspondem ao valor de chave.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

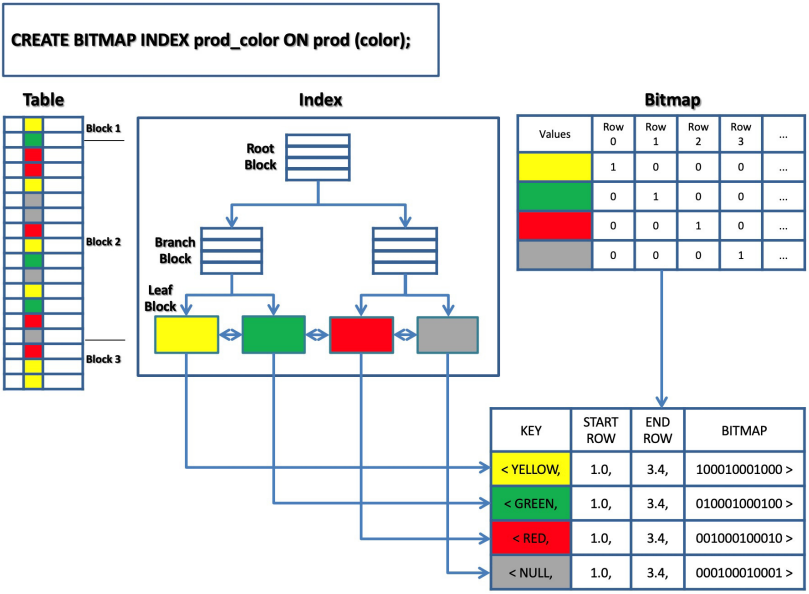

Figura 4.11: Funcionamento do índice bitmap. Adaptado de: Oracle-btree and bitmap index https://www.programmersought.com/article/8668116234/

Para tornar esse conceito mais claro, a figura a seguir demonstra de forma simbólica como acontece essa interpretação e associação dos registros de uma tabela, e a criação do mapa de `bits` de um índice `Bitmap`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

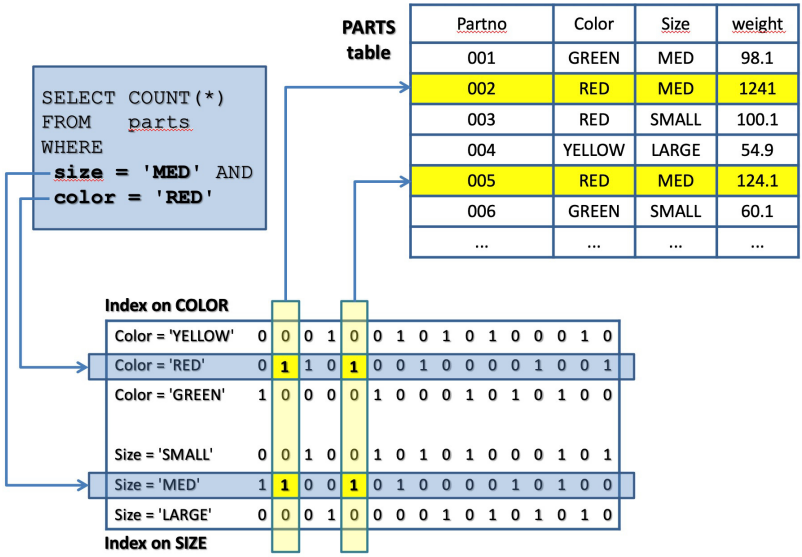

Figura 4.12: Funcionamento do índice bitmap. Adaptado de: Oracle-btree and bitmap index https://www.programmersought.com/article/8668116234/

Note que nessa imagem temos uma tabela chamada `PARTS` (`PARTS` table) que representa informações sobre peças com suas cores, tamanhos e pesos. Logo ao lado, temos um comando `select` cujo resultado deverá ser a contagem de peças cujo tamanho seja igual a `'med'` , e cuja cor seja igual a `'RED'` . Abaixo da instrução SQL temos uma representação de como o Oracle montaria esse mapa de `bits` e como ele localizaria esses registros, conforme critérios do `select` . Note que cada 0 e 1 representa um registro da tabela para cada valor distinto da chave de índice, sendo que ele será 0 caso não corresponda ao valor, e 1 caso corresponda.

Nesse exemplo, temos índices `Bitmap` criados tanto para a

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

coluna `color` , quanto para a coluna `size` . No final, aqueles registros que satisfazem aos dois critérios, ou seja, aqueles que ao se combinarem possuírem valor 1 (ligados), serão retornados como resultado. Note também que diversos registros possuem valor 1 na coluna `color` , mas se for visto na coluna `size` , seus valores são 0, e vice-versa, dessa forma esses registros não satisfazem aos critérios de seleção. Após combinarem os valores, com base nos critérios da cláusula `where` , os registros são localizados na tabela e retornados como resultado da instrução SQL. Como resultado desta instrução, temos as peças 002 e 005 (coluna `partno`), que atendem ao critério da instrução SQL. Agora vamos ver quando melhor utilizá-los, suas vantagens e desvantagens.

Pode-se usar o índice `Bitmap` como uma alternativa ao índice `B-tree` quando:

- As tabelas são muito grandes (milhões de linhas);
- As tabelas possuem uma cardinalidade baixa, isto é, a coluna (ou colunas) na qual o índice seria baseado possui um número pequeno de valores distintos (exemplo: sexo, estado civil, UF);
- As instruções (mais usadas) podem ter índices `Bitmap` em todas as colunas que normalmente aparecem nas cláusulas `where` ;
- As instruções possuem condições na cláusula que são satisfeitas com milhares de linhas.

Vantagens no uso de índice `Bitmap` :

- Oferece recuperação rápida para colunas que têm apenas poucos valores distintos e que são frequentemente

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

consultadas juntas. Contudo, `FULL TABLE SCANS` serão mais eficientes se as colunas forem consultadas individualmente;

- São especialmente aplicáveis para consulta em grandes tabelas;
- Podem ser usados em qualquer ordem e em qualquer combinação, assim são mais flexíveis que um índice `B-tree` , o qual requer o uso de pelo menos a primeira coluna do índice, em caso de índice composto;
- Se usado apropriadamente, é bastante compacto, bem mais que o índice `B-tree` , pois ocupam menos espaço;
- As colunas deste índice devem ser de baixa cardinalidade, ou seja, as colunas devem ter poucos valores distintos. Caso contrário, muitos `Bitmaps` devem ser construídos e mantidos. Logo, podem ocupar mais espaço também;
- São bastante eficientes no caso de operações de merge de índices (junção), pois operações `bit a bit` são bastante rápidas.

Desvantagens no uso de índice `Bitmap` :

- O Oracle não é capaz de bloquear um único `bit` e, consequentemente, bloqueios para `bitmapped index` são feitos para o bloco. Dado que um grande número de linhas (ou `bits`) pode estar em um único bloco, muitas linhas serão locadas. Isso torna o índice `Bitmap` inadequado para aplicações com um número moderado de transações;
- O índice `Bitmap` não é adequado para consultas por faixas de valores, ou para colunas com grande número de valores distintos. Se for o caso, deve-se usar índices `B-tree` para essas colunas.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Um exemplo de utilização de um índice `Bitmap` seria no caso da coluna referente ao gênero em uma tabela de pessoas. Como os possíveis valores poderiam ser `masculino` e `feminino`, por exemplo, a cardinalidade seria de apenas dois valores exclusivos e tornaria esta coluna uma forte candidata a um índice do tipo `Bitmap`. Isso reforça a ideia de que esse tipo de índice deve ser utilizado em colunas que possuem baixa cardinalidade. Veja a seguir alguns exemplos da criação de índices `Bitmap`:

Sintaxe:

```
create bitmap index nome_index on tabela (coluna1, colunan, ....)
```

Exemplo:

Criação do índice `id_genero_idx` para a coluna `id_genero` da tabela `pessoas`.

```
create bitmap index id_genero_idx on pessoas (id_genero);
```

Criação do índice `id_region_idx` para a coluna `region_id` da tabela `countries`.

```
create bitmap index id_region_idx on countries (region_id);
```

Criação dos índices `size_idx` e `color_idx` para as colunas `size` e `color`, respectivamente, da tabela `parts` (vista nas imagens anteriores).

```
create bitmap index size_idx on parts (size);

create bitmap index color_idx on parts (color);
```

Agora que já conhecemos um pouco sobre esses dois tipos de índices, veja um pequeno comparativo entre eles para que você tenha ideia de como utilizá-los de forma adequada, aproveitando o

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

melhor de cada tipo e conhecendo suas principais características.

Índices `B-tree`	Índices `Bitmap`
Ideal para colunas de alta cardinalidade.	Ideal para colunas de baixa cardinalidade.
O custo para atualizações em chaves de índice é relativamente baixo.	O custo para atualizações em colunas indexadas pode ser muito alto.
Não é eficiente para consultas usando o operador lógico `OR` .	Eficiente para consultas usando o operador lógico `OR` .
Usado principalmente onde há um grande volume de transações (OLTP: *On-Line Transaction Processing*). Ambiente transacional.	Muito útil para *Data Warehousing* (ambiente cujo principal foco é a seleção/recuperação de dados). Ambiente gerencial.

Se você necessitar criar índices para uma determinada tabela que já contenha dados e ainda está com dúvidas com relação a quais colunas seriam as mais indicadas, segue uma instrução SQL que vai ajudá-lo a determinar as colunas candidatas a composição de um índice.

```
SQL> select tcst_1.column_name
  2        ,tcst_1.num_distinct
  3        ,tcst_1.num_nulls
  4        ,(select num_rows
  5          from   dba_tables t
  6          where  t.table_name = tcst_1.table_name
  7          and    t.owner = tcst_1.owner) as tot_linhas
  8        ,((tcst_1.num_distinct * 100) / (select num_rows
  9                                         from   dba_tables t
 10                                         where  t.table_name = tcs
t_1.table_name
 11                                         and    t.owner = tcst_1.o
wner)) as "% total"
 12        ,tcst_1.last_analyzed
 13  from   dba_tab_col_statistics tcst_1
 14  where  tcst_1.table_name    = 'EMPLOYEES'
 15  AND    tcst_1.last_analyzed = (select max(tcst_2.last_analyz
ed)
 16                                 from   dba_tab_col_statistics
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
 tcst_2
 17                                where  tcst_2.table_name = tc
st_1.table_name)
 18  order  by tcst_1.column_name
 19  /

COLUMN_NAME          NUM_DISTINCT  NUM_NULLS TOT_LINHAS     % tot
al  LAST_ANA
-------------------- ------------ ---------- ---------- ---------
-- ---------
COMMISSION_PCT                  7         72        107  6,542056
07  05/04/16
DEPARTMENT_ID                  11          1        107  10,28037
38  05/04/16
EMAIL                         107          0        107         1
00  05/04/16
EMPLOYEE_ID                   107          0        107         1
00  05/04/16
FIRST_NAME                     91          0        107   85,0467
29  05/04/16
HIRE_DATE                      98          0        107   91,5887
85  05/04/16
JOB_ID                         19          0        107  17,75700
93  05/04/16
LAST_NAME                     102          0        107  95,32710
28  05/04/16
MANAGER_ID                     18          1        107  16,82242
99  05/04/16
PHONE_NUMBER                  107          0        107         1
00  05/04/16
SALARY                         57          0        107   53,2710
28  05/04/16

11 linhas selecionadas.

SQL>
```

Com base no dicionário de dados e nas estatísticas colhidas para uma tabela em questão, nesse exemplo, a tabela `employees` , essa instrução busca informações que ajudará você a determinar as colunas mais adequadas para a criação de índices. Além das informações referentes ao total de linhas (`tot_linhas`),

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

percentual de linhas referente ao total (`% total`) e a data da última estatística colhida (`last_analyzed`), este `select` traz duas informações muito importantes, o `num_distinct` e `num_nulls` .

- `num_distinct` : indica o número de valores distintos em uma coluna. Caso o valor desta coluna seja muito baixo, por exemplo, refletir menos de 5% do total de registros da tabela, isso indica uma seletividade baixa, podendo ser considerada fraca, e não ser uma boa candidata para a criação de um índice `B-tree` . Por outro lado, se nossa intenção é criar um índice `Bitmap` , esta pode ser uma alternativa para a criação do índice, pois, conforme vimos anteriormente, colunas com poucos valores distintos são ótimas candidatas a índice `Bitmap` . Agora, se o número de valores distintos for alto em relação ao total de registros da tabela, por exemplo, cerca de 50% do total de registros, isso pode indicar que a coluna é uma forte candidata à criação de um índice `B-tree` . Já para a criação de um índice `Bitmap` , não será uma boa alternativa.
- `num_nulls` : indica a quantidade de valores nulos para uma determinada coluna da tabela. Se uma coluna possuir poucos valores nulos, ela estará apta a ser uma boa candidata para a criação de um índice.

Como vimos, o tipo mais comum de índice é o `B-tree` . Vimos também que um índice pode ser simples, formado apenas por um coluna, ou pode ser composto, formado por mais de uma coluna, podendo chegar a trinta. Vimos também que a ordem das colunas, dispostas na definição do índice, faz diferença. O otimizador leva em consideração estes e outros aspectos para

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

decidir ou não pelo uso de tal índice. Dependendo de como dispomos as colunas que fazem parte de um índice, nas instruções SQL, o otimizador poderá utilizá-lo de forma diferente ou nem utilizá-lo.

Por exemplo, se temos um índice composto de três colunas, mas na nossa instrução SQL estamos utilizando apenas duas destas colunas, o otimizador, mesmo assim, poderá utilizá-lo se ele achar que ainda assim terá um melhor desempenho. Por outro lado, dependendo de quais colunas desse índice estivermos utilizando, ele poderá não o considerar e, se não houver outra opção, poderá fazer uma varredura completa na tabela (`FULL TABLE SCAN`), o que poderá causar uma demora não esperada.

Agora que já vimos o quanto os índices podem influenciar a busca pelos dados, vamos conhecer os meios de acesso e junções de tabelas, e entender melhor como o otimizador faz para a recuperar os dados quando utilizamos uma ou mais tabelas em nossos programas.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 5

MÉTODOS DE JUNÇÃO DE TABELAS

Quando estudamos sobre a linguagem SQL, vemos como são realizadas as junções entre tabelas em um banco de dados. Essas junções são referentes à forma como relacionamos as tabelas em nossas instruções SQL, ou seja, como devem ser aplicadas e escritas respeitando a sintaxe da linguagem para que os registros sejam retornados de acordo com os nossos objetivos. Na linguagem SQL, por exemplo, são vistas junções como o tipo `inner join`, onde os registros são retornados caso haja igualdade de valores entre duas colunas ou mais colunas definidas no relacionamento entre mais de uma tabela. Ou, no caso do `outer left join`, que indica que mesmo não havendo valores correspondentes entre duas ou mais colunas, são retornados os registros da tabela da esquerda, ou ainda um relacionamento `outer right join`, onde os registros retornados são os da tabela da direita, mesmo que não exista correspondência na tabela da esquerda.

Pois bem, essas junções são definidas pela pessoa desenvolvedora no momento em que são escritas as instruções, e são, por assim dizer, visualizadas de forma explícita. Já os métodos de junção de tabelas, dos quais falaremos agora, possuem outro papel. Sua finalidade se caracteriza pelo método que será usado

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

pelo otimizador para corresponder ao tipo de junção que estamos solicitando. Ou seja, estamos falando da forma como o otimizador vai realizar essas junções internamente para melhor retornar as informações.

Desde o Oracle 6, o otimizador vem utilizando métodos de junção de tabelas com o intuito de unir linhas através de caminhos diferentes, escolhendo uma ou mais formas de acordo com as características de cada cenário. No geral, são quatro os métodos de junção no Oracle: *NESTED LOOPS JOINS*, *HASH JOINS*, *SORT MERGE JOINS* e *CARTESIAN JOINS*. Cada método é composto por recursos exclusivos e também algumas limitações e características.

Pode haver situações em que problemas de performance podem estar relacionados à maneira como o otimizador escolhe esses métodos no momento de recuperar os dados das tabelas em uma instrução SQL. Portanto é muito importante conhecer cada método e entender o porquê da sua escolha pelo otimizador. Vale lembrar que os métodos escolhidos pelo otimizador têm muito a ver com a maneira como sua instrução SQL foi escrita. Saber desses conceitos e de como o otimizador faz essas escolhas vai lhe dar subsídios para que você saiba avaliar sua instrução SQL e, com isso, alterá-la com o intuito de modificar essas escolhas e resolver problemas de performance que por ventura estejam ocorrendo devido ao plano atual. Para isso, é de fundamental importância que você tenha conhecimento sobre alguns pontos:

- Que tabela controlará sua instrução SQL e quando cada tabela será acessada, dado o caminho escolhido para ela? Quais são os caminhos alternativos?

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Quais são as possibilidades de junção do Oracle? Lembre-se que cada possibilidade de junção para o Oracle pode gerar diferentes resultados, dependendo da ordem de junção, da seletividade dos índices e da memória disponível.
- Quais índices estão disponíveis e qual a seletividade de tais índices? A seletividade de um índice pode não apenas fazer com que o otimizador use ou não tal índice, mas também pode mudar o modo como a instrução é vista pelo otimizador, determinando o uso ou não de outros índices em tal instrução.
- Quais `hints` oferecem caminhos alternativos e quais `hints` possibilitam ou impedem que um índice seja usado? Os `hints` podem mudar a ordem de controle das tabelas, alteram o modo como o Oracle realiza junções e podem determinar quais índices ele usará ou não.
- Que versão do Oracle você está usando? As opções variam dependendo da versão e release em que você está trabalhando. O otimizador também pode trabalhar de modo diferente dependendo da versão. Sempre consulte a documentação da versão do seu banco de dados.

Neste livro, abordaremos os seguintes métodos: *NESTED LOOPS*, *SORT MERGE* e *HASH*. Vamos falar um pouco de cada método de junção, mostrando exemplos de planos de execução em que os métodos aparecem e suas principais características. Mas antes disso vamos ver um pouco de como o otimizador trata esses métodos e os escolhe nas abordagens baseadas em Custo e em Regra.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

5.1 EXECUTANDO OPERAÇÕES DE JUNÇÃO NO MODO CBO

Para escolher um plano de execução para uma determinada instrução SQL que possua relacionamento entre vários objetos, o otimizador deve tomar uma série de decisões sobre quais caminhos e métodos utilizar. Essas decisões são tomadas com base em recursos e meios que estejam disponíveis no momento para tais objetos e precisam levar em conta que as escolhas estão relacionadas entre si.

Basicamente, para que o otimizador possa executar uma instrução SQL, envolvendo junções entre duas ou mais tabelas, ele precisará decidir por:

- Métodos de Acesso: para instruções SQL simples, o otimizador deve escolher um método de acesso para recuperar dados de cada tabela relacionada e pertencente à instrução.
- Método de Junção: para realizar as junções das tabelas, o Oracle deve escolher e executar uma operação de junção. Métodos de Junção incluem `NESTED LOOPS`, `SORT MERGE`, `CLUSTER`, `CARTESIAN` e `HASH`.
- Ordem da Junção: para executar uma instrução SQL que possua junção de mais de duas tabelas, o otimizador junta duas das tabelas primeiramente e, em seguida, junta o resultado à tabela seguinte. Esse processo continua até que todas as tabelas sejam relacionadas.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

5.2 ESCOLHENDO MÉTODOS DE JUNÇÃO NO MODO CBO

Como já sabemos, o otimizador vai gerar uma série de planos de execução para então decidir pelo plano que custará menos, ou seja, aquele que se demonstrar mais eficaz na recuperação das informações. Para isso, ele realiza diversos cálculos e algumas dezenas de processamentos. Tudo isso em um espaço de tempo muito rápido e antes de executar a instrução SQL propriamente dita. É nesse momento que ele consulta estatísticas e recursos disponíveis no banco de dados para determinar que caminho tomar. Umas dessas decisões é a escolha do método de junção. O otimizador estima o custo de cada método de junção e opta pelo método com o menor custo. Se uma junção retornar muitas linhas, ele então considera três fatores.

1) Uma junção `NESTED LOOPS` é ineficiente quando uma junção retorna uma grande quantidade de linhas (normalmente, mais de 10.000 linhas são consideradas um volume alto) e o otimizador pode optar por não utilizá-lo. O custo de uma junção `NESTED LOOPS` é calculado pela seguinte fórmula, considerando A e B, duas fontes de dados (tabelas ou resultado da junção de mais tabelas):

```
Custo = Custo de Acesso de A + (Custo de Acesso de B * Número de
Linhas de A)
```

2) Se você estiver usando o otimizador em modo CBO, então uma junção `HASH` pode ser um método mais eficiente quando o retorno for um grande número de linhas. O custo de uma junção `HASH` é calculado pela seguinte fórmula:

```
Custo = (Custo de Acesso de A * Número de Partições Hash de B) +
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Custo de Acesso de B
```

3) Se você estiver usando o RBO, modo baseado em Regras, uma junção do tipo `MERGE` pode ser uma junção mais eficiente quando o retorno dela for um grande número de linhas. O custo de uma junção `MERGE` é calculado pela seguinte fórmula:

```
Custo = Custo de acesso de A + Custo de Acesso de B + (Custo de O
rdenação de A + custo de Ordenação de B)
```

> **NOTA:** Quando os dados são pré-ordenados, não há custos de ordenação. Por exemplo, quando a ordenação informada na instrução SQL for a mesma ordenação do índice utilizado no plano de execução para o acesso aos dados.

5.3 ESCOLHENDO PLANOS DE EXECUÇÃO PARA TIPOS DE JUNÇÃO NO MODO CBO

Basicamente, como já falamos em algumas ocasiões, um plano de execução é constituído por formas de acesso utilizadas pelo otimizador para recuperar e retornar os dados em uma instrução SQL. Para que o otimizador monte esses planos, ele precisa conhecer como acessar, como unir o resultado deste acesso e qual a melhor ordem para fazer isso. Esses pontos são a receita para ele montar um plano de execução. Com essas três combinações, ele gerará diversos planos e encontrará no final aquele que custará menos para ser executado. Um dos fatores que influenciam na geração de um plano de execução está na escolha do melhor método após o otimizador ter escolhido um método de acesso. A seguir, veja algumas considerações que se aplicam às abordagens

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

baseadas em Custos (CBO) e Regras (RBO):

1. Ao executarmos uma instrução SQL que contenha mais de uma fonte de dados, o otimizador primeiramente verifica se essa junção trará um resultado contendo no máximo uma linha de retorno em cada uma delas, ou seja, se estamos falando de relacionamentos um para um (1:1). Ele consegue fazer essa verificação através das informações contidas no dicionário de dados, constatando ou não a existência de chaves primárias ou chaves únicas (índices únicos) que por ventura existam para os atributos das tabelas utilizadas para a junção. Se isso for verdade, o otimizador dá prioridade para essas tabelas, colocando-as em primeiro lugar na ordem de junção. Essa ordem à qual me refiro é aquela ordenação que envolve a forma como colocamos as tabelas e restrições nas cláusulas `from` e `where`, respectivamente. Lembre-se de que quando não estamos trabalhando em modo baseado em Regra ou forçando um `hint ORDERED`, por exemplo, o otimizador definirá qual a melhor ordem para realizar a junção dessas fontes de dados. Depois disso, ele otimiza a junção das demais tabelas da instrução.

2. Já para as junções que envolvem `outer join`, isto com relação à ordem de junção, a tabela que contém este operador deve vir após a outra tabela da condição na cláusula `from`. O otimizador não considera ordenações que violam esta regra. Da mesma forma, quando uma subconsulta for convertida em um `antijoin` ou `semijoin`, as tabelas da subconsulta devem vir depois dessas tabelas no bloco de consulta externo ao qual elas foram relacionadas ou correlacionadas. No entanto, `hash antijoins` e `semijoins` são capazes de substituir esta condição de ordenação em certas circunstâncias. Lembre-se destas regras

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

quando você estiver querendo forçar a ordem de execução das tabelas na sua instrução SQL.

> **NOTA:** Nas documentações mais recentes da Oracle se fala muito em `Row Source`, que em tradução livre seria "fonte de linhas" ou "fonte de dados". No entanto, podemos traduzir este conceito `Row Source`, como sendo tabelas, `views` ou resultados de alguma junção, por exemplo. Uma fonte de dados não necessariamente é uma tabela, pode também ser uma `view`. E quando falamos sobre os dados que processamos em uma junção, principalmente, em retornos processados pelo otimizador, vemos com frequência o termo `Row Source`.

Dando continuidade, o otimizador gera um conjunto de planos de execução, de acordo com possíveis ordens de junção, métodos de junção e métodos de acesso disponíveis, sempre se baseando na instrução SQL informada. Em seguida, estima o custo de cada plano e escolhe aquele que for gerar um menor custo. Esta estimativa é realizada da seguinte forma:

- O custo de uma operação `NESTED LOOPS` é baseado no custo de ler cada linha selecionada da tabela externa, e cada uma das linhas correspondentes da tabela interna em memória. Ele então estima estes custos usando as estatísticas colhidas do dicionário de dados. Como já vimos, a seletividade, o número de registros, as informações sobre blocos de dados e índices, são algumas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

das informações que o otimizador utiliza para chegar a um custo.

- Já o custo de uma junção `SORT MERGE` baseia-se, principalmente, no custo de ler todas as fontes de dados (tabelas e `views`) em memória e ordená-las. Também vimos que dependendo do tipo de ordenação disposto na instrução SQL, caso ela esteja em paridade com índices e atributos de junção, o otimizador poderá descartar o trabalho de ordenar, pois nesses casos, as entradas dos índices, ou seja, os valores, já estariam ordenados.
- Quanto ao custo de uma junção `HASH` , este é baseado em grande parte no custo da construção de uma tabela `hash` para uma das tabelas da junção, usando as linhas da outra tabela para testá-lo. Vale lembrar que este tipo de junção cria uma tabela `hash` em memória e a utiliza para fazer as associações com a segunda tabela da junção.

O otimizador também considera outros fatores ao determinar o custo de cada operação. Por exemplo:

- Uma `SORT AREA` menor, provavelmente, aumentará o custo para uma junção do tipo `SORT MERGE` porque a ordenação leva mais tempo de CPU e I/O em uma área com menor tamanho.
- É provável que uma quantidade maior de leituras `multiblock` diminua o custo de uma junção `SORT MERGE` em relação a uma junção `NESTED LOOPS` . Se um grande número de blocos sequenciais for lido do disco em uma única operação, então um índice da tabela interna em uma junção `NESTED LOOPS` pode ser menos eficiente para melhorar o desempenho em uma `FULL TABLE SCAN` . A

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

título de curiosidade, a contagem de leitura de `multiblocks` é especificada por parâmetro de inicialização. Para mais informações, consulte a documentação do banco de dados Oracle e também o DBA.

NOTA: `SORT AREA` é a quantidade máxima de memória que o Oracle usa para uma ordenação. Depois que a ordenação estiver concluída, mas antes que as linhas sejam retornadas, o Oracle libera toda a memória alocada para a ordenação, exceto o valor especificado no parâmetro. Depois que a última linha é retornada, o Oracle libera o restante da memória. Este valor pode ser configurado através de parâmetro.

NOTA: `I/O` significa entrada e saída de disco. O desempenho de muitos softwares pode ser limitado pela entrada/saída de disco (`I/O - Input/Output`). Muitas vezes, a atividade da CPU deve ser suspensa enquanto a atividade de `I/O` é concluída. O Oracle foi projetado para que o desempenho não seja limitado pela `I/O` , contudo, o ambiente como um todo deve estar ajustado adequadamente para que isto não seja o motivo de problemas, desde a parte de hardware até o software de banco de dados, passando pelo sistema operacional, tráfego de rede e outros.

A escolha do otimizador por determinadas ordens de junção

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

pode ser substituída com o uso do `hint ORDERED`. Se o `hint ORDERED` especificar uma ordem de junção que viola a regra para um `outer join` (visto, anteriormente), então o otimizador ignora o `hint` e escolhe a ordem com base na sua análise, com os dados que ele colher das estatísticas. Além disso, você pode não só influenciar o otimizador na escolha da ordem de junção, como também, influenciar na escolha por determinados métodos de junção utilizando `hints`, como já foi mencionado.

Agora vamos conhecer as características de cada método de junção e ver como funcionam.

5.4 NESTED LOOPS JOIN

Esse método funciona da seguinte forma: para cada linha de uma tabela, o otimizador localiza todas as linhas da outra tabela que satisfaçam a condição do `join` (utilizando os relacionamentos entre as tabelas). A escolha por qual tabela será lida primeiro é feita através de alguns critérios. O otimizador escolhe uma das tabelas como tabela externa ou tabela de entrada (também chamada de *Driving table* ou, em uma tradução livre, Tabela de Condução). A outra tabela é chamada de interna.

Falarei sobre as `Driving Tables` em um capítulo específico, pois, na minha opinião, este assunto é de grande importância. Todo o plano de execução começa por uma decisão muito importante por parte do otimizador, que é a escolha de qual tabela que será lida primeiro. É a partir dela que será definido o norte para a leitura das demais tabelas de uma instrução SQL.

No `NESTED LOOPS`, para cada linha encontrada na tabela

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

externa, o otimizador tenta localizar todas as linhas na tabela interna através do relacionamento entre elas, combinando os dados em par de linhas as quais satisfaçam as condições de pesquisa. Feito isso, retorna essas linhas como resultado da instrução. Esta escolha de qual tabela será a interna ou externa pode ser influenciada pela localização da tabela na cláusula `from` . Já falamos sobre como podemos influenciar o otimizador através da forma como escrevemos uma instrução SQL. Veja na figura a seguir como funciona esse tipo de método:

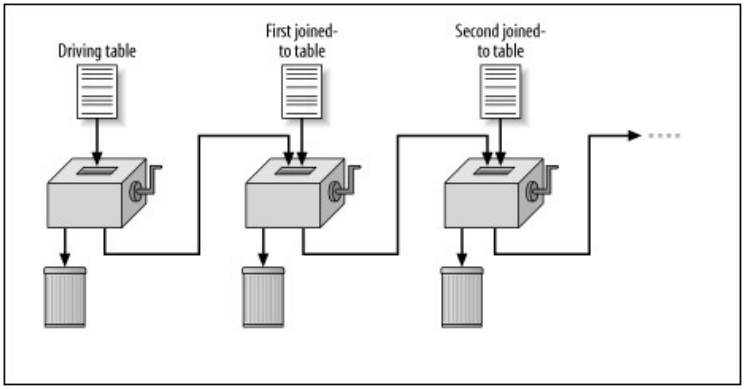

Figura 5.1: Fonte: SQL Tuning. Por Dan Tow. (https://books.google.com.br/books?hl=pt-BR&id=dOYPtfVqS_MC&q=Joins#v=onepage&q=Merge-Join&f=false)

Observe que o desenho demonstra, de forma simbólica, o funcionamento do método `NESTED LOOPS` , como sendo uma sequência de caixas, cada qual equipada com uma manivela, que é responsável por processar os dados e fazer a entrega dos resultados processados. Na primeira caixa, da esquerda para a direita, intitulada como `Driving table` , acontece a recuperação de todas as linhas da primeira tabela a ser lida (tabela externa), que satisfaz as condições de pesquisa que diz respeito a somente ela,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

descartando as linhas que não entram nesta condição (representado pela lixeira). Esse primeiro descarte de linhas tem a ver com critérios de seleção onde uma ou mais colunas possam ser comparadas com valores fixos ou variáveis `bind`, através de predicados de filtros, por exemplo. Após isso, as linhas que passaram nesta filtragem são encaminhadas a segunda caixa (primeiro `join`), que tem como objetivo selecionar apenas as linhas que correspondem às linhas de uma segunda tabela do relacionamento. Havendo outras tabelas, neste caso, ilustradas pela terceira caixa, este trabalho será realizado novamente até que se tenha todos os resultados dentro das condições de pesquisa impostas, retornando o resultado. Note que, a cada processamento, as linhas que não correspondem aos critérios vão sendo descartadas.

Um exemplo de método `NESTED LOOPS` é demonstrado a seguir. Temos uma instrução SQL cujo intuito é selecionar o nome, número de departamento e a localização de cada empregado que faz parte da tabela `employees`.

Instrução SQL:

```
select e.first_name, d.department_id
from   employees e, departments d
where  e.department_id = d.department_id
```

Resultado da instrução SQL:

```
FIRST_NAME           DEPARTMENT_ID
-------------------- -------------
Steven                          90
Neena                           90
Lex                             90
Alexander                       60
Bruce                           60
David                           60
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
-------------------- ---------------
William                         110

106 rows selected.
```

Plano de execução:

```
Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE
1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES'
2    1     NESTED LOOPS
3    2       TABLE ACCESS (FULL) OF 'DEPARTMENTS'
4    2       INDEX (RANGE SCAN) OF 'EMP_DEPARTMENT_IX' (NON-UNIQUE)
```

Figura 5.2: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img01code.txt

Utilizando o método `NESTED LOOPS` , o otimizador realizou os seguintes passos:

- O otimizador percorreu a tabela `employees` que, nesse exemplo, trata-se da tabela externa ou guia, como também é chamada, para obter todas as linhas;
- Para cada linha retornada da tabela `employees` , o otimizador encontra a linha correspondente na tabela `departments` usando o `department_id` como relacionamento;
- O otimizador combina cada linha da tabela externa, no caso, `employees` , com a linha da tabela `departments` , retornando o resultado.

O método `NESTED LOOPS` poderá ser utilizado pelo otimizador em algumas situações:

- Onde tenha uma tabela externa e uma interna;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Quando não precisar retornar todas as linhas;
- Quando existir um `hint first_rows` na instrução SQL ou o parâmetro `optimizer_mode` estiver configurado com esta opção;
- Em casos em que há um índice na coluna do `join` em uma tabela interna;
- Pode aparecer em execuções paralelas se a tabela externa for acessada por completo (`FULL SCAN`);
- Quando os atributos do relacionamento das tabelas externa e interna são correspondentes;
- Em tabelas que possuem índices do tipo `B-tree` e com poucos dados;

O otimizador pode não usar `NESTED LOOPS` nestes casos:

- Quando o volume de registros em ambas as tabelas for baixo;
- Quando o caminho de acesso da tabela interna é independente dos dados provenientes da tabela externa.

Você verá o uso mais frequente do `join NESTED LOOPS` quando o modo do otimizador estiver configurado para trabalhar em `first_rows` , pois ele tem como objetivo mostrar resultados instantâneos ao usuário na medida em eles são recuperados. Nesse caso, não há necessidade de armazenamento dos dados em cache antes de serem retornados ao usuário. No caso da junção `HASH` , por exemplo, haveria essa necessidade.

Em caso de tabelas com muitos dados e em que não há a existência de índices ou índices ineficientes, o tempo de resposta pode se tornar baixo se este método for utilizado. Para esse exemplo, poderíamos dizer que a escolha do método pelo

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

otimizador seria pelo fato de estar atendendo às situações 4, 6 e 7 da listagem apresentada anteriormente.

5.5 SORT MERGE JOIN

Nesse método, o otimizador combina as linhas ordenadas de cada tabela e se baseia em igualdade de condições. O otimizador ordena cada conjunto de linhas a serem combinadas pelos valores que forem usados na condição do `join` das tabelas, se eles não estiverem ordenados. Depois de montados os dois conjuntos, um para cada tabela, o otimizador combina cada linha de um conjunto de valores, com cada linha do outro, de acordo com as colunas usadas na condição, retornando o resultado em linhas combinadas.

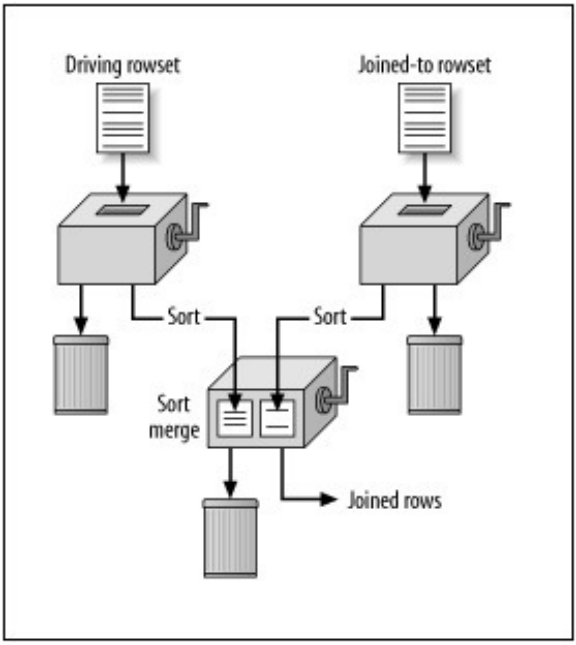

Figura 5.3: Fonte: SQL Tuning. Por Dan Tow. (https://books.google.com.br/books?hl=pt-BR&id=dOYPtfVqS_MC&q=Joins#v=onepage&q=Merge-Join&f=false)

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Na figura que mostra o método de junção `SORT MERGE`, as duas caixas, `Driving rowset` e `Joined-to rowset`, representam o filtro dos dois conjuntos de valores das tabelas. Nesse momento, esses conjuntos são filtrados, cada qual por condições que dizem respeito a somente eles, e os valores são ordenados, descartando as linhas que não condizem com as condições de cada conjunto. Após isso, são encaminhados para a caixa inferior, onde são combinados um a um através do `join`, gerando o resultado final.

Para exemplificar este método, pode-se ter como exemplo a seguinte instrução que seleciona informações sobre o empregado referentes ao nome, departamento e cargo.

Instrução SQL:

```
select /*+ LEADING(job_history) */
       e.employee_id
      ,e.first_name
      ,d.department_id
      ,d.department_name
      ,j.job_id
from   employees   e
      ,departments d
      ,job_history j
where  e.department_id = d.department_id
and    e.employee_id   = j.employee_id
and    d.department_id = 10
and    e.employee_id   = 200
```

Resultado da instrução SQL:

```
EMPLOYEE_ID FIRST_NAME           DEPARTMENT_ID DEPARTMENT_NAME
             JOB_ID
----------- -------------------- ------------- ------------------
------------ ----------
        200 Jennifer                        10 Administration
             AD_ASST
        200 Jennifer                        10 Administration
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

AC_ACCOUNT

Plano de execução:

```
Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=71)
1    0   MERGE JOIN (CARTESIAN) (Cost=2 Card=1 Bytes=71)
2    1     MERGE JOIN (CARTESIAN) (Cost=2 Card=1 Bytes=58)
3    2       TABLE ACCESS (BY INDEX ROWID) OF 'JOB_HISTORY' (Cost=1 Card=1 Bytes=20)
4    3         INDEX (RANGE SCAN) OF 'JHIST_EMPLOYEE_IX' (NON-UNIQUE) (Cost=1 Card=1)
5    2       BUFFER (SORT) (Cost=1 Card=1 Bytes=38)
6    5         TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES' (Cost=1 Card=1 Bytes=38)
7    6           INDEX (UNIQUE SCAN) OF 'EMP_EMP_ID_PK' (UNIQUE)
8    1     BUFFER (SORT) (Cost=1 Card=1 Bytes=13)
9    8       INDEX (UNIQUE SCAN) OF 'DEPT_ID_PK' (UNIQUE)
```

Figura 5.4: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap05img02code.txt

Utilizando o método `SORT MERGE`, o otimizador realiza os seguintes passos:

- Acessa o índice `jhist_employee_ix` referente à tabela `job_history`.
- Recupera e ordena as linhas da tabela `employees` pela coluna `employee_id`;
- Depois disso, o otimizador combina (Merge) o conjunto de linhas das tabelas `job_history` e `employees`.
- Recupera e ordena as linhas da tabela `departments` pela coluna `department_id`, através do índice `dept_id_pk`;
- Depois disso, o otimizador combina (`MERGE`) o conjunto de linhas referente ao primeiro `MERGE`, entre `employees` e `job_history`, com os dados de `departments`, e retorna o resultado.

Considerações sobre esse método:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- O otimizador lê as linhas de cada uma das tabelas do relacionamento;
- Em um relacionamento de duas tabelas, elas são ordenadas na mesma sucessão, baseando-se em um predicado (coluna do relacionamento);
- As duas tabelas serão unidas formando assim uma tabela composta;
- Este método pode ser utilizado pelo otimizador quando o `hint ALL_ROWS` é especificado;
- Pode ser utilizado quando não houver índices nas colunas do relacionamento;
- A escolha ou não por este método pode ser influenciada pela configuração dos parâmetros `SORT_AREA_SIZE` e `DB_FILE_MULTI_BLOCK_READ_COUNT`.

Quando o otimizador usa a junção `SORT MERGE`?

- Quando a condição de junção é uma condição de desigualdade, por exemplo, o sinal de menor (<), menor igual (<=), maior (>) e de igualdade (=). Isso ocorre porque o método `HASH` não pode ser usado para condições de desigualdade e, se o volume de dados for grande, o uso da junção `NESTED LOOPS` não é uma boa opção.
- Se algum outro atributo da tabela, que não seja o atributo da junção, necessitar ser ordenado (cláusula `order by`). Nesses casos, ele preferirá `SORT MERGE` ao método `HASH`, pois o custo será menor.

O método `SORT MERGE` pode ser visto com o uso dos `hints` `ALL_ROWS` e `FIRST_ROWS`, porque ele funciona de forma a primeiro ordenar as origens dos dados e depois retornar os

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

resultados. Portanto, se o conjunto de dados for volumoso e o otimizador estiver configurado com o parâmetro `OPTIMIZER_MODE` igual a `FIRST_ROWS`, o otimizador poderá preferir o método de junção `SORT MERGE` à junção `NESTED LOOPS` devido a essa grande quantidade de dados. No caso de o otimizador estar configurado com o parâmetro `OPTIMIZER_MODE` igual a `ALL_ROWS` e se qualquer condição de desigualdade for usada na instrução SQL, o otimizador pode preferir usar o método `SORT MERGE` ao `HASH`.

O método `SORT MERGE`, provavelmente, será melhor em performance do que o método `NESTED LOOPS` quando se fala em processamento de um grande volume de dados. Contudo, um melhor cenário ainda poderia ser o uso de uma junção `HASH` dependendo do caso. Por outro lado, o método `SORT MERGE` executa melhor do que um método `HASH` quando as colunas da condição de relacionamento entre as tabelas já estão ordenadas, ou ordenações não são necessárias.

Um ponto importante a ser ressaltado é que, ao contrário do `NESTED LOOPS`, com o qual a leitura da tabela mais interna é condicionada a quantidade de vezes que é lida a tabela externa, no método de junção `SORT MERGE` cada uma das tabelas envolvidas é acessada no máximo uma vez. Assim, este método prova ser melhor do que o método `NESTED LOOPS` quando o conjunto de dados é maior.

5.6 HASH JOIN

Usando esse método, o otimizador acessa uma tabela, geralmente aquela com o menor dos resultados, e monta na

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

memória uma tabela `hash` com base na chave de junção. Depois disso, ele varre a outra tabela da junção, aquela com o maior dos resultados, e verifica na tabela `hash` as combinações de linhas, retornando os resultados. Nesse método, alguns parâmetros são consultados para que este método seja utilizado. Especificando o `hint` `ORDERED`, juntamente na instrução SQL, pode-se determinar qual tabela será usada para a criação da tabela `hash`.

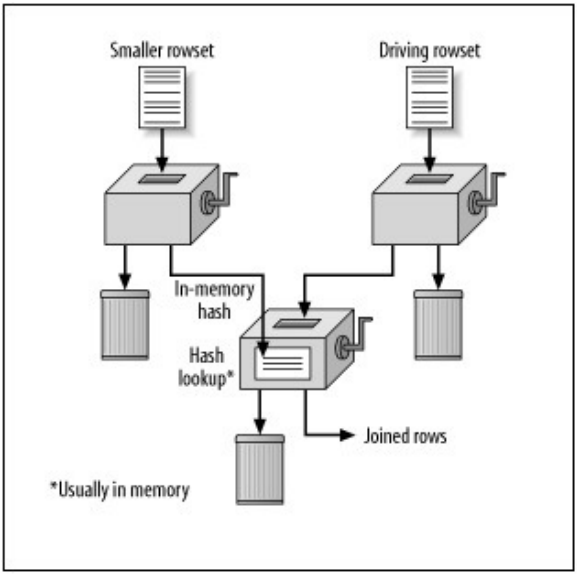

Figura 5.5: Fonte: SQL Tuning. Por Dan Tow. (https://books.google.com.br/books?hl=pt-BR&id=dOYPtfVqS_MC&q=Joins#v=onepage&q=Merge-Join&f=false)

Conforme pode ser visto na imagem, no método de junção `HASH`, as duas caixas na parte superior representam os filtros referentes às duas tabelas. A saída de uma das caixas representa a escolha do otimizador por uma das tabelas (`Smaller rowset`), para que seja montada a tabela `hash`. A outra saída é referente à tabela que será varrida e combinada com os valores da tabela

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

hash .

Este exemplo demonstra o uso do método de junção HASH combinado com o hint USE_HASH . A instrução SQL selecionará o nome do empregado e o código do departamento de todos os empregados da tabela employees .

Instrução SQL:

```
select /*+ USE_HASH(d) */
       e.employee_id
      ,e.first_name
      ,d.department_id
      ,d.department_name
from   employees   e
      ,departments d
where  e.department_id = d.department_id
and    e.employee_id   = 160
```

Resultado da instrução SQL:

```
EMPLOYEE_ID FIRST_NAME           DEPARTMENT_ID DEPARTMENT_NAME
----------- -------------------- ------------- ------------------
------------
        160 Louise                           80 Sales
```

Plano de execução:

```
Execution Plan
----------------------------------------------------------
  0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=1 Bytes=51)
  1    0   HASH JOIN (Cost=3 Card=1 Bytes=51)
  2    1     TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES' (Cost=1 Card=1 Bytes=38)
  3    2       INDEX (RANGE SCAN) OF 'EMP_EMP_ID_PK' (UNIQUE) (Cost=1 Card=1)
  4    1     TABLE ACCESS (FULL) OF 'DEPARTMENTS' (Cost=1 Card=41 Bytes=533)
```

Figura 5.6: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img03code.txt

Os seguintes passos serão executados pelo otimizador:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Monta uma tabela `hash` na memória com a tabela `employees`;
- Apanha cada linha da tabela `department` e as compara com a tabela `hash`.

Considerações sobre este método:

- Pode ser muito poderoso quando aplicado corretamente;
- Pode ser útil para juntar tabelas com poucos dados e poucas colunas, com tabelas com muitos dados e muitas colunas;
- Pode sofrer influência dependendo do que está definido nos parâmetros `HASH_AREA_SIZE`, `HASH_MULTIBLOCK_IO_COUNT` e `DB_BLOCK_SIZE`;
- Será usado somente quando o predicado da junção for uma igualdade.

Quando o otimizador usa junção `HASH`?

- O otimizador usa junção `HASH` quando une grandes tabelas ou uma grande fração de pequenas tabelas.
- Ao contrário do `NESTED LOOPS`, a saída do resultado de uma junção `HASH` não é instantânea, pois os relacionamentos de `HASH` são bloqueados no momento da construção da tabela `hash`.

Você pode ver mais junções `HASH` nos planos de execução quando o modo do otimizador estiver como `ALL_ROWS`, porque ele funciona de forma a exibir os resultados depois que todas as linhas, de pelo menos uma das tabelas, estiverem com seus respectivos `hashs` gerados na tabela `hash`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

5.7 CARTESIAN JOIN

Sempre que temos uma instrução SQL com mais de uma tabela, ou seja, sempre que informamos mais de uma tabela na cláusula `from` de uma instrução, é provável que informaremos os relacionamentos referentes a estas tabelas. Caso esses relacionamentos não sejam mencionados na instrução, o motor do Oracle gerará um resultado o qual chamamos de "Produto Cartesiano", que é derivado de uma junção cartesiana (`CARTESIAN join`).

Imaginemos duas tabelas em uma instrução SQL. A tabela `'A'` com 10 registros e a tabela `'B'` com 20 registros. Caso não seja informado pelo menos um relacionamento na cláusula `where` , ligando essas duas tabelas, o resultado será o retorno de 200 linhas, ou seja, para cada linha da tabela `'A'` , ele retornará todas as linhas da tabela `'B'` , ou vice-versa. O número total de linhas resultantes dessa junção é calculado usando a seguinte fórmula:

```
Tabela A x Tabela B = Linhas retornadas
```

Quando submetemos uma instrução com essas características, o otimizador utiliza o método `CARTESIAN join` para fazer a junção dessas tabelas. Este método faz unir todas as linhas de uma tabela com cada linha da outra, criando o produto cartesiano desses dois conjuntos. Veja o exemplo a seguir:

```
SQL> select ename, dname
  2  from   emp
  3        ,dept
  4  /
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2372241248

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |           |    56 |   896 |     5   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN  |           |    56 |   896 |     5   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL    | DEPT      |     4 |    40 |     2   (0)| 00:00:01 |
|   3 |   BUFFER SORT          |           |    14 |    84 |     3   (0)| 00:00:01 |
|   4 |    INDEX FAST FULL SCAN| EMP_IDX_2 |    14 |    84 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 5.7: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap05img04code.txt

Nesse exemplo não há relacionamentos informados para as tabelas `emp` e `dept`. Com isso, o otimizador acabou escolhendo o método `CARTESIAN Join`. É comum ver este tipo de método sendo relacionado juntamente com `MERGE` ou `HASH`.

Os seguintes passos serão executados pelo otimizador:

- O otimizador une todas as linhas de uma tabela com todas as linhas da outra tabela, criando o produto cartesiano dos dois conjuntos.
- O número total de linhas resultante do relacionamento será a multiplicação entre as quantidades retornadas destas linhas, uma pela outra.

Considerações sobre esse método:

- Nesse método, nenhuma condição de junção existe. Em alguns casos, o otimizador pode selecionar uma condição de filtro comum entre as duas tabelas como uma possível condição de associação, se ele julgar prudente.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Se um método `CARTESIAN join` aparecer em um plano de execução, isso pode ter ocorrido por uma condição de junção omitida inadvertidamente por esquecimento, por exemplo. Em geral, se uma instrução SQL possui mais de uma tabela, são necessárias condições de junção entre elas para evitar uma junção cartesiana.

Quando o otimizador usa `CARTESIAN join` ?

- Quando não existir nenhuma condição de junção. Em alguns casos, o otimizador pode utilizar uma condição de filtro comum entre as duas tabelas como uma possível condição de junção.
- O `CARTESIAN join` é um método eficiente. O otimizador pode, por exemplo, decidir gerar um produto cartesiano de duas tabelas muito pequenas primeiramente, em vez de uni-las separadamente a uma tabela com grande volume de dados.
- Ele poderá ser usado pelo otimizador quando um `hint ORDERED` for usado na instrução SQL, especificando uma ordem na qual uma determinada tabela venha antes que sua tabela de junção seja especificada.

Nota: No tópico *Driving Tables e os Planos Cartesianos (`CARTESIAN join`)* será vista com maiores detalhes a questão do `CARTESIAN join` como método de junção escolhido pelo otimizador, mesmo que não haja tabelas sem relacionamento na instrução SQL.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Portanto, quando você verificar a existência de um `CARTESIAN join` no plano de execução, isso não quer dizer que algum relacionamento está faltando na instrução SQL. Pode ser que o otimizador decidiu optar por utilizá-lo em substituição a um determinado método ou o tenha adicionado para auxiliá-lo em alguma busca. Para ter certeza disso, você poderá avaliar sua instrução SQL e verificar se realmente não existem tabelas que não estejam de alguma forma sendo relacionadas a qualquer outra na instrução. Em uma instrução SQL, a menos que estejamos querendo produzir um `CARTESIAN join`, todas as tabelas envolvidas estarão ligadas direta ou indiretamente, umas às outras.

No plano de execução, caso o `CARTESIAN join` esteja sendo mostrado, deve ser possível visualizar quais objetos estão fazendo parte deste `join`, dando para identificá-los na instrução. A inexistência de `CARTESIAN join` no plano também não quer dizer que todos os relacionamentos foram informados. Nesses casos, não muito comuns, mas que podem acontecer, o otimizador pode dar um jeito para resolver essas ausências.

Veja o exemplo a seguir em que, mesmo não existindo um relacionamento entre as tabelas `emp` e `dept`, o otimizador conseguiu achar uma forma de relacioná-las sem utilizar o método `CARTESIAN join`. Contudo, embora no plano de execução pareça estar tudo bem, os dados retornados podem não estar dentro do objetivo esperado, ou sua performance pode não ser a melhor. Nesses casos, somente analisando a estrutura da instrução SQL, revendo cada tabela e relacionamento escrito é possível detectar onde faltam possíveis relacionamentos que possam causar o `CARTESIAN join`, e inseri-los para deixar a instrução da forma correta.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Neste exemplo, estamos solicitando que sejam listadas as colunas `ename` (Nome do Empregado), `dname` (Nome do Departamento) e `deptno` (IDs do Departamento, tanto da tabela de empregados, quanto da tabela de departamentos) das tabelas `emp` (Empregados) e `dept` (Departamentos). Não estamos relacionando as tabelas. Apenas estamos filtrando uma delas pela coluna `deptno`. Nesta primeira execução, vamos filtrar pela coluna `deptno` da tabela `emp`. Veja a seguir o plano de execução:

```
SQL> select ename, dname
  2  from   emp e, dept d
  3  where  e.deptno = 20;

Execution Plan
----------------------------------------------------------
Plan hash value: 2723707427

-----------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |    20 |   380 |     5   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN|      |    20 |   380 |     5   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL  | EMP  |     5 |    45 |     2   (0)| 00:00:01 |
|   3 |   BUFFER SORT        |      |     4 |    40 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | DEPT |     4 |    40 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("E"."DEPTNO"=20)
```

Figura 5.8: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap05img05code.txt

Note que, nesse caso, como esperado, o plano de execução mostra o método `CARTESIAN Join`. Note também que o otimizador optou por realizar varreduras completas nas tabelas `emp` e `dept`. Agora vamos ver os dados retornados, mas antes vamos realizar uma contagem dos registros dessas tabelas:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select count(*)
  2  from   emp e
  3  where  e.deptno = 20;

  COUNT(*)
----------
         5

SQL> select count(*)
  2  from   dept d;

  COUNT(*)
----------
         4

SQL>
```

A tabela `emp` com o filtro buscando apenas empregados com departamento igual a 20, retornou a contagem de 5 registros. Já a tabela `dept`, executada sem filtro, retornou 4. Se executarmos uma instrução SQL utilizando essas tabelas sem relacioná-las, utilizando a regra comentada anteriormente, o resultado deverá retornar 20 registros (5 x 4 = 20). Agora vemos o resultado:

```
SQL> select ename, dname, e.deptno, d.deptno
  2  from   emp e, dept d
  3* where  e.deptno = 20

ENAME      DNAME              DEPTNO     DEPTNO
---------- -------------- ---------- ----------
SMITH      ACCOUNTING             20         10
SMITH      RESEARCH               20         20
SMITH      SALES                  20         30
SMITH      OPERATIONS             20         40
JONES      ACCOUNTING             20         10
JONES      RESEARCH               20         20
JONES      SALES                  20         30
JONES      OPERATIONS             20         40
SCOTT      ACCOUNTING             20         10
SCOTT      RESEARCH               20         20
SCOTT      SALES                  20         30
SCOTT      OPERATIONS             20         40
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
ADAMS      ACCOUNTING              20         10
ADAMS      RESEARCH                20         20
ADAMS      SALES                   20         30
ADAMS      OPERATIONS              20         40
FORD       ACCOUNTING              20         10
FORD       RESEARCH                20         20
FORD       SALES                   20         30
FORD       OPERATIONS              20         40

20 rows selected.
```

Veja que o resultado retornou 20 registros. Nesse caso, o otimizador gerou um produto cartesiano como era esperado. Note também que, para cada ID de Departamento da tabela `dept`, se repetiu o ID do Departamento da tabela `emp`. Agora vamos inverter o filtro colocando-o na tabela `dept`. Primeiro faremos a recontagem dos dados, em seguida, veremos o resultado e depois o plano de execução:

```
SQL> select count(*)
  2  from   emp e;

  COUNT(*)
----------
        14

SQL> select count(*)
  2  from   dept d
  3  where  d.deptno = 20;

  COUNT(*)
----------
         1

SQL>
```

Como o filtro em cima da tabela `dept` retorna apenas um registro, o produto cartesiano desta instrução SQL retornará a mesma quantidade de registros da tabela `emp` (14 x 1 = 14). Vamos dar uma olhada no resultado.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select ename, dname, e.deptno, d.deptno
  2  from   emp e, dept d
  3* where  d.deptno = 20

ENAME      DNAME              DEPTNO     DEPTNO
---------- -------------- ---------- ----------
SMITH      RESEARCH               20         20
ALLEN      RESEARCH               30         20
WARD       RESEARCH               30         20
JONES      RESEARCH               20         20
MARTIN     RESEARCH               30         20
BLAKE      RESEARCH               30         20
CLARK      RESEARCH               10         20
SCOTT      RESEARCH               20         20
KING       RESEARCH               10         20
TURNER     RESEARCH               30         20
ADAMS      RESEARCH               20         20
JAMES      RESEARCH               30         20
FORD       RESEARCH               20         20
MILLER     RESEARCH               10         20

14 rows selected.
```

Veja que o resultado agora retornou 14 registros. O otimizador gerou um produto cartesiano como era também esperado. Note também que, para cada ID de Departamento da tabela `emp`, se repetiu o ID do Departamento da tabela `dept`. Agora vamos ver o plano de execução:

```
SQL> select ename, dname
  2  from   emp e, dept d
  3  where  d.deptno = 20;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1818732800

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |           |    14 |   266 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |           |    14 |   266 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPT      |     1 |    13 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN          | PK_DEPTNO |     1 |       |     0   (0)| 00:00:01 |
|   4 |   INDEX FULL SCAN             | EMP_IDX_2 |    14 |    84 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPTNO"=20)
```

Figura 5.9: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img06code.txt

No plano de execução não aparece nenhuma referência ao produto cartesiano, embora o resultado tenha ocorrido como tal. O que aconteceu é que o otimizador constatou que a coluna referente ao predicado existe nas duas tabelas, e que uma delas, aquela utilizada na instrução SQL, possui uma chave primária. Com isso, ele acabou utilizando as duas colunas como forma de juntá-las e utilizou para isso um método de acesso via índice, não utilizando o método de junção `CARTESIAN join`. Nesse caso, analisando apenas o plano de execução, sem analisar os dados, teríamos um produto cartesiano maquiado, escondido digamos assim, onde não vemos qualquer referência de que ele esteja acontecendo. Por isso, é sempre válido ficar atento não só com relação aos planos de execução, mas sim aos dados. Se tiver dúvidas, primeiro de tudo, valide o plano de execução, mas depois também analise os dados retornados e a própria instrução em si.

No próximo tópico, vamos ver uma comparação entre os três

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

métodos: NESTED LOOPS , SORT MERGE e HASH . Veremos um resumo de suas características e quando melhor são usados.

5.8 COMPARAÇÃO ENTRE OS MÉTODOS NESTED LOOPS, SORT MERGE E HASH

Cada método de junção possui características próprias para juntar duas ou mais tabelas. O otimizador, dependendo da situação, vai optar por um ou outro método levando em consideração as informações que ele conhece sobre os objetos que fazem parte da instrução SQL e também a forma como você escreveu tal instrução. Você precisará ter conhecimento sobre os métodos de junção para entender e conseguir ler os planos de execução gerados pelo otimizador.

A seguir é mostrado um comparativo entre os três tipos de métodos de junção abordados neste capítulo, listando suas características, desvantagens e vantagens, e outras informações pertinentes a suas funcionalidades:

Comparativo entre os principais métodos de junção.

Categoria	Junção NESTED LOOPS	Junção SORT MERGE	Junção HASH
Sugestão (hint) do otimizador	USE_NL	USE_MERGE	USE_HASH
Quando pode ser usado	Qualquer junção	Somente equijunções	Somente equijunções
Recursos relacionados	CPU, E/S de disco	Memória, segmentos temporários	Memória, segmentos temporários
	Eficaz com		

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Características	índices altamente seletivos e buscas restritivas. Usada para retornar rapidamente a primeira linha de um resultado	Melhor do que `NESTED LOOPS` quando um índice estiver faltando ou o critério de busca não for muito seletivo. Pode trabalhar com memória limitada.	Melhor do que `NESTED LOOPS` quando um índice estiver faltando ou o critério de busca não for muito seletivo. Normalmente é mais rápida do que um `SORT MERGE` .
Desvantagens	Não eficaz quando os índices estão faltando ou se os critérios do índice não forem limitadores.	Exige uma classificação (ordenação) nas duas tabelas. Ela é montada para a melhor vazão e não retorna a primeira linha até que todas as linhas sejam encontradas.	Pode exigir uma grande quantidade de memória para a tabela `hash` ser montada. Não retorna as primeiras linhas rapidamente. Poder ser extremamente lenta se tiver que realizar a operação no disco.

Entendendo as características de cada método é possível analisá-los com intuito de saber como o otimizador fez para chegar ao resultado final. Em resumo, o meio como as tabelas são acessadas e manipuladas influencia na performance dos seus programas e pode determinar o sucesso ou não da sua execução.

Além das informações referentes aos métodos de junção, nos planos de execução são apresentadas informações sobre os métodos de acesso utilizados para acessar tabelas e índices, como `TABLE ACCESS (BY INDEX ROWID)` ou `INDEX (RANGE SCAN)` . Essas informações indicam o meio de acesso que o otimizador utiliza para acessar os dados ao aplicar o método de junção escolhido. No próximo capítulo, abordarei tais métodos em detalhes.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

5.9 TIPOS DE JUNÇÃO

Os métodos de junção também podem ser classificados em tipos. Dependendo das condições de junção existentes na instrução SQL, elas podem fazer referência a um determinado tipo de junção. Alguns tipos de junção como `semijoins`, `antijoins` e `outer` podem aparecer referenciados no plano de execução, juntamente do nome do método. Já os tipos como `inner joins` se mostram implícitos e podem não trazer referência visual no plano.

Alguns tipos de junção são mais comuns quando estamos trabalhando com subqueries (subconsultas), como é o caso dos tipos `semijoins` e `antijoins`. Então você poderá vê-los apenas nesses casos. É interessante conhecer esses conceitos, pois, principalmente na documentação da Oracle eles são mencionados com frequência dentro de outros contextos. Os tipos que veremos a seguir são: `inner join`, `outer join`, `semijoin`, `antijoin` e `star query`.

inner joins

Junções do tipo `inner join` (também chamadas de `sample joins`) retornam apenas linhas que satisfazem a condição de junção existente na instrução SQL. As `inner joins` são classificadas ainda em `equijoins` ou `nonequijoins`.

equijoins : uma `equijoin` ocorre quando na instrução SQL a condição de junção entre duas tabelas contém um operador de igualdade. Veja o exemplo:

```
SQL> select e.employee_id, e.last_name, d.department_name
  2  from   employees e, departments d
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  3  where  e.department_id=d.department_id;

Execution Plan
----------------------------------------------------------
Plan hash value: 2052257371

-----------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |   106 |  3286 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN         |             |   106 |  3286 |     5  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPARTMENTS |    27 |   432 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMPLOYEES   |   107 |  1605 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Figura 5.10: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap05img07code.txt

Note a condição de junção `e.department_id = d.department_id`. Se uma linha na tabela de Empregados contiver um ID de departamento que corresponda ao valor de uma linha na tabela Departamentos, o Oracle retornará o resultado da junção. Caso contrário, ele não retornará resultado algum.

nonequijoins **:** uma `nonequijoin` ocorre quando, na instrução SQL, a condição de junção contém um operador que não é um operador de igualdade. A instrução SQL a seguir lista todos os empregados cuja data de contratação ocorreu quando o empregado 176 (que está listado na tabela `job_history`, porque ele mudou de emprego em 2007) estava trabalhando na empresa:

```
SQL> select e.employee_id, e.first_name, e.last_name, e.hire_date
  2  from   employees e, job_history h
  3  where  h.employee_id = 176
  4  and    e.hire_date between h.start_date and h.end_date;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2373703682

-----------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |    84 |  3948 |     5   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |             |    84 |  3948 |     5   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| JOB_HISTORY |     2 |    40 |     2   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMPLOYEES   |    42 |  1134 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("H"."EMPLOYEE_ID"=176)
   3 - filter("E"."HIRE_DATE">="H"."START_DATE" AND
              "E"."HIRE_DATE"<="H"."END_DATE")
```

Figura 5.11: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap05img08code.txt

Nesse exemplo, a condição de junção entre as tabelas `employees` e `job_history` não contém um operador de igualdade, caracterizando uma `nonequijoin`. Este tipo de junção não costuma ser comum.

outer joins

Uma junção do tipo `outer join` retorna todas as linhas que satisfazem a condição de junção e também retorna algumas ou todas as linhas de uma tabela para a qual nenhuma linha da outra satisfaz a condição de junção. Assim, uma `outer join` estende o resultado de uma *simple join* (ou `inner join`).

No padrão ANSI, a cláusula `outer joins` especifica uma junção externa. Com relação à sintaxe, na cláusula `from`, uma tabela é colocada à esquerda da palavra-chave `outer join` e uma outra tabela é colocada à direita desta mesma palavra-chave. As

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`outer joins` são subdivididas ainda em: `left`, `right` ou `full`, que indicará de que lado será realizada a junção externa (`outer join`). Dessa forma, quando você escrever uma `outer join`, você precisa informar de que forma ela será realizada, indicando uma dessas opções.

A junção `right outer join` é uma das operações `join` que permitem especificar uma cláusula `join`. Ela preserva as linhas não correspondentes da segunda tabela (a da direita na cláusula `outer join`), unindo-as à primeira tabela (a da esquerda na cláusula `outer join`), com valores nulos. A junção `a left outer join b`, por exemplo, é equivalente à junção `b right outer join a`, apenas com as colunas em uma ordem diferente.

Por exemplo, na instrução a seguir, a tabela de Empregados (`employees`) é a tabela da esquerda na cláusula `outer join` e também a tabela de junção externa, pois estamos declarando um `left outer join`.

```
SQL> select employee_id, last_name, first_name, department_name
  2  from   employees left outer join departments
  3  on     (employees.department_id=departments.department_id);
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2296652067

-----------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |   107 |  4066 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN OUTER   |             |   107 |  4066 |     5  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| EMPLOYEES   |   107 |  2354 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| DEPARTMENTS |    27 |   432 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("EMPLOYEES"."DEPARTMENT_ID"="DEPARTMENTS"."DEPARTMENT_ID"(+))
```

Figura 5.12: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img09code.txt

`outer joins` requerem que a tabela externa na junção seja a tabela de condução (`Driving Table`). Neste exemplo, a tabela `employees` é a tabela de condução, e a tabela `departments` é a tabela de junção (ou Tabela Interna).

Se trocarmos o `left` por `right` mantendo a ordem das tabelas, veremos que a tabela de condução acaba mudando no plano de execução, pois estamos direcionando a junção externa para a outra direção, a da tabela à direita. Veja a seguir o resultado.

```
SQL> select employee_id, last_name, first_name, department_name
  2  from   employees right outer join departments
  3  on     (employees.department_id=departments.department_id);
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2392326604

-----------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |   106 |  4028 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN OUTER   |             |   106 |  4028 |     5  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPARTMENTS |    27 |   432 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMPLOYEES   |   107 |  2354 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("EMPLOYEES"."DEPARTMENT_ID"(+)="DEPARTMENTS"."DEPARTMENT_ID")
```

Figura 5.13: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap05img10code.txt

A tabela de condução (Driving Table) é a tabela `departments`, pois estamos direcionando a junção externa para a tabela da direita na `outer join` através de `right outer join` na cláusula `from`. Nela, temos a tabela `employees` aparecendo à esquerda da cláusula `outer join` e `departments` aparecendo à direita dessa cláusula.

A seguir, veja as características para reforçar as regras da sintaxe `ANSI` para o uso de `outer joins` na cláusula `from`:

- Para uma instrução SQL que executa uma junção externa (`outer join`) das tabelas A e B e retorna todas as linhas de A (uma junção externa esquerda), use a sintaxe `left [outer] join` na cláusula `from` ou aplique o operador de junção externa (+) para todas as colunas de B na condição de junção na cláusula `where`. Para todas as linhas em A que não têm linhas correspondentes em B, o otimizador retornará nulo para qualquer coluna de B.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Para uma instrução SQL que executa uma junção externa (`outer join`) das tabelas A e B e retorna todas as linhas de B (uma junção externa direita), use a sintaxe `right [outer] join` na cláusula `from` ou aplique o operador de junção externa (+) para todas as colunas de A na condição de junção na cláusula `where` . Para todas as linhas em B que não têm linhas correspondentes em A, Oracle retornará nulo para qualquer coluna de A.
- Para uma instrução SQL que executa uma junção externa e retorna todas as linhas de A e B, estendida com nulos, se eles não satisfazem a condição de junção (uma junção externa completa), use a sintaxe `full [outer] join` na cláusula `from` .

A Oracle recomenda que você use a sintaxe `outer join` padrão `ANSI` , aquela usada na cláusula `from` , em vez do operador nativo (+) nas junções da cláusula `where` . As instruções SQL de junção externa que usam o operador nativo do Oracle (+) estão sujeitas a regras e restrições, as quais não se aplicam à sintaxe `outer join` da cláusula `from` .

Quando estamos falando de `outer joins` , o otimizador escolhe métodos de acessos levando em consideração este tipo. No plano de execução é possível observar o tipo referente ao método de acesso. Vamos agora dar uma olhada nos métodos de junção com relação às características voltadas para estes tipos de junção. Entretanto, não vamos nos aprofundar muito em cada um deles, pois já discutimos sobre estes métodos de junção anteriormente. O que vamos ver são apenas algumas características destes tipos junção com relação aos métodos de junção.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

NESTED LOOPS OUTER joins

O banco de dados usa o tipo de operação NESTED LOOPS OUTER join para executar um loop entre uma outer join relacionado a duas tabelas. A outer join retorna as linhas da tabela externa mesmo quando nenhuma linha correspondente está na tabela interna, como já mencionamos.

Em um NESTED LOOPS padrão, o otimizador escolhe a ordem das tabelas, definindo qual será a tabela de condução e qual será a tabela de junção com base no custo. No entanto, em um outer join de uma junção NESTED LOOPS , a condição de junção determinará a ordem das tabelas, pois o Oracle usa a tabela externa para conduzir as linhas da tabela interna. O otimizador usa junções NESTED LOOPS para processar outer joins nas seguintes circunstâncias:

- Quando for possível designar uma tabela externa para controlar uma tabela interna.
- Quando o volume de dados é baixo o suficiente para tornar o método NESTED LOOPS eficiente.

```
SQL> select /*+ USE_NL(e d) */ ename, dname, d.deptno, e.deptno
  2  from   emp e
  3        ,dept d
  4  where  d.deptno = e.deptno(+);
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 582125897

---------------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |    14 |   308 |     5   (0)| 00:00:01 |
|   1 |  NESTED LOOPS OUTER          |           |    14 |   308 |     5   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL          | DEPT      |     4 |    52 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS BY INDEX ROWID| EMP       |     4 |    36 |     1   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | EMP_IDX_1 |     5 |       |     0   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("D"."DEPTNO"="E"."DEPTNO"(+))
```

Figura 5.14: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img11code.txt

> **Nota:** Para esse exemplo foi utilizado o `hint` `USE_NL` para forçar o otimizador a utilizar o método de junção `NESTED LOOPS`.

`HASH JOIN OUTER` joins

O otimizador usa junções `HASH` para processar uma `outer join` quando uma das seguintes condições for atendida:

- Quando o volume de dados é grande o suficiente para tornar o método `HASH` eficiente.
- Quando não é possível designar uma tabela externa para controlar uma tabela interna.

Neste tipo de método, o custo determina a ordem das tabelas. A tabela externa, incluindo as linhas não correspondentes na tabela

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

interna (`preserved rows`), pode ser usada tanto como tabela `hash` , quanto como tabela conduzida pela tabela `hash` .

No exemplo a seguir é mostrada uma instrução SQL com `outer join` , em que o otimizador utilizou o método de junção `HASH` , como pode ser visto no plano de execução. Essa instrução SQL tem como objetivo mostrar todos os clientes com limites de crédito superiores a 1.000 dólares. Uma `outer join` é necessária para que a instrução retorne os clientes que não têm pedidos. Para este exemplo, da forma como foi escrita a instrução, a tabela externa será a de clientes, já a tabela interna será a de pedidos. O resultado esperado será o retorno de linhas de clientes, mesmo aquelas sem linhas correspondentes na tabela de pedidos.

```
SQL> select cust_last_name
  2         ,sum(nvl2(o.customer_id,0,1)) "Count"
  3  from   customers c
  4        ,orders    o
  5  where  c.credit_limit > 1000
  6  and    c.customer_id = o.customer_id(+)
  7  group  by cust_last_name;

Execution Plan
----------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |           |       |       |     7 (100)|          |
|  1 |  HASH GROUP BY      |           |   168 |  3192 |     7  (29)| 00:00:01 |
|* 2 |   HASH JOIN OUTER   |           |   318 |  6042 |     6  (17)| 00:00:01 |
|* 3 |    TABLE ACCESS FULL| CUSTOMERS |   260 |  3900 |     3   (0)| 00:00:01 |
|* 4 |    TABLE ACCESS FULL| ORDERS    |   105 |   420 |     2   (0)| 00:00:01 |
----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("C"."CUSTOMER_ID"="O"."CUSTOMER_ID")
   3 - filter("C"."CREDIT_LIMIT">1000)
   4 - filter("O"."CUSTOMER_ID">0)
```

Figura 5.15: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img12code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nesse exemplo, poderíamos utilizar uma subquery `not exists` para retornar as linhas, que chegaríamos ao mesmo resultado de linhas. No entanto, como estamos consultando todas as linhas da tabela, já que `FULL TABLE SCANS` estão sendo realizados em cada tabela, a junção `HASH` é a mais adequada (a menos que o otimizador, para a subquery `not exists`, não optasse por um método `NESTED LOOPS`).

Quando estamos trabalhando com `views` e submetendo-as a condições `outer join`, o otimizador não pode operar como uma junção normal como entre duas tabelas. Portanto, primeiramente, ele gerará uma fonte de dados referente a `view`, retornando todas as linhas dela. Somente depois disso, em cima desta fonte de linha gerada, é que o otimizador aplicará a junção `outer join` com a tabela interna.

Quando isso acontecer, considere utilizar as tabelas da `view` em vez da própria `view`, caso você perceba que este é o motivo do problema de performance, ou apenas visando à prevenção futura. Às vezes acabamos utilizando uma determinada `view` por ser mais prático, mesmo não necessitando das informações de outras tabelas que por ventura podem estar compondo a `view`. Veja o exemplo a seguir.

Segue script da `view` V_ORDERS:

```
create or replace view v_orders as
select l.product_id
      ,sum(l.quantity*unit_price) revenue
        ,o.order_id
        ,o.customer_id
from   orders o
      ,order_items l
where  o.order_id = l.order_id
group  by l.product_id
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
        ,o.order_id
        ,o.customer_id;
```

Seguem a instrução SQL e seu plano de execução:

```
SQL> select c.cust_last_name
  2        ,sum(revenue)
  3  from   customers c
  4        ,v_orders o
  5  where  c.credit_limit   > 2000
  6  and    o.customer_id(+) = c.customer_id
  7  group  by c.cust_last_name;

Execution Plan
----------------------------------------------------------------------------
| Id  | Operation              |  Name        | Rows  | Bytes | Cost (%CPU)|
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |              |   144 |  4608 |    16  (32)|
|   1 |  HASH GROUP BY         |              |   144 |  4608 |    16  (32)|
|*  2 |   HASH JOIN OUTER      |              |   663 | 21216 |    15  (27)|
|*  3 |    TABLE ACCESS FULL   | CUSTOMERS    |   195 |  2925 |     6  (17)|
|   4 |    VIEW                | V_ORDERS     |   665 | 11305 |            |
|   5 |     HASH GROUP BY      |              |   665 | 15960 |     9  (34)|
|*  6 |      HASH JOIN         |              |   665 | 15960 |     8  (25)|
|*  7 |       TABLE ACCESS FULL| ORDERS       |   105 |   840 |     4  (25)|
|   8 |       TABLE ACCESS FULL| ORDER_ITEMS  |   665 | 10640 |     4  (25)|
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("O"."CUSTOMER_ID"(+)="C"."CUSTOMER_ID")
   3 - filter("C"."CREDIT_LIMIT">2000)
   6 - access("O"."ORDER_ID"="L"."ORDER_ID")
   7 - filter("O"."CUSTOMER_ID">0)
```

Figura 5.16: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img13code.txt

Visualizando o plano de execução, o otimizador escolhe realizar uma junção `HASH` entre as tabelas da `view` `v_orders` : `orders` e `orders_items` . Além de realizar uma junção `HASH` , ele já faz isso agrupando os dados, como está descrito no script original da `view` .

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

SORT MERGE OUTER joins

Quando uma `outer join` não consegue utilizar a tabela externa (preserved rows) como tabela controle para conduzir a tabela interna (opcional), neste caso, ele não pode utilizar junções `HASH` ou `NESTED LOOPS`. Nessas situações, ele utiliza junções `SORT MERGE OUTER`. O otimizador utilizará esse tipo de junção nos seguintes casos:

- Quando uma junção `NESTED LOOPS` é ineficiente. Uma junção `NESTED LOOPS` pode ser ineficiente por conta do volume de dados.
- Quando o otimizador acha que é mais barato usar uma junção `SORT MERGE` em vez de uma junção `HASH` devido a ordenações existentes derivadas de outras operações.

A seguir um exemplo de uma junção `SORT MERGE OUTER`. Utilizamos o `hint` `use_merge` para forçar um plano utilizando este tipo de método.

```
SQL> select /*+ USE_MERGE(e d) */ ename, dname, d.deptno, e.deptn
o
  2  from   emp e
  3        ,dept d
  4  where  d.deptno = e.deptno(+)
  5  order  by e.sal;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2089385360

-----------------------------------------------------------------------------------------
| Id  | Operation                      | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |            |    14 |   364 |     6  (34)| 00:00:01 |
|   1 |  SORT ORDER BY                 |            |    14 |   364 |     6  (34)| 00:00:01 |
|   2 |   MERGE JOIN OUTER             |            |    14 |   364 |     5  (20)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| DEPT       |     4 |    52 |     2   (0)| 00:00:01 |
|   4 |     INDEX FULL SCAN            | PK_DEPTNO  |     4 |       |     1   (0)| 00:00:01 |
|*  5 |    SORT JOIN                   |            |    14 |   182 |     3  (34)| 00:00:01 |
|   6 |     TABLE ACCESS FULL          | EMP        |    14 |   182 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("D"."DEPTNO"="E"."DEPTNO"(+))
       filter("D"."DEPTNO"="E"."DEPTNO"(+))
```

Figura 5.17: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img14code.txt

`FULL OUTER joins`

Uma `FULL OUTER join` é uma combinação de `outer joins`, tanto de uma `left join` quanto de uma `right join`. Além das linhas com correspondência (`inner join`), as linhas de ambas as tabelas que não foram retornadas do resultado da `inner join` são preservadas e mostradas com valores nulos onde não há correspondência. Em outras palavras, `FULL OUTER JOINS` unem tabelas em conjunto, mas mostram linhas com ou sem correspondências dentre as tabelas relacionadas.

No exemplo a seguir, são recuperados todos os departamentos e todos os empregados em cada departamento, mesmo aqueles empregados sem departamentos, bem como os departamentos que não possuem empregados.

```
SQL> select d.department_id
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
 2         ,e.employee_id
 3  from    employees e full outer join departments d
 4  on      e.department_id = d.department_id
 5  order   by d.department_id;

Execution Plan
----------------------------------------------------------
Plan hash value: 3770333139

-----------------------------------------------------------------------------------------
| Id  | Operation                 | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |            |   122 |  3172 |     6  (34)| 00:00:01 |
|   1 |  SORT ORDER BY            |            |   122 |  3172 |     6  (34)| 00:00:01 |
|   2 |   VIEW                    | VW_FOJ_0   |   122 |  3172 |     5  (20)| 00:00:01 |
|*  3 |    HASH JOIN FULL OUTER   |            |   122 |  1342 |     5  (20)| 00:00:01 |
|   4 |     INDEX FAST FULL SCAN| DEPT_ID_PK |    27 |   108 |     2   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL     | EMPLOYEES  |   107 |   749 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Figura 5.18: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img15code.txt

A partir do banco Oracle 11g, o otimizador usa automaticamente um método de execução nativo baseado em `HASH` para executar `full outer joins` sempre que possível. Quando o otimizador o utiliza, no plano de execução é mostrado o passo `hash join full outer`, como neste exemplo (Passo 3). A operação `hash join full outer`, apresentada nesse plano de execução, indica que na instrução SQL o otimizador usou o método de execução `hash join full outer`. Normalmente, quando a condição `full outer join` entre duas tabelas for uma `equijoin`, o método `hash join full outer` é possível de ser utilizado, e o otimizador o usa automaticamente.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: O descritivo `HASH JOIN FULL OUTER` pode ser diferente dependendo da versão do Oracle. Contudo, essas diferenças podem estar na localização de cada palavra ou na não existência de algumas delas.

Para influenciar o otimizador a considerar `hash join full outer`, use o `hint` `NATIVE_FULL_OUTER_JOIN`. Para influenciar o não uso deste método, use o `hint` `NO_NATIVE_FULL_OUTER_JOIN`. O `hint` `NO_NATIVE_FULL_OUTER_JOIN` influencia o otimizador a não utilizar o método nativo ao juntar as tabelas. Em vez disso, uma `full outer join` pode ser executada utilizando uma junção `left outer` mais um `antijoin`, como podemos ver no exemplo a seguir:

```
SQL> select /*+ NO_NATIVE_FULL_OUTER_JOIN */ d.department_id
  2        ,e.employee_id
  3  from   employees e full outer join departments d
  4  on     e.department_id = d.department_id
  5  order  by d.department_id;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3122951222

-----------------------------------------------------------------------------------------------
| Id  | Operation              | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                    |   124 |  3224 |     4  (25)| 00:00:01 |
|   1 |  SORT ORDER BY         |                    |   124 |  3224 |     4  (25)| 00:00:01 |
|   2 |   VIEW                 |                    |   124 |  3224 |     3   (0)| 00:00:01 |
|   3 |    UNION-ALL           |                    |       |       |            |          |
|   4 |     NESTED LOOPS OUTER|                    |   107 |  1177 |     2   (0)| 00:00:01 |
|   5 |      TABLE ACCESS FULL| EMPLOYEES          |   107 |   749 |     2   (0)| 00:00:01 |
|*  6 |      INDEX UNIQUE SCAN| DEPT_ID_PK         |     1 |     4 |     0   (0)| 00:00:01 |
|   7 |     NESTED LOOPS ANTI |                    |    17 |   119 |     1   (0)| 00:00:01 |
|   8 |      INDEX FULL SCAN  | DEPT_ID_PK         |    27 |   108 |     1   (0)| 00:00:01 |
|*  9 |      INDEX RANGE SCAN | EMP_DEPARTMENT_IX  |    41 |   123 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   6 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID"(+))
   9 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Figura 5.19: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img16code.txt

`OUTER joins` com várias tabelas

A partir do banco de dados Oracle 12c, várias tabelas podem se relacionar externamente com uma tabela (`outer join`) na sua versão nativa, aquela em que a sintaxe inclui o sinal de positivo entre parênteses (+) ao lado da coluna em uma cláusula `where` . Em versões anteriores, isso não era permitido. Para realizar tal operação era preciso convertê-la para a sintaxe `ANSI` . Esse novo recurso da versão 12c permite que o Oracle possa gerar uma `view` (`CARTESIAN`) que contenha várias tabelas para então serem relacionadas a uma tabela com `outer join` . Veja os exemplos a seguir:

Executando a instrução na versão 11g:

```
SQL> select first_name, department_name, start_date, end_date
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  2  from   employees e, departments d, job_history j
  3  where  d.department_id = e.department_id(+)
  4  and    j.employee_id   = e.employee_id(+);
where  d.department_id = e.department_id(+)
                       *
ERROR at line 3:
ORA-01417: a table may be outer joined to at most one other table
```

Quando tentamos executar a instrução em um banco de dados Oracle 11g, o Oracle retorna o erro ORA-01417, informando que este tipo de junção não é permitido, pois uma tabela só pode se relacionar externamente com apenas uma tabela (e também não pode ser ela mesma). Executando a mesma instrução na versão 12c, temos o resultado a seguir:

```
SQL> select first_name, department_name, start_date, end_date
  2  from   employees e, departments d, job_history j
  3  where  d.department_id = e.department_id(+)
  4  and    j.employee_id   = e.employee_id(+);

Plan hash value: 4129722845
-----------------------------------------------------------------------------------------
| Id  | Operation        | Options      | Object      |Rows| Bytes |Cost(%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |              |             |  1 |    99 |    6 (0) | 00:00:01 |
|*  1 |  HASH JOIN       | OUTER        |             |  1 |    99 |    6 (0) | 00:00:01 |
|   2 |   MERGE JOIN     | CARTESIAN    |             |  1 |    61 |    4 (0) | 00:00:01 |
|   3 |    TABLE ACCESS  | STORAGE FULL | DEPARTMENTS |  1 |    30 |    2 (0) | 00:00:01 |
|   4 |    BUFFER        | SORT         |             |  1 |    31 |    2 (0) | 00:00:01 |
|   5 |     TABLE ACCESS | STORAGE FULL | JOB_HISTORY |  1 |    31 |    2 (0) | 00:00:01 |
|   6 |   TABLE ACCESS   | STORAGE FULL | EMPLOYEES   |  1 |    38 |    2 (0) | 00:00:01 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
1 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID"(+)
      AND "J"."EMPLOYEE_ID"="E"."EMPLOYEE_ID"(+))
```

Figura 5.20: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img17code.txt

A instrução acima foi executada em um banco de dados Oracle 12c e, como era esperado, ela ocorreu com sucesso. Veja no plano de execução que o otimizador, em resumo, gera um produto

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

cartesiano entre as tabelas `departments` e `job_history`, formando uma só fonte de dados. Depois disso, ele aplica um `hash outer join` entre a fonte de dados gerada e a tabela `employees`. Nesse caso, a fonte de dados gerada a partir das duas tabelas, `departments` e `job_history`, conduzirá a junção (`Driving Table`), pois essa fonte de dados acaba sendo a fonte externa, derivada das Outer Tables `departments` e `job_history`, obedecendo a regra para este tipo de método de junção. O otimizador identifica a `outer join` através do sinal de positivo ao lado da coluna referente à tabela onde pode não haver correspondência. A sintaxe nativa do Oracle é mais simples do que a sintaxe `ANSI`, onde há uma definição mais detalhada de quem são as Outer Tables e Inner Tables na instrução, e a direção que as define, `left` ou `right`.

Apenas a título de curiosidade, segue a instrução anterior, aquela em que o Oracle gerou um erro, mas agora convertida para o padrão `ANSI` para que seja possível sua execução em versões anteriores a 12c (incluindo a 12c), utilizando `outer join` com várias tabelas.

```
SQL> select first_name, department_name, start_date, end_date
  2  from   departments d left outer join employees e on (e.depar
tment_id=d.department_id)
  3      right outer join job_history j on (e.employee_id=j.emplo
yee_id);
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2328914618

-----------------------------------------------------------------------------------------
| Id  | Operation            | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |    10 |   620 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN OUTER     |             |    10 |   620 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL  | JOB_HISTORY |    10 |   200 |     2   (0)| 00:00:01 |
|   3 |   VIEW               |             |   106 |  4452 |     5  (20)| 00:00:01 |
|*  4 |    HASH JOIN OUTER   |             |   106 |  3180 |     5  (20)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| DEPARTMENTS |    27 |   432 |     2   (0)| 00:00:01 |
|   6 |     TABLE ACCESS FULL| EMPLOYEES   |   107 |  1498 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."EMPLOYEE_ID"(+)="J"."EMPLOYEE_ID")
   4 - access("E"."DEPARTMENT_ID"(+)="D"."DEPARTMENT_ID")
```

Figura 5.21: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img18code.txt

Nesse plano de execução gerado em um banco de dados 11g, podemos observar que o otimizador realiza uma junção `hash outer join` entre as tabelas `departments` e `employees`. A tabela `departments` é a Outer Table, enquanto a `employees` é a Inner Table. Sabemos disso, pois, na instrução SQL, estamos utilizando uma junção do tipo `left outer join` e a tabela da esquerda é a tabela `departments`. Portanto, seguindo as regras de `outer join` para métodos `hash`, a tabela externa (`outer table`) deve ser a tabela de condução. Neste caso, se observarmos o plano de execução, é exatamente isso que o otimizador fez, pois ela aparece primeiro. Logo após, o otimizador gera uma `view` desses dados e aplica uma nova operação de `hash`, agora entre a `view` e a tabela `job_history` gerando o resultado final. Veja que, para essa nova operação de `hash`, lendo a nossa instrução SQL novamente, vemos que há um `right outer join`

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

indicando a tabela da direita como sendo a tabela externa, neste caso, a tabela `job_history`. Como a tabela `job_history` é a tabela externa, logo ela foi a escolhida como a tabela de condução nesta junção `HASH`.

semijoins

Uma junção `semijoin` retorna linhas que correspondem a uma subconsulta do tipo `exists`, sem duplicar linhas do lado esquerdo do predicado quando várias linhas do lado direito satisfazem os critérios da subconsulta. Por exemplo:

```
SQL> select *
  2  from   departments
  3  where  exists (select *
  4                 from   employees
  5                 where  departments.department_id = employees.
department_id
  6                 and    employees.salary > 10000);
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2317224448

-----------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |    10 |   280 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN SEMI    |             |    10 |   280 |     5  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPARTMENTS |    27 |   567 |     2   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMPLOYEES   |    68 |   476 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("DEPARTMENTS"."DEPARTMENT_ID"="EMPLOYEES"."DEPARTMENT_ID")
   3 - filter("EMPLOYEES"."SALARY">10000)
```

Figura 5.22: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img19code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nessa instrução, apenas uma linha precisa ser retornada referente ao departamento, mesmo que muitas linhas referentes aos funcionários possam corresponder na subconsulta. Se não houver nenhum índice na coluna `salary`, um `semijoin` pode ser usado para melhorar o desempenho da consulta. No banco de dados em que essa instrução SQL foi executada, não havia um índice para a coluna `salary`, com isso, o otimizador optou por uma junção `semijoin`.

O otimizador usa um algoritmo de `NESTED LOOPS` por padrão para subconsultas com os operadores `IN` ou `exists` que não possam estar na consulta principal. No entanto, se os `hints` `MERGE_SJ`, `HASH_SJ` ou `NL_SJ` forem utilizados e algumas condições forem satisfeitas, a subconsulta pode ser alterada para um acesso via `SORT MERGE` ou `HASH semijoin`.

> **Nota:** Com relação a alterações que podem ser feitas em subconsultas, o otimizador só não conseguirá alterar para um acesso via `semijoin` caso a instrução SQL com subconsulta contenha uma condição `OR`. O mesmo também se aplica para casos em que a intenção do otimizador seja alterar para um acesso via `antijoins`.

antijoins

Um `antijoin` retorna linhas do lado esquerdo do predicado para o qual não há linhas correspondentes no lado direito do predicado. Ou seja, ele retorna linhas que não correspondem (`not in`) a subconsulta do lado direito. Por exemplo, um `antijoin`

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

pode selecionar uma lista de empregados que não estão em um determinado conjunto de departamentos:

```
SQL> select *
  2  from   employees
  3  where  department_id not in (select department_id
  4                               from   departments
  5                               where  location_id = 2500);

Execution Plan
----------------------------------------------------------
Plan hash value: 3077464062

-----------------------------------------------------------------------------------------
| Id  | Operation                 | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |             |    97 |  7372 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN RIGHT ANTI SNA|             |    97 |  7372 |     5  (20)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL       | DEPARTMENTS |     1 |     7 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL       | EMPLOYEES   |   107 |  7383 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("DEPARTMENT_ID"="DEPARTMENT_ID")
   2 - filter("LOCATION_ID"=2500)
```

Figura 5.23: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img20code.txt

O otimizador usa um algoritmo de `NESTED LOOPS` para subconsultas `not in` por padrão. No entanto, se o `hint` `MERGE_AJ`, `HASH_AJ` ou `NL_AJ` é utilizado e algumas condições forem satisfeitas, a subconsulta não correlacionada `not in` pode ser alterada para um `SORT MERGE` ou `hash antijoin`. Neste exemplo, um `hash antijoin` foi realizado para limitar a quantidade de empregados, trazendo somente aqueles que não estejam em departamentos onde a localização seja 2500.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

star queries

Alguns Data Warehouses são projetados em torno de um esquema em estrela, que inclui uma tabela chamada de tabela de fatos e várias tabelas de pequenas dimensões, chamada de tabela de pesquisa. A tabela de fatos armazena informações primárias. Cada tabela de pesquisa armazena informações sobre um atributo na tabela de fatos. Um exemplo seria uma tabela contendo dados de vendas como sendo a tabela de fatos e outras tabelas ligadas a ela com informações específicas sobre os Produtos, os Clientes e os Canais de Venda, cada uma ligada à tabela Vendas.

Portanto, uma `star query` é exatamente essa junção entre uma tabela de fatos e várias tabelas de pesquisa. Cada tabela de pesquisa é unida por suas chaves primárias às chaves estrangeiras correspondentes à tabela de fatos, mas as tabelas de pesquisa não são unidas entre si, lembrando a forma de uma estrela, por isso o nome `star query`. A tabela de fatos normalmente tem um índice concatenado nas colunas de chave estrangeira para facilitar esse tipo de junção, ou um índice de `bitmap` separado em cada uma dessas colunas.

Esse modelo só é reconhecido pelo otimizador baseado em custo. Desta forma, ele gerará planos de execução mais eficientes, voltados para esse tipo. Já no modo baseado em Regra, o otimizador não reconhece esse modelo, gerando planos de execução sem levar em conta as características deste tipo. Veja a seguir, parte de um `schema` de tabelas onde é mostrado um exemplo de uma `star query`:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

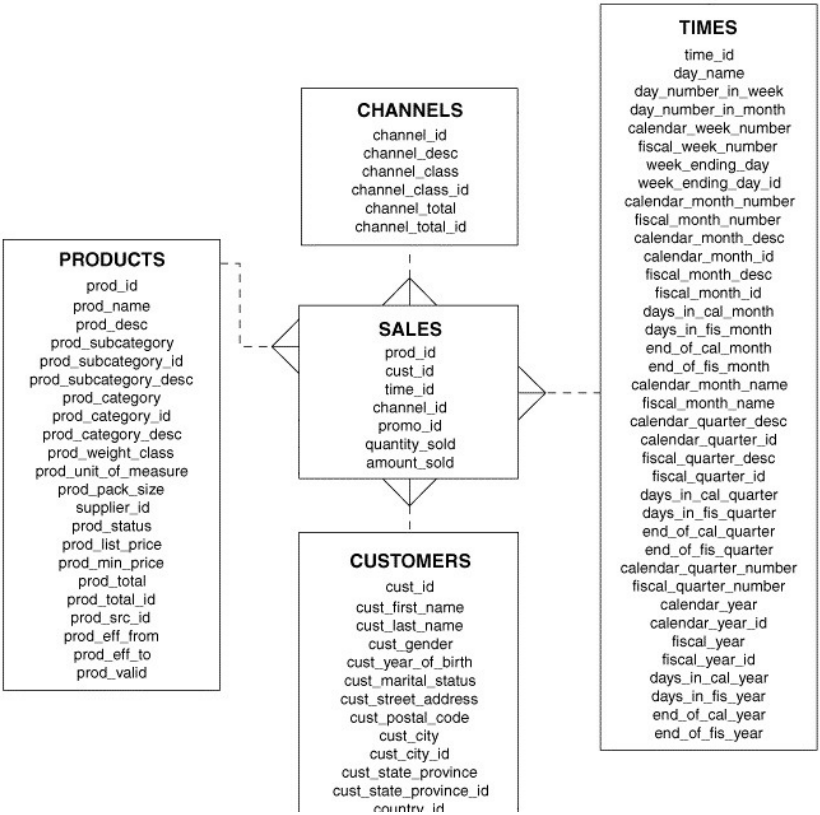

Figura 5.24: Exemplo de modelagem Star Query. Fonte: https://newbiedba.wordpress.com/2017/04/23/migrating-oracle-sample-schema-to-mongodb/

Um schema como esse do exemplo mostra uma tabela de fatos típica que contém campos-chave ligando-a a outras tabelas. A tabela de fatos contém dados de Vendas (`sales`) com as referências para dados de Clientes (`customers`), Tempo (`times`), Produto (`products`) e Mercado (`channels`). Nesse caso, para essas referências, existem tabelas de dimensão correspondentes para Cliente, Tempo, Produto e Mercado. A tabela de dimensão de Produto, por exemplo, conterá informações

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

sobre cada ID de produto que aparece na tabela de fatos, ou seja, cada tabela de pesquisa contém dados específicos de cada tipo de informação. A seguir um exemplo de plano de execução para uma instrução SQL recuperando dados de estrutura em estrela.

Instrução SQL:

```
SQL> select t.calendar_month_name
  2        ,p.prod_category
  3        ,c.channel_desc
  4        ,cr.cust_first_name
  5        ,sum(ps.amount_sold) as sum_of_sales
  6  from   times      t
  7        ,products   p
  8        ,sales      ps
  9        ,channels   c
 10        ,customers  cr
 11  where  t.time_id     = ps.time_id
 12  and    ps.prod_id    = p.prod_id
 13  and    ps.channel_id = c.channel_id
 14  and    ps.cust_id   = cr.cust_id
 15  group  by t.calendar_month_name
 16           ,p.prod_category
 17           ,c.channel_desc
 18           ,cr.cust_first_name;
```

Plano de execução:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name          | Rows |Bytes|Cost(%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |               |    1 |  166|   3  (34)| 00:00:01|
|   1 |  HASH GROUP BY                     |               |    1 |  166|   3  (34)| 00:00:01|
|   2 |   NESTED LOOPS                     |               |      |     |          |         |
|   3 |    NESTED LOOPS                    |               |    1 |  166|   2   (0)| 00:00:01|
|   4 |     NESTED LOOPS                   |               |    1 |  151|   2   (0)| 00:00:01|
|   5 |      NESTED LOOPS                  |               |    1 |  111|   2   (0)| 00:00:01|
|   6 |       NESTED LOOPS                 |               |    1 |   86|   2   (0)| 00:00:01|
|   7 |        TABLE ACCESS FULL           | SALES         |    1 |   61|   2   (0)| 00:00:01|
|   8 |        TABLE ACCESS BY INDEX ROWID| CHANNELS      |    1 |   25|   0   (0)| 00:00:01|
|*  9 |         INDEX UNIQUE SCAN          | CHANNELS_PK   |    1 |     |   0   (0)| 00:00:01|
|  10 |       TABLE ACCESS BY INDEX ROWID  | CUSTOMERS     |    1 |   25|   0   (0)| 00:00:01|
|* 11 |        INDEX UNIQUE SCAN           | CUSTOMERS_PK  |    1 |     |   0   (0)| 00:00:01|
|  12 |      TABLE ACCESS BY INDEX ROWID   | PRODUCTS      |    1 |   40|   0   (0)| 00:00:01|
|* 13 |       INDEX UNIQUE SCAN            | PRODUCTS_PK   |    1 |     |   0   (0)| 00:00:01|
|* 14 |     INDEX UNIQUE SCAN              | TIMES_PK      |    1 |     |   0   (0)| 00:00:01|
|  15 |    TABLE ACCESS BY INDEX ROWID     | TIMES         |    1 |   15|   0   (0)| 00:00:01|
----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   9 - access("PS"."CHANNEL_ID"="C"."CHANNEL_ID")
  11 - access("PS"."CUST_ID"="CR"."CUST_ID")
  13 - access("PS"."PROD_ID"="P"."PROD_ID")
  14 - access("T"."TIME_ID"="PS"."TIME_ID")
Note
-----
   - dynamic sampling used for this statement (level=2
```

Figura 5.25: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/5250316ee9eb07930c709aca8c651c8c118e753d/cap05img21code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Esse exemplo foi executado em um banco de dados Oracle XE, no qual os recursos como particionamento e índices `Bitmap` não estão disponíveis. Em bancos próprios, específicos para fins de armazenamento em grande volume, recursos como esses, para modelos como este, vão proporcionar quantidades maiores de opções de planos de execução e execuções mais eficientes. É preciso ter atenção, pois muitos desses recursos precisarão ser adquiridos, ou, caso eles já estejam instalados, podem iniciar algum tipo de cobrança ao serem utilizados.

Agora que conhecemos os tipos de junção, vamos nos aprofundar um pouco mais no plano de execução, entendendo como funcionam os métodos de acesso. Você vai ver que os assuntos são interligados e que são a base para todo o plano de execução.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 6

MÉTODOS DE ACESSO

Anteriormente, conhecemos os métodos de junção que são utilizados pelo otimizador para fazer a ligação, ou melhor dizendo, a junção, entre as tabelas em uma instrução SQL. Além dos métodos de junção, o otimizador precisa escolher os caminhos ou métodos de acesso disponíveis para localizar os dados de uma tabela ou de um índice, os chamados *Access Paths*. Para cada tabela mencionada em uma instrução SQL, pode haver vários métodos de acesso que poderão ser utilizados com o objetivo de localizar e recuperar as linhas de determinada tabela ou de seus índices. Os métodos de acesso serão melhor escolhidos com base na forma como a instrução SQL foi escrita e também pelos métodos de junção escolhidos pelo otimizador.

Os métodos de acesso determinam como os dados serão localizados e recuperados e, por consequência disso, trazem impacto direto na performance. Se um método não for bem escolhido, ele pode comprometer a performance de uma instrução SQL. Desta forma, não menos importante que qualquer assunto visto até aqui, os métodos de acesso devem ser compreendidos para que possamos, caso necessário, influenciar o otimizador na mudança de sua escolha.

Em algumas situações, os métodos de acesso podem ser

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

influenciados, por exemplo, ao utilizarmos `hints`, ou pela atualização ou não das estatísticas de tabelas e índices. Se as estatísticas estiverem atualizadas, o otimizador poderá se beneficiar dessas informações para poder escolher métodos que sejam mais eficientes. Caso contrário, o uso de estatísticas desatualizadas, que não mostram a real situação na qual se encontram tais objetos, pode trazer problemas de desempenho, começando pela escolha do método de acesso inadequado.

6.1 COMO O ARMAZENAMENTO E A LEITURA DOS REGISTROS ACONTECEM?

Não nos aprofundaremos neste assunto, mas vamos relembrar o que já conversamos em capítulos anteriores, como acontecem o armazenamento e a leitura de dados no Oracle. O Oracle armazena os dados em blocos. Geralmente, o tamanho dos blocos é definido com base nos limites máximos e mínimos permitidos pelo sistema operacional. Esses blocos possuem tamanhos fixos e, à medida que vão sendo preenchidos, outros blocos são alocados. Geralmente, o tamanho padrão do bloco é de 8k de armazenamento, mas esse tamanho pode variar dependendo do sistema operacional ou do objetivo do banco de dados.

Quando estamos falando de tabelas transacionais, as operações nesses blocos são bastante intensas, pois, geralmente, diversos comandos de manipulação afetam esses blocos, como `Inserts`, `Updates` e `Deletes`. O Oracle não reutiliza blocos que foram utilizados por uma determinada tabela mesmo quando esses blocos contêm dados que não são mais utilizados. O Oracle mantém um histórico dos blocos utilizados por uma tabela e cria uma marcação

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

para cada tabela chamada de *High Water Mark* (HWM).

Essa marcação indica o último bloco utilizado pela tabela no momento em que os dados foram armazenados. A marcação vai se deslocando, ou seja, pulando para o próximo bloco do segmento, à medida que os dados são criados em uma tabela. Mesmo que sejam excluídos, essa marcação continuará apontando para o último bloco alocado. Veja a instrução a seguir, que recupera da `view` `DBA_TABLES` o indicador do último bloco utilizado por cada uma das tabelas `emp` e `dept` no banco de testes utilizado para este livro.

```
SQL> select table_name, blocks from dba_tables where table_name i
n ('EMP', 'DEPT');

TABLE_NAME                          BLOCKS
------------------------------ ----------
EMP                                     1
DEPT                                    1
```

Com isso, esses blocos podem ficar fragmentados, pois em um mesmo bloco podem existir dados referentes a outras tabelas ou apenas endereçamentos, ou blocos inteiros marcados como não utilizados. Por isso, de tempos em tempos é necessário fazer uma desfragmentação desses blocos. Um exemplo disso foi falado lá no capítulo sobre índices, quando mencionamos o processo de `rebuild` . Outra forma de limpar essas marcações seria excluir a tabela (`drop table`) ou dar um `TRUNCATE` nela. Este último, além de excluir todos os dados, liberará os espaços marcados, mas lembre-se de que para este comando não há `rollback` , pois ele é seguido de um `commit` implícito executado pelo próprio Oracle. Aqui não vamos nos aprofundar nos detalhes mais técnicos da HWM, mas imagino que já deu para você entender aonde quero chegar. Enquanto esses espaços não forem liberados, eles

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

continuarão sendo lidos pelo Oracle, mesmo se tratando de dados históricos e que não são mais pertinentes à pesquisa. Portanto, além de ser um desperdício de memória, isso torna o processo de leitura sequencial muito mais lento pois, ao executar tais leituras, lerá blocos sem dados ou ainda blocos com dados do passado que não terão a menor importância para a pesquisa em questão.

Pois bem, depois de devidamente armazenados, sempre que o Oracle precisa recuperar uma informação, ele faz a leitura de um desses blocos e o aloca na memória. Quando o Oracle faz essa alocação em memória, ele aloca um bloco inteiro, independentemente de o bloco conter ou não somente dados referentes a tabelas utilizadas em uma determinada instrução SQL, como já foi dito. Ele armazena na memória porque os dados nesta área são muito mais fáceis de serem manipulados e muito mais rápidos de serem acessados. Depois de carregado na memória, ele seleciona do bloco apenas os registros que lhe convém, aqueles que satisfazem às condições da instrução SQL.

Mais para a frente, quando falarmos sobre estatísticas, veremos que há um indicador referente ao número de leituras físicas e um outro referente a leituras de memória. É a essas leituras que estamos nos referindo agora. A leitura de blocos e não de dados individuais, digamos assim, faz com que menos leituras físicas sejam realizadas, o que levaria a um custo bem maior em relação à leitura realizada na memória. O que quero que você entenda com isso é que, dependendo de como o Oracle esteja fazendo o armazenamento e a leitura dos dados e respectivamente dos blocos, isso pode influenciar o otimizador na escolha pelo método de acesso.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

NOTA: Na leitura sequencial, o Oracle faz a leitura de todos os registros de uma tabela, verificando um a um quais são aqueles que atendem aos critérios de busca. Esse tipo de leitura pode ser útil quando a consulta retorna a maior parte dos registros de uma tabela, isso porque a leitura sequencial permite que sejam realizadas leituras de vários blocos de dados em uma única operação. Dessa forma, reduzir a quantidade de acessos ao disco trará benefícios no desempenho, já que esse tipo de acesso é uma operação lenta e custa muito ao otimizador. Já quando as leituras são realizadas através de índices, o acesso ao disco é limitado ao tamanho do bloco, o que pode ocasionar um menor desempenho em relação à leitura sequencial em casos específicos. Isso explica o porquê de em alguns cenários, nos quais há índices criados e bem utilizados em uma instrução SQL, o otimizador optar por não os utilizar. Um exemplo seria um plano de execução mostrando um acesso Full Table Scan, mesmo que a coluna de um determinado predicado seja uma coluna indexada.

Agora vamos ao tema propriamente dito. Você já deve ter visto os métodos de acesso anteriormente quando mostramos alguns exemplos de plano de execução. Então vamos ver as características de cada um deles e como funcionam.

6.2 FULL TABLE SCANS

Um acesso do tipo Full retorna linhas de uma tabela. Este

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

método faz com que o Oracle leia todas as linhas de uma tabela, examinando cada linha para determinar se a condição imposta pela cláusula `where` foi satisfeita, ou, no caso de não haver cláusula `where`, ele retornará todas as linhas lidas. De um ponto de vista mais específico, o Oracle lê todos os blocos de dados de forma sequencial. O tempo de leitura destes blocos poderá ser lento caso os dados estejam fragmentados em vários endereços distintos no disco.

Em resumo:

- Faz a leitura diretamente na tabela, passando por todos os registros contidos nela.
- Permite a leitura de `multiblock` I/O, com a qual é realizada a leitura de vários blocos de forma sequencial através de uma leitura simples de I/O.
- Permite o uso de paralelização.
- Embora sempre visto como vilão, dependendo do cenário, pode não ser uma opção tão inadequada.

Veja o plano de execução a seguir:

```
SQL> select salary
  2  from   employees
  3  where  salary > 4000;

Execution Plan
----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |       |       |     3 (100)|          |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |    98 |  6762 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
```

Figura 6.1: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img01code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Como não existe nenhum índice para a coluna de `salary` , o otimizador não pode usar um Index Range Scan, portanto uma varredura de tabela completa é realizada (`FULL TABLE SCAN`).

> **NOTA:** Dentro do plano de execução, o asterisco (*) ao lado de uma linha denota que existem informações referentes aos predicados com relação a acessos e filtros que estão sendo utilizados neste plano.

6.3 ROWID SCANS

Este método também retorna linhas da tabela, fazendo um acesso através de `rowids` . O Oracle obtém o `rowid` através da cláusula `where` ou através de acesso a índices que por ventura existam para as colunas da tabela que está sendo acessada. Contudo, isso ocorrerá desde que o otimizador escolha o `Access Path` através destes índices.

Em resumo:

- O acesso à tabela é através do `rowid` . Não necessita ler todos os registros da tabela.
- Este tipo de acesso é o mais rápido para acessar um dado.

É o método de acesso mais rápido disponível, pois o Oracle simplesmente recupera o bloco de dados especificado e extrai as linhas que satisfazem os critérios de seleção. É visto com frequência em planos de execução como `TABLE ACCESS BY ROWID` .

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Vamos ver um acesso direto através de `rowid`, em que ele é utilizado como predicado para a pesquisa:

```
SQL> select * from dept where rowid = ':x';

Execution Plan
----------------------------------------------------------
Plan hash value: 3453257278

--------------------------------------------------------------------------------
| Id  | Operation                   | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |     1 |    20 |     1   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY USER ROWID| DEPT |     1 |    20 |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

Figura 6.2: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img02code.txt

Agora vamos ver outro exemplo onde o acesso à tabela é realizado por `rowids` recuperados através das leituras de um índice:

```
SQL> select empno, ename from emp where empno = 10;

Execution Plan
----------------------------------------------------------
Plan hash value: 4024650034

--------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    10 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    10 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_PK |     1 |       |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

Figura 6.3: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img03code.txt

O `rowid` indica com precisão o local físico onde o registro está localizado. Ele mostra o arquivo de dados e o bloco em que a

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

linha de registro se encontra. Além disso, o `rowid` também informa qual é a posição do registro dentro do bloco. Com isso, o Oracle é capaz de ir diretamente ao registro desejado, sem precisar fazer qualquer outro tipo de verificação. Para que o otimizador utilize esse método de acesso, pelo menos uma das colunas acessadas não deve fazer parte do índice. Caso contrário, se todas as colunas acessadas estiverem indexadas, esse método de acesso provavelmente não será utilizado. Contudo, muito cuidado em utilizar a informação de `rowid` propriamente dita para operações de manipulação de dados em seus programas e instruções SQL, pois, mesmo que ele seja um identificador único e físico, ele pode perder a referência em caso de migração do banco de dados para discos diferentes, ou mesmo no caso de importação e exportação de tabelas.

6.4 INDEX SCANS

Quando a instrução SQL utiliza colunas indexadas na cláusula `where`, o otimizador faz uso desses índices disponíveis para o acesso rápido às informações. Se as colunas retornadas pela instrução também pertencerem a índices, então o Oracle poderá utilizar somente os índices em vez de acessar as tabelas.

Características:

- Acessa os dados da tabela através de índices.
- Costuma trabalhar com índices `B-tree`.

O acesso por índices pode acontecer das seguintes formas:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

INDEX UNIQUE SCAN

Um `SCAN` do tipo `UNIQUE` retorna apenas um `rowid`. Geralmente é utilizado quando estamos trabalhando com índices únicos ou chaves primárias. Trata-se de um método para pesquisar um único valor de chave através de um índice exclusivo. Seu retorno será sempre um valor único. Para esse tipo de acesso, você deve fornecer pelo menos a coluna principal do índice (a primeira coluna na sequência do índice) para acessar os dados através dele, no entanto, isso pode retornar mais de uma linha como resultado, o que não garante sua exclusividade.

```
SQL> select first_name
  2  from   employees
  3  where  employee_id = 100;

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |    11 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    11 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------------
```

Figura 6.4: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img04code.txt

INDEX RANGE SCAN

Um `SCAN` do tipo `RANGE` pode retornar zero ou mais `rowids`. É usado quando estamos trabalhando com índices não únicos na recuperação de um intervalo de busca. Este método é usado para acessar vários valores de coluna. Para este tipo, deve-se fornecer pelo menos a coluna principal do índice (a primeira coluna na sequência do índice) para acessar os dados através dele. Pode ser usado para operações de intervalo, como em operações de

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

maior que (>), menor que (<), diferente (<>), maior igual que (>=), menor igual que (<=) e between.

```
SQL> select first_name
  2  from   employees
  3* where  employee_id > 300;
```

```
Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |    11 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    11 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP_EMP_ID_PK |     1 |       |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------
```

Figura 6.5: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img05code.txt

No caso de um acesso por um índice *non-unique* em que podem ser retornados vários valores para um determinado valor de predicado, pode acontecer de uma varredura via `INDEX RANGE SCAN` ser realizada pelo otimizador, mesmo que nesse predicado haja uma operação de igualdade. Exemplo:

```
SQL> select mgr from emp where mgr = 5;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2539435852

---------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |     1 |     4 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| MGR_IDX |     1 |     4 |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------
```

Figura 6.6: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img06code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Neste outro exemplo, são mostrados dois acessos `INDEX RANGE SCAN` na mesma tabela para índices diferentes em uma operação do tipo `AND-EQUAL`.

```
SQL> select /*+ and_equal(emp emp_idx_1, job_index) */ EMP.*
  2  from   emp
  3  where  job    = 'ANALYST'
  4  and    deptno = 20;

Execution Plan
----------------------------------------------------------
Plan hash value: 720197672

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |     1 |    38 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMP       |     1 |    38 |     3   (0)| 00:00:01 |
|   2 |   AND-EQUAL                  |           |       |       |            |          |
|*  3 |    INDEX RANGE SCAN          | JOB_INDEX |     3 |       |     1   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | EMP_IDX_1 |     5 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 6.7: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img07code.txt

Nesse exemplo, a operação `AND-EQUAL` mescla os `rowids` obtidos pelas varreduras do `JOB_INDEX` e do `DEPTNO_INDEX`, resultando em um conjunto de `rowids` que satisfazem a instrução. Para esse exemplo, forçamos o `hint` `AND_EQUAL` para fazer a demonstração.

INDEX RANGE SCAN DESCENDING

Um acesso via `INDEX RANGE SCAN DESCENDING` é idêntico a um `INDEX RANGE SCAN`, exceto que os dados são retornados em ordem decrescente. Os índices, por padrão, são armazenados em ordem crescente. Geralmente, esse tipo de acesso é usado pelo otimizador quando existe uma ordenação de dados por ordem

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

decrescente. O intuito disso é para retornar os dados mais recentes primeiro ou para quando se procura um valor menor que um valor especificado.

```
SQL> select first_name
  2  from   employees
  3  where  employee_id > 300
  4  order  by employee_id desc;

Execution Plan
----------------------------------------------------------
Plan hash value: 1224472879

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows| Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |   1 |    11 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID | EMPLOYEES     |   1 |    11 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN DESCENDING| EMP_EMP_ID_PK |   1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 6.8: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img08code.txt

INDEX FULL SCAN

Um Scan do tipo `FULL` é usado quando em uma instrução SQL é usada pelo menos uma das colunas de um índice ou quando tal instrução não possui predicados, ou ainda se desejar evitar operações de `SORT`. Uma `FULL SCAN` está disponível se um predicado fizer referência a uma das colunas do índice. Contudo, o predicado não precisa ser a primeira coluna do índice (`index driver`). Um `FULL SCAN` pode ser usado para eliminar uma operação de ordenação (`SORT`), porque os dados são ordenados pela chave de índice. Outra característica desse tipo de acesso é que ele lê os blocos individualmente, o que pode se mostrar pouco eficiente em alguns casos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Uma leitura `FULL SCAN` também poderá ser utilizada quando não há nenhum predicado se ambas as condições a seguir forem atendidas:

- Todas as colunas na tabela referenciada na consulta estão incluídas no índice.
- Pelo menos uma das colunas de índice não é nula.

De modo geral, quando são realizadas leituras através de índices, é possível que eles sejam verificados por completo (`FULL SCAN`) em vez de uma varredura por faixa (`RANGE SCAN`), isto, onde não há a existência de predicados para a tabela. Acessos do tipo `INDEX FULL SCAN` estão disponíveis apenas quando o otimizador estiver operando em modo CBO, caso contrário, ficaria mais complicado determinar se uma verificação completa seria eficiente ou não. Desta forma, a escolha por este método se mostra mais interessante quando o otimizador conta com estatísticas que indicam que ele vai ser mais eficiente do que um acesso via `FULL TABLE SCAN` ou um `SORT` , por exemplo.

Em um outro cenário, podemos esperar que um acesso via `INDEX FULL SCAN` seja realizado quando uma varredura ilimitada de um índice é feita e queremos que os dados recuperados sejam ordenados já pela ordem em que este índice está. Com isso, o otimizador, nesse caso, pode decidir que selecionar todos os dados a partir do índice e não os ordenar é mais eficiente do que fazer um `FULL TABLE SCAN` ou um `FAST FULL INDEX SCAN` e, em seguida, ordenar.

```
SQL> select empno, ename from emp order by empno, ename;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 4088573028

---------------------------------------------------------------------------
| Id  | Operation        | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT |          |    14 |   140 |     1   (0)| 00:00:01 |
|   1 |  INDEX FULL SCAN | EMP_IDX2 |    14 |   140 |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------
```

Figura 6.9: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img09code.txt

O Oracle localiza o primeiro `leaf block` de índice e, em seguida, prossegue varrendo outros `leaf nodes`. Para cada entrada de índice, o Oracle obtém o `rowid` da entrada e, em seguida, recupera a linha da tabela especificada pelo `rowid`. Como o índice é ordenado por `empno` e `ename`, o Oracle evita uma operação separada para ordenar as linhas recuperadas.

FAST FULL SCAN

A `FAST FULL SCAN` é uma alternativa a uma `FULL TABLE SCAN` quando o índice contém todas as colunas necessárias para a instrução SQL e pelo menos uma das colunas da chave de índice tem a restrição `not null`. Um `FAST FULL SCAN` acessa os dados no próprio índice, sem acessar a tabela. Ele não pode ser usado para eliminar uma operação de `SORT`, porque os dados não são ordenados pela chave de índice. Ele lê o índice inteiro usando leituras `multiblock`, ao contrário de um `FULL SCAN`, e pode ser paralelizado. Veja o exemplo:

```
SQL> select /*+ INDEX_FFS(emp EMP_IDX2) */ empno, ename from emp;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2700697847

---------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |    14 |   140 |     2   (0)| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| EMP_IDX2 |    14 |   140 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------------
```

Figura 6.10: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img10code.txt

Esse tipo de acesso pode ser sugerido usando o `hint INDEX_FFS` (usado nesse exemplo). Ele foi bem-sucedido, porque todas as colunas que foram selecionadas são as colunas que fazem parte do índice. Mesmo se estivesse sendo selecionada apenas uma das duas colunas, esse tipo de leitura também funcionaria, pois ainda estaríamos selecionando somente dados do índice. Veja o mesmo exemplo, agora selecionando apenas o nome do empregado.

```
SQL> select /*+ INDEX_FFS(emp EMP_IDX2) */ ename from emp;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2700697847

---------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |    14 |    84 |     2   (0)| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| EMP_IDX2 |    14 |    84 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------------
```

Figura 6.11: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img11code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

INDEX SKIP SCAN

Um acesso via `INDEX SKIP SCAN` ocorre quando a coluna inicial de um índice composto é "ignorada" ou não especificada na instrução SQL.

O otimizador considera utilizar o método `INDEX SKIP SCAN`, pois, muitas vezes, pular a varredura de alguns blocos de índice pode ser mais rápido do que fazer uma varredura completa na tabela (`FULL TABLE SCAN`) ou ainda mais rápido do que executar varreduras completas no índice (`INDEX FULL SCANS`).

Para que o otimizador considere utilizar um `INDEX SKIP SCAN`, os seguintes critérios precisam ser atendidos:

- A coluna principal de um índice composto não deve estar especificada no predicado da instrução SQL. Por exemplo, o predicado da instrução SQL não faz referência à coluna `CUST_GENDER` que faz parte da chave de índice composta `CUST_GENDER` e `CUST_EMAIL`.
- Existirem poucos valores distintos na coluna principal do índice composto, mas nas colunas subsequentes existirem vários valores distintos. Por exemplo, para a chave de índice composta pelas colunas `CUST_GENDER` e `CUST_EMAIL`, a coluna `CUST_GENDER` possuir apenas dois valores distintos, enquanto a coluna `CUST_EMAIL` tem milhares deles.

O método `INDEX SKIP SCANS` funciona realizando uma divisão lógica de um índice composto de subíndices menores. O número de valores distintos nas colunas principais do índice determina o número de subíndices lógicos que serão criados.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Quanto menor o número de valores distintos, menos subíndices lógicos o otimizador precisará criar e mais eficiente a varredura se torna. A varredura lê cada índice lógico separadamente e "salta" blocos de índice que não atendem à condição de filtro referente à coluna utilizada no predicado (*non-leading column*), ou seja, a coluna que não é a primeira do índice.

A seguir, um exemplo do método `INDEX SKIP SCAN`:

```
SQL> select ename, empno from emp where ename = 'TURNER';

Execution Plan
----------------------------------------------------------
Plan hash value: 2936864451

-----------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     1 |    10 |     1   (0)| 00:00:01 |
|*  1 |  INDEX SKIP SCAN  | EMP_IDX_2 |     1 |    10 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
```

Figura 6.12: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img12code.txt

Nesse exemplo, temos um índice composto pelas colunas `empno` e `ename`, disposto nesta ordem. Note que estamos utilizando apenas a coluna `ename` como predicado e não estamos utilizando em nenhum momento a coluna `empno`, que é coluna principal do índice. Com isso, o otimizador escolheu utilizar a varredura pelo índice, mas fez isso através do método de acesso `INDEX SKIP SCAN`, como mostrado.

Uma opção que poderia ser levada em conta, caso um problema de performance fosse detectado devido a esse tipo de acesso, seria tentar considerar a possibilidade de utilizar as colunas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

principais do índice, também, como predicados na instrução SQL. Isso poderia fazer com que o otimizador optasse por caminhos mais interessantes e mais "baratos" para recuperar os dados.

> **NOTA:** Tome o cuidado ao escrever a instrução SQL, pois se a coluna principal de um índice estiver sendo utilizada como predicado na instrução, ela pode ser ignorada caso alguma operação esteja sendo realizada em cima dela, como `to_char`, `to_number` ou outra função do gênero que venha a transformar o dado.
>
> **Nota:** Para este exemplo, foi criado o seguinte índice:
>
> ```
> create index emp_idx_2 on emp (empno, ename);
> ```

INDEX JOIN

Um `INDEX JOIN SCANS` é uma `HASH JOIN` referente a vários índices que, em conjunto, retornam todas as colunas solicitadas por uma instrução. O banco de dados não precisa acessar a tabela neste caso, porque todos os dados são recuperados através dos índices. O otimizador considera utilizar este método de acesso em alguns casos, para evitar o acesso à tabela, economizando em termos de custo. Ele vai considerar este método nas seguintes circunstâncias:

- Quando uma `HASH JOIN` de vários índices recupera todos os dados solicitados pela consulta sem exigir acesso à tabela.
- Quando o custo de recuperar as linhas da tabela for maior

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

do que ler a `hash` de índices. Por outro lado, o custo de um `INDEX JOIN` pode ser frequentemente alto. Por exemplo, a operação de varrer dois índices e juntá-los pode ser, na maioria das vezes, mais dispendioso do que escolher o índice mais seletivo e, em seguida, testar a tabela. Contudo, dependendo do cenário, o otimizador pode optar por escolhê-lo.

Veja um exemplo a seguir no plano de execução:

```
SQL> select /*+ INDEX_JOIN(employees) */
  2         last_name
  3        ,email
  4  from   employees
  5  where  last_name like 'A%'
  6  /

Execution Plan
----------------------------------------------------------
Plan hash value: 325493464

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                  |     3 |    48 |     3  (34)| 00:00:01 |
|*  1 |  VIEW                  | index$_join$_001 |     3 |    48 |     3  (34)| 00:00:01 |
|*  2 |   HASH JOIN            |                  |       |       |            |          |
|*  3 |    INDEX RANGE SCAN    | EMP_NAME_IX      |     3 |    48 |     1   (0)| 00:00:01 |
|   4 |    INDEX FAST FULL SCAN| EMP_EMAIL_UK     |     3 |    48 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 6.13: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img13code.txt

O Oracle recupera todos os `rowids` no índice `EMP_EMAIL_UK` e, em seguida, recupera `rowids` em `EMP_NAME_IX` para sobrenomes que começam com a letra `A`. O banco de dados usa um `HASH JOIN` para pesquisar os dois conjuntos de `rowids` buscando as correspondências. Depois

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

disso, o Oracle verifica na tabela de empregados os registros correspondentes as estes `rowids`.

> **NOTA:** Para forçar este método de acesso, você pode utilizar o `hint INDEX_JOIN`.

> **NOTA:** Para este exemplo, foram criados os seguintes índices:
>
> ```
> create index emp_name_ix on employees (last_name, first_name
> );
>
> create index emp_email_ix on employees (email);
> ```

6.5 BITMAP SCANS

No capítulo sobre índices, vimos que em um índice de `bitmap`, a chave é a combinação dos dados indexados com um intervalo de `rowids`.

O banco de dados armazena pelo menos um `bitmap` (mapa de bits) para cada chave de índice. Cada valor no `bitmap` (que consiste de uma série de valores 1 e 0) aponta para uma linha dentro de um Range de `rowids`. Assim, em um índice de `bitmap`, cada entrada do índice aponta para um conjunto de linhas em vez de uma única linha.

Também vimos que uma junção por `bitmap` usa um mapa de bits para valores de chave e usa uma função de mapeamento que

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

converte cada posição de bit em um `rowid`. Os `bitmaps` podem mesclar, de forma eficiente, índices que correspondem a várias condições em uma cláusula `where` usando operações booleanas para resolver condições como `and` e `or`.

Para esse tipo de índice também existem métodos de acesso que podem ser utilizados pelo otimizador. A seguir, teremos uma breve descrição desses métodos. Não vamos nos aprofundar nas características de cada um, pois não os vemos com frequência. No entanto, isso dará a você a ideia de como trabalham. Você encontrará mais detalhes na documentação disponível pelo Oracle no próprio site.

Eis alguns métodos referentes a `BITMAP SCAN`:

BITMAP CONVERSION TO ROWID

Uma conversão de `bitmap` deriva de uma entrada de `bitmap` e uma linha em uma tabela. A conversão pode ser de uma entrada de `bitmap` para `rowid` (`TO ROWID`), ou de um `rowid` para entrada de `bitmap` (`FROM ROWID`). Estamos listando aqui como método de acesso, mas na verdade, caberia mais como uma operação após recuperar os dados utilizando um método de acesso, propriamente dito.

```
SQL> select cust_last_name, cust_first_name
  2  from   customers
  3  where  cust_year_of_birth < 1918;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------------------------------------------
|Id| Operation                          |Name             |Rows|Bytes|Cost(%CPU)| Time     |
----------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |                 |    |     |421 (100) |          |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED|CUSTOMERS        |3604|68476|421   (1) | 00:00:01|
| 2|   BITMAP CONVERSION TO ROWIDS      |                 |    |     |          |          |
|*3|    BITMAP INDEX RANGE SCAN         |CUSTOMERS_YOB_BIX|    |     |          |          |
----------------------------------------------------------------------------------------------
```

Figura 6.14: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img14code.txt

Esse plano mostra que o Oracle usa um `RANGE SCAN` para localizar todos os valores de chave menores que `1918` (Etapa 3). Converte os valores 1 (Verdadeiro) do `bitmap` para `rowids` (Etapa 2) e, em seguida, usa os `rowids` para obter as linhas da tabela `customers` (Passo 1).

BITMAP INDEX SINGLE VALUE

Esse tipo de método de acesso usa um índice `bitmap` para procurar um único valor de chave. O otimizador considera usar este método quando o predicado contém um operador de igualdade. Seu funcionamento consiste em, com uma instrução SQL, varrer um `bitmap` único (`SINGLE`) procurando por posições que contém um valor 1 (verdadeiro, aquele que condiz com o predicado). Os valores encontrados são convertidos em `rowids` e, em seguida, estes `rowids` são usados para localizar os registros.

```
SQL> select *
  2  from   customers
  3  where  cust_marital_status = 'Widowed'
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------------------------------------
|Id| Operation                          | Name                |Rows|Bytes|Cost (%CPU)| Time|
----------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |                     |    |    |412 (100)|        |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED|CUSTOMERS            |3461|638K|412   (2)|00:00:01|
| 2|   BITMAP CONVERSION TO ROWIDS      |                     |    |    |         |        |
|*3|    BITMAP INDEX SINGLE VALUE       |CUSTOMERS_MARITAL_BIX|    |    |         |        |
----------------------------------------------------------------------------------------
```

Figura 6.15: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img15code.txt

Esse plano mostra que o Oracle lê a entrada com a chave igual a `'Widowed'` no índice de `bitmap` de `customers` (etapa 3). Depois converte os valores 1 (verdadeiros) no `bitmap` para os `rowids` (etapa 2) e usa os `rowids` para obter as linhas da tabela `customers` (passo 1).

BITMAP INDEX RANGE SCANS

Esse tipo de método de acesso usa um índice de `bitmap` para procurar um intervalo de valores. O otimizador considera este método de acesso quando o predicado seleciona um intervalo de valores. Este tipo de varredura funciona de forma semelhante ao método `INDEX RANGE SCAN`.

```
SQL> select cust_last_name, cust_first_name
  2  from   customers
  3  where  cust_year_of_birth < 1918
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------------------------------------
|Id| Operation                          | Name             |Rows|Bytes|Cost(%CPU)|Time    |
----------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |                  |    |     |421 (100)|         |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS        |3604|68476|421   (1)|00:00:01 |
| 2|   BITMAP CONVERSION TO ROWIDS      |                  |    |     |         |         |
|*3|    BITMAP INDEX RANGE SCAN         | CUSTOMERS_YOB_BIX|    |     |         |         |
----------------------------------------------------------------------------------------
```

Figura 6.16: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img16code.txt

Esse plano mostra que o Oracle obtém todos os `bitmaps` para as chaves de `CUST_YEAR_OF_BIRTH` que sejam inferiores a `1918` (etapa 3). Depois converte os `bitmaps` para `rowids` (etapa 2) e, em seguida, obtém as linhas (etapa 1).

BITMAP MERGE

Esse caminho de acesso faz o merge de vários `bitmaps` e retorna um único `bitmap` como resultado. Normalmente, o otimizador considera usar um `BITMAP MERGE` para combinar `bitmaps` gerados a partir de uma varredura de intervalo de índice. O merge é realizado através da operação `or` entre dois `bitmaps`. O `bitmap` resultante desta operação seleciona todas as linhas do primeiro `bitmap`, além de todas as linhas de cada `bitmap` subsequente até passar por todos os `bitmaps`.

```
SQL> select cust_last_name, cust_first_name
  2  from   customers
  3  where  cust_gender = 'F'
  4  and    cust_year_of_birth < 1918
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------------------------------------------
|Id| Operation                          | Name                 |Rows|Bytes|Cost(%CPU)| Time   |
----------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |                      |    |     |288 (100)|        |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED|CUSTOMERS             |1802|37842|288   (1)|00:00:01|
| 2|   BITMAP CONVERSION TO ROWIDS      |                      |    |     |         |        |
| 3|    BITMAP AND                      |                      |    |     |         |        |
|*4|     BITMAP INDEX SINGLE VALUE      |CUSTOMERS_GENDER_BIX|    |     |         |        |
| 5|     BITMAP MERGE                   |                      |    |     |         |        |
|*6|      BITMAP INDEX RANGE SCAN       |CUSTOMERS_YOB_BIX   |    |     |         |        |
----------------------------------------------------------------------------------------------
```

Figura 6.17: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img17code.txt

Esse plano mostra que o Oracle obtém todos os `bitmaps` para as chaves de `CUST_YEAR_OF_BIRTH` que sejam inferiores a `1918` (etapa 6) e, em seguida, mescla esses `bitmaps` para criar um único `bitmap` (etapa 5). Depois obtém um único `bitmap` para a chave `CUST_GENDER` de `'F'` (etapa 4) e, em seguida, executa uma operação `and` nesses dois `bitmaps` para gerar um único `bitmap` que contém valores 1 para as linhas desejadas (etapa 3).

> **Nota:** Se você estiver utilizando o banco de dados Oracle XE, ele não suporta o uso de índices `bitmap`. Dessa forma, você não conseguirá reproduzir cenários como os apresentados nestes exemplos.

6.6 CLUSTER SCANS

Esse método poderá ser usado quando as informações da tabela estiverem armazenadas em `cluster`. Um `INDEX CLUSTER` tem como base chaves de `cluster` utilizando índices para localizar

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

dados. O `INDEX CLUSTER` é como um índice `B-tree`, mas de chaves de `cluster` e não de dados propriamente ditos. Uma varredura por `cluster` (`CLUSTER SCAN`) recupera todas as linhas que têm o mesmo valor de chave de `cluster`, referente a uma tabela armazenada em um `cluster` indexado. O otimizador considera esse tipo de método quando uma instrução SQL acessa uma tabela que pertence a um `cluster` indexado.

Em um `INDEX CLUSTER`, todas as linhas são armazenadas com o mesmo valor de chave de `cluster` em um mesmo bloco de dados. Por exemplo, se duas tabelas, a de empregados e a de departamentos, estiverem agrupadas em um determinado `cluster` e a chave deste `cluster` for o atributo referente ao ID do departamento (comum em ambas as tabelas), então o banco de dados armazenará todos os empregados com departamento 10 no mesmo bloco, todos do departamento 20 no mesmo bloco, e assim por diante. Isso, porque o índice baseado em `cluster` associa o valor da chave de `cluster` ao endereço de Bloco do Banco de Dados (chamado de Database Block Address - DBA) do bloco que contém os dados. Por exemplo, a entrada de um índice para a chave 30 mostra o endereço do bloco que contém linhas referentes aos empregados do departamento 30.

Quando são solicitadas linhas que estão em um `cluster`, o banco de dados varre o índice para obter os DBAs dos blocos que contêm as linhas. Com isso, o banco de dados Oracle localiza essas linhas com base nesses DBAs. Veja um exemplo a seguir.

```
-- cria um Cluster para armazenar IDs de Departamentos.
SQL> create cluster emp_dept_cluster (deptno number(2)) size 512;

-- cria um índice para o Cluster (Index Cluster) recém-criado.
SQL> create index idx_emp_dept_cluster on cluster emp_dept_cluste
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
r;

-- cria a tabela `emp_curso` com base na tabela `emp` e indica um
 `Cluster` para armazenamento por ID do Departamento.
SQL> create table emp_curso
  2     cluster emp_dept_cluster (deptno)
  3     as select * from emp;

-- cria a tabela `dept_curso` com base na tabela `dept` e indica
um `Cluster` para armazenamento por ID do Departamento.
SQL> create table dept_curso
  2     cluster emp_dept_cluster (deptno)
  3     as select * from dept;

-- seleciona todas as colunas da table EMP_CURSO onde o código do
 departamento seja igual a 30.
SQL> select *
  2  from   emp_curso
  3  where  deptno = 30;

Execution Plan
----------------------------------------------------------
Plan hash value: 4111351371

-----------------------------------------------------------------------------------------
| Id  | Operation           | Name                   | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                        |     6|   522 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS CLUSTER| EMP_CURSO             |     6|   522 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN  | IDX_EMP_DEPT_CLUSTER |     1|       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 6.18: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img18code.txt

Para executar a verificação, o Oracle primeiro obtém o `rowid` da linha que se refere ao departamento 30, varrendo o índice do `Cluster` (etapa 2). O Oracle então localiza as linhas em `emp_curso` usando esse `rowid` (etapa 1).

Note que, com base no `CLUSTER` e no `INDEX CLUSTER` criados, no plano de execução é mostrada a opção do otimizador pelo acesso via `CLUSTER SCAN` (`TABLE ACCESS CLUSTER`).

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

NOTA: No exemplo, foram criadas novas tabelas, `emp_curso` e `dept_curso`, contudo não se faz necessária a duplicação de objetos para o emprego desta forma de armazenamento. Fiz desta forma para não mexer na estrutura atual das tabelas na base de testes utilizada para este livro.

No capítulo sobre índices, não abordamos esse tipo de índice, pois não são comuns no dia a dia. Por isso, optei por comentar sobre eles aqui neste capítulo, mas de forma breve, falando um pouco sobre suas características e funcionamento, bem como seu método de acesso pelo otimizador.

6.7 HASH SCANS

Este método poderá ser usado quando as informações da tabela estiverem armazenadas em `HASH CLUSTER`. Um `HASH CLUSTER` é como um `Cluster` indexado, exceto que a chave de índice é substituída por uma função de `hash`. Neste caso, não existe um `CLUSTER INDEX` intermediando. Em um `HASH CLUSTER`, os dados são as próprias entradas de índice. O Oracle usa uma varredura por `hash` para localizar linhas em um `HASH CLUSTER` com base em um valor de `hash`. O Oracle considera uma varredura por `HASH` quando uma consulta acessa uma tabela que possui um `HASH CLUSTER`. Em um `HASH CLUSTER`, todas as linhas com o mesmo valor de `hash` são armazenadas no mesmo bloco de dados.

Para executar uma varredura por `HASH CLUSTER`, o Oracle

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

primeiro obtém o valor de `hash` aplicando uma função `hash` a um valor de chave de `cluster` especificado pela instrução. O Oracle então verifica os blocos de dados que contêm linhas com esse valor de `hash`.

```
SQL> create cluster emp_dept_cluster
  2     (deptno number(2)) size 8192 hashkeys 100;

Cluster created.

SQL> create table emp_curso
  2     cluster emp_dept_cluster (deptno)
  3     as select * from emp;

Table created.

SQL> create table dept_curso
  2     cluster emp_dept_cluster (deptno)
  3     as select * from dept;

Table created.

SQL> select *
  2  from   emp_curso
  3  where  deptno = 30;

Execution Plan
----------------------------------------------------------
Plan hash value: 3281001180

--------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     1 |    87 |     0   (0)|
|*  1 |  TABLE ACCESS HASH| EMP_CURSO |     1 |    87 |            |
--------------------------------------------------------------------
```

Figura 6.19: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img19code.txt

Para executar um `HASH SCAN`, o Oracle primeiro obtém o

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

valor de `hash` aplicando uma função de `hash` ao valor de chave 30 e, em seguida, usa esse valor de `hash` para verificar os blocos de dados e recuperar as linhas (etapa 1).

> **NOTA:** No exemplo, foram criadas novas tabelas, `emp_curso` e `dept_curso`, contudo não se faz necessária a duplicação de objetos para o emprego desta forma de armazenamento. Fiz dessa forma para não mexer na estrutura atual das tabelas na base de testes utilizada para este livro.

6.8 SAMPLE TABLE SCANS

Uma `SAMPLE TABLE SCANS` recupera uma amostra aleatória de dados de uma tabela simples ou de uma instrução SQL complexa, como uma instrução que envolve junções com outras tabelas ou até `views`. O otimizador pode escolher esse tipo de acesso quando, em uma cláusula `from`, estiver incluída a cláusula `sample` ou `sample block`. A cláusula `sample` tanto pode operar em cima das linhas propriamente ditas de uma tabela ou nos blocos de dados. Veja as seguintes opções:

- `sample (sample_percent)`: o Oracle lê uma porcentagem especificada de linhas da tabela com o intuito de executar uma varredura da tabela por amostragem de linhas.
- `sample block (sample_percent)`: o Oracle lê uma porcentagem especificada de blocos de uma determinada tabela para executar uma varredura da tabela por

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

amostragem de blocos de dados.

Com isso, você pode escolher se o Oracle realizará varreduras por linhas ou por blocos de dados. O `sample_percent` especifica a porcentagem da contagem total de linhas ou blocos para que sejam incluídos na amostra. O valor deve estar no intervalo de .000001% a 100%, mas não incluindo o 100%. Esse percentual indica a probabilidade de cada linha, ou cada conjunto de linhas na amostragem por bloco, ser selecionada para a amostra solicitada. Isso não significa que o Oracle recupere exatamente o `sample_percent` das linhas. Veja os exemplos a seguir:

Amostragem por linhas:

```
SQL> select * from employees sample(1);

Execution Plan
----------------------------------------------------------
Plan hash value: 2998680986

-------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |     1 |    69 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS SAMPLE| EMPLOYEES |     1 |    69 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Figura 6.20: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img20code.txt

Amostragem por blocos:

```
SQL> select * from departments sample block(1);
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1904088435

-------------------------------------------------------------------------------
| Id  | Operation           | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |             |     1 |    21 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS SAMPLE| DEPARTMENTS |     1 |    21 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Figura 6.21: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img21code.txt

A amostragem de blocos só é possível durante o `FULL TABLE SCANS` ou em `INDEX FAST FULL SCANS`. Se existir um caminho de execução mais eficiente, então o Oracle não utilizará amostra por blocos. Para garantir a amostragem por blocos para uma tabela ou índice específico, use o `hint` `FULL` ou `INDEX_FFS`.

6.9 ESCOLHENDO OS MÉTODOS DE ACESSO NO MODO RBO

Nos dias de hoje, em grande parte pela evolução das versões mais recentes do Oracle, é incomum ver um banco de dados Oracle operando em modo baseado em Regras. Contudo, no passado, não havia outra forma de funcionamento, então o modo baseado em regras era o *modus operandi* da época para versões mais antigas. De qualquer forma, sabemos que, mesmo que o otimizador esteja trabalhando em modo custo, é necessário que informações estatísticas e de objetos da base de dados estejam sendo geradas e armazenadas para que esse modo opere de forma adequada. Caso contrário, o otimizador ainda pode funcionar em cima de diretrizes que o levem a operar de forma mais próxima ao modo baseado em regras (RBO).

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Como hoje em dia o mais comum é vermos bancos trabalhando em modo baseado em custo, por razões óbvias até, por serem mais robustos e orientados a trabalhar dentro da necessidade de melhores performances, contando com muito mais tecnologia, informações, controles e recursos para melhor operar sobre os dados do banco, não precisamos nos aprofundar muito, apenas conhecer algumas características interessantes.

Quando estamos trabalhando com o otimizador baseado em Regras, ele se utiliza de um rank para a escolha do melhor método de acesso a ser usado para acessar os dados de uma tabela, um índice, uma `view` , um `cluster` ou `hash` . Esta lista indica a classificação do que seria a melhor escolha, começando pela melhor opção no topo deste rank. Dentro desta lista estão os métodos de acesso que vimos até agora, mas utilizados de forma ranqueada, obedecendo ao que foi definido pela Oracle. Este rank é definido utilizando-se como base a ciência heurística, ou seja, por comprovações e experiências empregadas em determinados cenários, e como os métodos poderiam ser mais bem executados levando em consideração suas características. É um trabalho de anos, envolvendo centenas de milhares de testes, na busca de melhorar esses métodos a cada versão do banco de dados, criando cálculos e fórmulas com o intuito de torná-los mais eficientes e principalmente mais rápidos.

No geral, o primeiro item do rank é o que proporcionará o acesso mais rápido, enquanto o último é aquele com o acesso mais lento. No entanto, o otimizador poderá optar por uma opção de acesso diferente mesmo se o cenário estiver apto para uma escolha melhor no rank. Tudo vai depender da forma como escrevemos a instrução SQL e também dos recursos de banco disponíveis no

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

momento. Temos que estar cientes de que nem sempre o otimizador escolherá a melhor alternativa e é aí que nós entramos para melhor auxiliá-lo nessa escolha, sendo por tentar induzi-lo a um método mais adequado através da criação de índices ou pelo uso de `hints` para influenciar o uso de métodos de junção diferentes, por exemplo, ou ainda, pela melhor construção dos nossos programas SQL, dando maior ênfase para este último. No geral, o otimizador analisará todos os métodos possíveis e escolherá um que julgará ser o melhor para a abordagem utilizada, sempre respeitando e tomando como base o que diz a instrução SQL.

Se me perguntarem qual o melhor modo de operação, eu vou responder que é o modo baseado em Custo (CBO). No entanto, dependendo da situação - e não estou falando apenas no contexto técnico mas no contexto geral, no qual, na maioria das vezes, estamos mexendo em performance por conta de uma urgência em resolver um problema (não importando como, naquele determinado momento) - forçar um otimizador a trabalhar em modo baseado em regras pode ser a saída mais rápida. Então, como eu disse, e sempre digo: não analise somente o problema técnico, analise a situação em um âmbito geral. Não adianta chegar à conclusão de que a melhor solução para resolver um problema de performance de um determinado programa é ele ser todo reescrito, levando semanas, quando, no máximo, um cliente precisa que o problema seja resolvido em duas horas. Assim, use tudo o que estiver ao seu alcance, mesmo que não dê naquele exato momento, para implantar uma solução adequada pensando em cenários futuros. Depois que a poeira baixar, você volta ao problema com calma e propõe um melhor cenário para que problemas como esse não aconteçam novamente.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Veja a seguir a tabela referente ao rank utilizado pelo otimizador para escolher um determinado caminho de acesso baseado em Regras.

Rank	Access Path
1	Single row by ROWID
2	Single row by Cluster Join
3	Single Row by Hash Cluster Key with Unique or Primary Key
4	Single Row by Unique or Primary Key
5	Clustered Join
6	Hash Cluster Key
7	Indexed Cluster Key
8	Composite Index
9	Single-Column indexes
10	Bounded Range Search on Indexed Columns
11	Unbounded Range Search on Indexed Columns
12	Sort-Merge Join
13	MAX or MIN of Indexed Columns
14	ORDER BY on Indexed Columns
15	Full Table Scan

O uso de `CLUSTER` e `HASH CLUSTER` não são muito comuns, embora estejam em uma posição favorável no rank em relação a outros métodos de acesso. Devemos ter em mente que a utilização desses dois métodos gera a necessidade de mais espaço em disco do que o uso de índices, por exemplo. Desta forma, analise a possibilidade de criar índices, principalmente quando problemas de performance já existirem no ambiente por conta de estarmos

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

manipulando quantidades de registros na casa dos milhões. De qualquer forma, recomendo que consultem informações mais detalhadas nos manuais da Oracle e também seu DBA para melhores recomendações.

O otimizador baseado em regras é muito elegante por sua simplicidade e muitas vezes faz escolhas de execução mais rápidas do que o CBO. De fato, os produtos Oracle Applications usaram o RBO até 2001, quando o produto Oracle Apps 11i foi introduzido. Foi somente com o lançamento do Oracle8i (8.1.6) que o CBO se tornou mais rápido do que a RBO em todos os casos.

Podemos fazer algumas observações gerais sobre as características do otimizador baseado em regras:

Opta sempre pelo índice: se um índice pode ser usado para acessar uma tabela, o otimizador escolhe o índice. Os índices são sempre preferidos em vez de `FULL TABLE SCAN` ou `SORT MERGE JOIN` (`SORT MERGE JOIN` não requer um índice).

Sempre começa pela tabela de condução (Driving Table): a última tabela na cláusula `from` será a tabela de condução - para o modo RBO, esta tabela deve ser a que contiver a menor quantidade de linhas. O modo RBO usa esta tabela de condução como a primeira tabela ao executar operações de `NESTED LOOPS` .

Full Table Scan como um último recurso: o modo RBO não atua em cima de queries paralelas (`parallel query`) e em leitura de multiblocos (`Multiblock Reads`), e também não considera o tamanho da tabela. Dessa forma, esse modo opta por não utilizar o método de acesso `FULL TABLE SCAN` e só optará por ele quando nenhum índice for encontrado.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Qualquer índice é índice: o modo RBO por vezes pode escolher um índice não muito adequado para atender a uma instrução SQL. Isso ocorre porque este modo não tem acesso às estatísticas que mostram a seletividade de colunas indexadas, por exemplo.

O simples às vezes é melhor: antes do Oracle8i, o modo RBO muitas vezes forneceu um plano de execução melhor para alguns bancos de dados.

6.10 ESCOLHENDO OS MÉTODOS DE ACESSO NO MODO CBO

Quando estamos trabalhando com o otimizador baseado em custo, ele utiliza a mesma tabela de rank utilizada pelo modo RBO. Contudo, operando em modo custo, o otimizador leva em consideração outras características como a seletividade do dado, número de blocos necessários para se fazer um `FULL TABLE SCAN` (varredura completa dos dados da tabela) *versus* o número de blocos obtidos em uma operação de leitura (geralmente blocos de 32 bits), número de linhas da tabela e outras informações estatisticamente geradas, conforme visto em capítulos anteriores. O CBO escolhe um método de acesso com base nos seguintes fatores:

- Verifica os métodos de acesso disponíveis para a instrução SQL.
- Verifica o custo estimado de execução da instrução, usando cada método de acesso ou combinação de métodos.

Para escolher um método de acesso, o otimizador determina

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

primeiro quais estão disponíveis, examinando as condições da cláusula `where` da instrução e da cláusula `from` para a cláusula `sample` ou `sample block` . O otimizador então gera um conjunto de planos de execução possíveis usando métodos de acesso disponíveis e estima o custo de cada plano, usando as estatísticas do índice, colunas e tabelas acessíveis à instrução. Finalmente, o otimizador escolhe o plano de execução com o custo estimado mais baixo.

Também já sabemos que algumas situações podem influenciar o otimizador. Uma delas é através de `hints` . A escolha do otimizador entre métodos de acesso disponíveis pode ser influenciada por dicas (`hints`), exceto quando a cláusula `from` da instrução contiver `sample` ou `sample block` .

Outro fator que pode influenciar o otimizador na escolha de determinado método é a existência de estatísticas desatualizadas. Por exemplo, se uma tabela não foi analisada desde que foi criada, e se tiver menos `db_file_multiblock_read_count` do que blocos sob a `High Water Mark` , então o otimizador pode achar que a tabela é pequena e poderá usar uma `FULL TABLE SCAN` . Revise as colunas `last_analyzed` e `blocks` pela `view` `all_tables` para ver o que as estatísticas refletem.

NOTA: `db_file_multiblock_read_count` especifica o número máximo de blocos lidos em uma operação de I/O durante uma varredura sequencial.

Agora vamos ver alguns exemplos de cenários e suas escolhas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

pelo otimizador:

Cenário 1: neste cenário, a instrução SQL usa uma condição de igualdade na cláusula `where` para selecionar todos os empregados com o nome igual a `'JACKSON'`.

Analisando o exemplo a seguir, se a coluna `last_name` for uma chave exclusiva ou primária, o otimizador determinará que há apenas um empregado chamado `'JACKSON'` e a instrução SQL retorna apenas uma linha. Nesse caso, a instrução será muito seletiva e é bem provável que o otimizador acessará a tabela usando acesso via `UNIQUE SCAN` no índice, que é característico para uma chave exclusiva (`INDEX UNIQUE`) ou uma chave primária (`Primary Key`). Vejamos o resultado:

```
SQL> select *
  2  from   employees
  3  where  last_name = 'JACKSON';

Execution Plan
----------------------------------------------------------
Plan hash value: 2077747057

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |             |     1 |    69 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |     1 |    69 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP_NAME_IX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Figura 6.22: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img22code.txt

Aqui aconteceu algo interessante. Para essa tabela, existe um índice composto para as colunas `last_name` e `first_name`, nesta ordem de colunas. Este índice apareceu no plano de execução (passo 2), no qual o otimizador acessou o índice através de um

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`INDEX RANGE SCAN`. Mas vamos testar com a coluna `email`. Vamos selecionar todos os atributos de `employees` em que o e-mail seja igual a `'DLORENTZ'`.

```
SQL> select *
  2  from   employees
  3  where  email = 'DLORENTZ';

Execution Plan
----------------------------------------------------------
Plan hash value: 2693862462

-----------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |     1 |    69 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     1 |    69 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMAIL_UK |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 6.23: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img23code.txt

Agora o plano de execução surtiu o efeito que esperávamos no exemplo anterior. No entanto, para o predicado que utilizamos nesse exemplo, existe um índice único criado, e o otimizador realizou um acesso através dele.

Cenário 2: considerando o mesmo exemplo do cenário 1, se a coluna `last_name` não for uma chave exclusiva ou primária - se constatarmos, veremos que não é -, o otimizador poderá usar as seguintes estatísticas para estimar a seletividade da consulta:

- `user_tab_columns.num_distinct`, que é o número de valores distintos para cada coluna na tabela.
- `user_tables.num_rows`, que é o número de linhas da tabela.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Ao dividir o número de linhas da tabela empregados (`employees`) pelo número de valores distintos na coluna `last_name` , o otimizador estima a porcentagem de empregados que têm o mesmo nome. Supondo que os valores de `last_name` são distribuídos uniformemente, o otimizador usa esta porcentagem como seletividade estimada para a instrução SQL. Vamos dar uma olhada nos dados referentes à seletividade dessa tabela e coluna:

```
SQL> select a.num_distinct
  2        ,b.num_rows
  3        ,round((b.num_rows / a.num_distinct),2) media_valores_
distintos
  4        ,concat(round(100-(100 / b.num_rows * a.num_distinct),
2),'%') pc_nomes_iguais
  5        ,concat(round(((a.num_distinct / b.num_rows) * 100),2)
,'%') pc_seletividade
  6  from   user_tab_columns a
  7        ,user_tables      b
  8  where  a.table_name  = b.table_name
  9  and    b.table_name  = 'EMPLOYEES'
 10  and    a.column_name = 'LAST_NAME';

NUM_DISTINCT   NUM_ROWS MEDIA_VALORES_DISTINTOS   PC_NOMES_IGUAIS
     PC_SELETIVIDADE
------------ ---------- ------------------------- ---------------
---- ----------------
         102        107 1.05                      4.67%
     95.33%
```

Quanto maior a seletividade, melhor. Veja que o percentual de seletividade deste índice (`pc_seletividade`) chega próximo a 100% (95.33%). Este deve ter sido um bom motivo para que o otimizador optasse pelo método de acesso `INDEX RANGE SCAN` . Observe também que nesta tabela temos apenas 4.67% de nomes iguais que dá uma média de 1.05, chegando muito perto de 1 para 1.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Cenário 3: a instrução a seguir seleciona todos os empregados com o ID menor que 7500.

```
SQL> select *
  2  from   employees
  3  where  employee_id < 7500;

 Execution Plan
 ----------------------------------------------------------------------------
 | Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
 ----------------------------------------------------------------------------
 |   0 | SELECT STATEMENT  |           |   107 |  7383 |     2   (0)| 00:00:01 |
 |*  1 |  TABLE ACCESS FULL| EMPLOYEES |   107 |  7383 |     2   (0)| 00:00:01 |
 ----------------------------------------------------------------------------
```

Figura 6.24: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img24code.txt

```
SQL> select high_value
  2        ,low_value
  3  from   user_tab_col_statistics
  4  where  table_name  = 'EMPLOYEES'
  5  and    column_name = 'EMPLOYEE_ID';

HIGH_VALUE                                          LOW_VALUE
--------------------------------------------------- -----------------
--------------
C20307                                              C202
```

Para estimar a seletividade da instrução SQL, o otimizador usa o valor limite de 7500 da condição da cláusula `where` e os valores das estatísticas `high_value` e `low_value` para a coluna `employee_id`, se disponível. Essas estatísticas podem ser localizadas nas `views` `user_tab_col_statistics` ou `user_tab_columns`. O otimizador assume que os valores de `employee_id` são distribuídos uniformemente com base na faixa entre o valor mais baixo e o valor mais alto. Neste caso, ele então determina qual porcentagem desse intervalo é menor que o valor

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

7500, utilizado no predicado, e usa esse valor como seletividade estimada para a instrução.

Nesse exemplo, o otimizador escolheu realizar um acesso `FULL TABLE SCAN` na tabela `employees`, pois mesmo a coluna `employee_id` tendo uma chave exclusiva associada a ela, por ser uma chave primária, todos os registros de `employees` são retornados com base no predicado. E como todas as colunas da tabela são selecionadas, não faz sentido utilizar o índice e ainda a varredura completa na tabela.

Cenário 4: este cenário mostra o uso de uma variável `bind` em vez de um valor literal, propriamente dito, como valor que tem o objetivo de limitar a quantidade de registros em uma condição `where`.

```
SQL> variable e1 number -- Define uma variável no SQL*Plus.

SQL> select *
  2  from   employees
  3  where  employee_id < :e1;

Execution Plan
----------------------------------------------------------
Plan hash value: 603312277

---------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     5 |   345 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     5 |   345 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP_EMP_ID_PK |     2 |       |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------
```

Figura 6.25: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img25code.txt

O otimizador não conhece qual o valor da variável `bind`

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`'e1'` . Este valor pode ser diferente cada vez que a instrução SQL é executada. Por esse motivo, o otimizador não pode usar como base os mesmos passos que vimos no exemplo anterior para determinar a seletividade desta instrução, ou melhor, da tabela `employees` . Nesse caso, ele tentará adivinhar heuristicamente um valor menor para a seletividade, usando um valor interno padrão. O otimizador faz essa suposição, digamos assim, sempre que uma variável `bind` é usada como um valor de limite em uma condição com um dos seguintes operadores: menor que (<), maior que (>), menor igual a (<=), ou maior igual a (>=). O otimizador atua desta forma porque ele só substituirá as variáveis quando o comando for executado, ou seja, no processo de geração e escolha dos planos de execução, o otimizador ainda estará trabalhando somente com as variáveis `binds` , não com os valores propriamente ditos.

A forma de tratar do otimizador, quando estamos falando de variáveis `bind` , pode fazer com que ele escolha planos de execução diferentes para a mesma instrução SQL para cenários utilizando variáveis `bind` e cenários utilizando valores constantes.

Por exemplo, podemos ver planos de execução diferentes sendo escolhidos pelo otimizador para uma instrução SQL contendo uma variável `bind` em um programa pré-compilador do Oracle, por exemplo, `Oracle Forms` ou `Reports` , e a mesma instrução SQL com um valor constante no `SQL*Plus` .

Isso não é incomum de acontecer pois, como dito, o Oracle não conhece o valor da variável `bind` que está sendo utilizada na instrução. Este tipo de cenário pode se apresentar ainda mais diferente caso o uso de variáveis `bind` reflitam restrições por faixa de dados, como o uso de `between` ou dos operadores menor que

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

(<) e maior que (>).

Cenário 5: como mencionado no cenário 4, devemos ficar atentos à análise de performance de alguma instrução SQL quando nela existirem variáveis do tipo `bind`. O plano de execução pode se mostrar bastante diferente ao analisarmos um plano gerado a partir de uma instrução usando este tipo de variável, sem a substituição dessas variáveis, e uma execução desta instrução, propriamente dita, onde as variáveis seriam substituídas pelos seus respectivos valores. A instrução a seguir, usa duas variáveis `bind` como valores de intervalo na condição utilizando o operador `between`:

```
SQL> variable low_e number
SQL> variable high_e number
SQL>
SQL> select *
  2  from   employees
  3  where  employee_id between :low_e and :high_e;

Execution Plan
----------------------------------------------------------
Plan hash value: 407010018

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     2 |   138 |     2   (0)| 00:00:01 |
|*  1 |  FILTER                     |               |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     2 |   138 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN         | EMP_EMP_ID_PK|     2 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(TO_NUMBER(:LOW_E)<=TO_NUMBER(:HIGH_E))
   3 - access("EMPLOYEE_ID">=TO_NUMBER(:LOW_E) AND "EMPLOYEE_ID"<=TO_NUMBER(:HIGH_E))
```

Figura 6.26: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img26code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Para começar, o otimizador transforma a condição `between` em duas condições, como pode ser visto no plano de execução, passo 3, e destacado a seguir. Mais informações sobre este tipo de transformação, você pode ver no tópico *Performance através da Sintaxe SQL*, no capítulo 2 do livro.

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(TO_NUMBER(:LOW_E)<=TO_NUMBER(:HIGH_E))
   3 - access("EMPLOYEE_ID">=TO_NUMBER(:LOW_E) AND "EMPLOYEE_ID"<
=TO_NUMBER(:HIGH_E))
```

O otimizador então estima heuristicamente uma pequena seletividade (um valor interno padrão) para as colunas indexadas a fim de favorecer o uso do índice. Contudo, em uma execução real, esses valores tanto podem ser bem distantes como bem próximos. Com isso, dependendo da quantidade de dados a serem retornados ou blocos de dados a serem lidos, o otimizador poderá optar por um ou por outro plano de execução.

Geralmente, uma dica interessante é investigar as execuções anteriores em busca de parâmetros mais assertivos, aqueles que são mais utilizados durante o dia a dia, por exemplo. Procure investigar com as pessoas que executam esses programas para saber que parâmetros elas costumam utilizar como entrada de dados. Depois disso, analise esses planos de execução informando esses valores, valores reais, em substituição às variáveis `binds`. Assim, é possível chegar mais próximo do que seria uma situação real de execução. Às vezes você vai perceber que, dependendo dos valores informados, principalmente quando estamos trabalhando com faixa de valores, o tempo de resposta pode variar consideravelmente em consequência da escolha do plano.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Cenário 6: neste cenário, temos uma instrução SQL que usa o operador `between` para selecionar todos os empregados com IDs entre 7500 e 7800.

```
SQL> select *
  2  from   employees
  3  where  employee_id between 7500 and 7800;

Execution Plan
----------------------------------------------------------
Plan hash value: 603312277

-----------------------------------------------------------------------------------------
| Id  | Operation                   | Name          |Rows  |  Bytes| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |   1  |    69 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |   1  |    69 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP_EMP_ID_PK |   1  |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID">=7500 AND "EMPLOYEE_ID"<=7800)
```

Figura 6.27: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap06img27code.txt

Como já vimos no cenário 5, para determinar a seletividade desta instrução, o otimizador reescreve a condição da cláusula `where`, transformando-a em duas condições, substituindo o operador `between` (passo 2):

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID">=7500 AND "EMPLOYEE_ID"<=7800)
```

Depois, o otimizador calcula a seletividade individual de cada condição, usando os mesmos métodos descritos nos cenários 3 e 4, mas não esquecendo de que, aqui, ele conhece os valores. O otimizador então usa essas seletividades (representadas aqui por S1

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

e S2) e aplica a função `abs` (valor absoluto) para estimar a seletividade da condição `between` (aqui chamada S). Veja a seguir como seria o cálculo:

```
S = ABS(S1 + S2 - 1)
```

Através desses conceitos, foi possível conhecer várias técnicas que são utilizadas pelo otimizador para encontrar o melhor plano de execução. Vimos também que vários aspectos podem influenciar esta escolha: a forma como escrevemos nossas instruções SQL, os recursos físicos disponíveis no momento e as estatísticas atualizadas ou não dos objetos que estamos acessando.

O interessante de entender como isso tudo funciona é que você começa a ter conhecimento dessas informações para poder interagir com o otimizador. Assim, você influencia a forma como ele trabalha e busca melhorias de performance ou a solução de problemas dessa natureza de forma mais eficaz em vez de ficar limitado à criação de índices.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 7

TABELA CONTROLE (DRIVING TABLES)

Conforme expliquei no capítulo anterior, uma das decisões que o otimizador precisa tomar quando começa a testar planos de execução é escolher qual tabela, entre todas as tabelas da instrução SQL, será a `Driving Table`, ou, em uma tradução livre, *Tabela de Condução*. Em muitas literaturas e materiais, você vai achar diversos nomes para a `Driving Table`, principalmente se estiver procurando materiais em português. Tabela de Entrada, Tabela Controle, Tabela de Condução, Tabela Externa, estes são alguns dos nomes que você vai encontrar por aí. Aqui vou chamá-la de tabela de condução, pois acredito ser mais pertinente com a função, pois ela conduzirá o início da busca pelos dados. Como já mencionei, uma tabela será a de condução, enquanto as demais tabelas de uma instrução SQL serão as de junção, também chamadas de `Tables Join` ou `Inner Tables`. Também comentei que este processo é de fundamental importância, pois entender os conceitos por trás das `Driving Tables` e `Tables Join` fará com que você comece a escrever instruções SQL muito mais eficientes e mais performáticas e a entender o porquê das escolhas do otimizador.

Ah! Antes de mais nada, essa questão de como são chamadas as

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

tabelas de condução e tabelas de junção pode dar um nó na cabeça no início, pois muitos autores falam se referindo ao seu funcionamento, principalmente, quando estamos falando sobre o método `NESTED LOOPS`. Você verá que os conceitos sobre `Driving Tables` são bem relevantes nos planos de execução quando os métodos `NESTED LOOPS` são utilizados. Você verá posteriormente que temos o conceito de tabela de condução nos outros métodos, contudo ele se torna mais relevante quando `loops` estão sendo utilizados no plano de execução. Pois bem, você pode se deparar com literaturas falando que a tabela de condução será a tabela externa. Como eu disse, isso pode trazer confusão na hora de interpretar um plano de execução, pois, ao mesmo tempo em que você pode chamá-la de tabela externa, você acaba verificando que ela é, na verdade, a mais interna dentro do plano de execução. No entanto, tal afirmação não é errada, apenas precisamos entender que, quando dizemos que ela é a tabela externa, estamos dizendo que ela será a tabela do primeiro `loop`, o `loop` mais externo. Observe que se estamos lendo um plano de execução de dentro para fora precisamos montar estes `loops` de fora para dentro. Mas por ora é isso... Reflita! Mais para a frente voltaremos a falar nisso.

Continuando, depois que o otimizador identifica qual será a tabela de condução, ela será utilizada pelo otimizador como ponto de partida para o processamento dos dados. Então, como já vimos, no plano de execução ela estará localizada no nível mais interno, no caso de um método `NESTED LOOPS`, por exemplo. Veja a instrução SQL a seguir:

```
select * from emp, dept where emp.deptno = dept.deptno
```

Nessa instrução, a tabela de condução poderia ser a tabela

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`dept` (departamentos). O otimizador poderia buscar linhas da tabela `dept` através de uma varredura completa na tabela e, em seguida, comparar com as linhas da tabela `emp` (empregados), trazendo somente as correspondentes. Nesse caso, `dept` seria a tabela de condução, ou seja, a tabela inicial escolhida pelo otimizador para iniciar a busca pelos dados que fariam parte do resultado.

A escolha de uma tabela de condução pelo otimizador é feita com base em diversos fatores. Por exemplo, se nesta mesma instrução houvesse um índice na coluna `deptno` da tabela `dept`, mas não na coluna `deptno` da tabela `emp`, haveria grandes chances de o otimizador escolher a tabela `emp` como tabela de condução. A busca seria pelas linhas da tabela `emp` e, em seguida, usando o índice de `dept` (coluna `deptno`), encontraria suas correspondentes na tabela `dept`.

Não podemos esquecer também que é neste primeiro processamento que o otimizador eliminará as linhas que não satisfizerem as condições da cláusula `where`, caso existam. Uma vez recuperadas tais linhas, somente aquelas que satisfizerem as condições existente na cláusula `where` serão usadas na eliminação das demais linhas que serão processadas nas tabelas subsequentes.

Vale lembrar que os predicados de filtro, aqueles que são inseridos na cláusula `where` e que servem para filtrar as linhas, são aplicados nas tabelas antes que suas junções aconteçam. Então, considere essa informação antes de posicioná-las na cláusula `from`.

Mas você deve estar se perguntando quais os critérios utilizados pelo otimizador para a escolha desta tabela. Pois bem,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

aspectos relacionados ao tamanho da tabela, cardinalidade, índices, entre outras características, podem afetar na escolha de uma tabela de condução. Um fator que também pode influenciar é o método de junção. No caso da junção do tipo `NESTED LOOPS`, por exemplo, o otimizador tende a escolher como tabela de condução aquela que possua menos dados, sendo pela quantidade total dos registros da tabela ou pela eliminação de linhas após a aplicação dos predicados que por ventura existam na instrução. Embora muitas literaturas tomem como regra que o otimizador, no caso do método `NESTED LOOPS`, utilize sempre a tabela como menor número de registros, isso pode não ocorrer em alguns casos. Outro fator que pode determinar a forma como o otimizador escolhe tais tabelas é o modo como o banco de dados está configurado.

O conceito de tabela de condução é utilizado tanto no modo RBO, quanto no modo CBO. Quando a tabela de condução não puder ser identificada pelo otimizador, a ordem de processamento se dará pela localização das tabelas na cláusula `from`, ou seja, na disposição em que elas estarão nesta cláusula. A ordem partirá da última tabela informada na clausula `from`, portanto, da leitura da direita para a esquerda, indo em direção à primeira tabela. Então, como melhor prática, devemos sempre colocar a tabela de condução no final da cláusula `from`, pois, assim, garantimos que, mesmo que o otimizador não consiga selecioná-la, mediante os diversos planos de execução testados por ele, a tabela de condução seja a melhor candidata. Em seguida, posicione as demais tabelas deixando sempre as tabelas com maiores quantidades de registros mais próximas do início da cláusula `from`.

Uma forma de explicitamente influenciar na escolha do otimizador é através da utilização de `hints`, como já foi falado.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Para efeito de conhecimento, há um `hint` chamado `rule` no otimizador baseado em custo, para que essa sequência seja respeitada e tenha o mesmo efeito como se o banco estivesse operando no modo RBO. Veja o exemplo a seguir:

```
select pda.*
from   po_distributions_all  pda
      ,po_line_locations_all plla
      ,po_lines_all          pla
      ,po_headers_all        pha
where  pha.po_header_id        = 345
and    pha.po_header_id        = pla.po_header_id
and    pla.po_line_id          = plla.po_line_id
and    pla.po_line_location_id = pda.po_line_location_id
```

Nesse exemplo, a tabela `po_headers_all` será a tabela de condução, enquanto a tabela `po_distributions_all` será a tabela com maior quantidade de linhas recuperadas e, portanto, está localizada mais próxima da cláusula `from` .

Caso você ache que o otimizador não está conseguindo posicionar a melhor candidata à tabela de condução, você poderá utilizar um `hint` para forçar a ordem com base na disposição que você colocar as tabelas na cláusula `from` . No entanto, na maioria das vezes, ele consegue detectar a melhor tabela para essa função.

Já utilizando o modo CBO, baseado em custo, a ordem das tabelas é irrelevante (a menos que você use um `hint` próprio para isso). No caso do modo CBO, estatísticas e dicionário de dados são informações utilizadas para determinar qual tabela é a melhor candidata para ser usada como tabela de condução.

Agora vamos ver um exemplo que demonstra a escolha pelo otimizador e como isso pode ser influenciado. Vamos executar a instrução SQL a seguir e visualizar o plano de execução em

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

seguida.

```
select * from emp, dept where emp.deptno = dept.deptno
```

Para entendermos os cenários que serão mostrados, vou passar algumas informações. Na base que utilizamos para esses testes, existem criadas as tabelas `emp` (empregados) e `dept` (departamentos). Para essas tabelas não existem colunas indexadas neste momento. Elas possuem poucos registros e são as tabelas que utilizamos em praticamente todos os exemplos deste livro. Nas primeiras páginas, você encontra o endereço de onde baixar os scripts para a criação desses objetos. Você pode utilizar uma base teste para executá-los e tentar simular os cenários que faremos agora. Ainda não falamos sobre como visualizar o plano de execução de uma instrução SQL, contudo, para explanar os conceitos que acabamos de ver, vou demonstrar como podemos visualizar um plano de execução de forma rápida. Para isso, vamos utilizar a ferramenta `SQL*Plus` da própria Oracle. Mais à frente, no capítulo 9, veremos em detalhes os Planos de Execução. Vamos utilizar a ferramenta `SQL*Plus` , que é a ferramenta padrão que serve de interface de comunicação com o banco de dados Oracle e que já é instalada, juntamente, com o banco.

Apenas abrindo parênteses, ainda sobre as ferramentas, eu gostaria de salientar que aprender a trabalhar com o SQL*Plus nos permitirá não ficarmos amarrados a uma ferramenta gráfica ou a ferramentas de terceiros, por exemplo. Outra vantagem é que, executando cada passo, você também entenderá melhor o funcionamento e os conceitos por trás desses recursos. Todavia, também gosto de frisar e de recomendar que, após a aprendizagem destes conceitos, você pode se utilizar de ferramentas gráficas, sim, pois elas podem lhe trazer mais produtividade e resultados

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

visualmente melhores, principalmente quando estivermos falando de planos de execução mais extensos.

Vale ressaltar que, por ora, vamos apenas visualizar o plano de execução sem falar das suas características de forma detalhada, pois o foco aqui é falar sobre as tabelas de condução, neste momento.

Voltando aos exemplos, vamos dar uma olhada na execução e no plano de execução a seguir.

```
SQL> select * from emp, dept where emp.deptno = dept.deptno;

EMPNO ENAME   JOB      MGR HIREDATE      SAL    COMM   DEPTNO  DE
PTNO DNAME           LOC
----- ------- -------- ---- ---------  ----- ------- -------- ---
---- ---------- --------
 7369 SMITH   CLERK    7902 17-DEC-80    800               20
  20 RESEARCH     DALLAS
 7499 ALLEN   SALESMAN 7698 20-FEB-81   1600     300       30
  30 SALES        CHICAGO
 7521 WARD    SALESMAN 7698 22-FEB-81   1250     500       30
  30 SALES        CHICAGO
 7566 JONES   MANAGER  7839 02-APR-81   2975               20
  20 RESEARCH     DALLAS
 7654 MARTIN  SALESMAN 7698 28-SEP-81   1250    1400       30
  30 SALES        CHICAGO
 7698 BLAKE   MANAGER  7839 01-MAY-81   2850               30
  30 SALES        CHICAGO
 7782 CLARK   MANAGER  7839 09-JUN-81   2450               10
  10 ACCOUNTING   NEW YORK
 7788 SCOTT   ANALYST  7566 09-DEC-82   3000               20
  20 RESEARCH     DALLAS
 7839 KING    PRESIDENT     17-NOV-81   5000               10
  10 ACCOUNTING   NEW YORK
 7844 TURNER  SALESMAN 7698 08-SEP-81   1500       0       30
  30 SALES        CHICAGO
 7876 ADAMS   CLERK    7788 12-JAN-83   1100               20
  20 RESEARCH     DALLAS
 7900 JAMES   CLERK    7698 03-DEC-81    950               30
  30 SALES        CHICAGO
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
 7902 FORD    ANALYST  7566 03-DEC-81   3000                20
  20 RESEARCH     DALLAS
 7934 MILLER  CLERK    7782 23-JAN-82   1300                10
  10 ACCOUNTING   NEW YORK
```

Pois bem, vemos que a instrução foi executada e que alguns registros foram retornados. Agora para vermos o plano de execução desta instrução, vamos utilizar o utilitário `autotrace`. Este utilitário mostra o plano de execução escolhido pelo otimizador para a execução da instrução SQL. Esse comando deve ser habilitado na sessão do SQL*Plus. Para habilitá-lo, utilizamos o comando `set`, como mostrado a seguir:

```
SQL> set autotrace trace explain
```

O utilitário `autotrace` pode ser configurado de algumas maneiras. Em geral, utilizamos o utilitário apenas o habilitando através do comando `set autotrace on`. Aqui, para nosso exemplo, utilizei as opções `trace e explain` para que ele apenas mostre o plano de execução. Desta forma, outras informações estatísticas ou até mesmo o resultado da instrução não serão mostrados, apenas o plano de execução. Mais adiante, teremos um capítulo falando sobre os utilitários que auxiliam no monitoramento e ajuste da performance, e então falarei mais detalhadamente sobre cada um deles. Após termos configurado o utilitário `autotrace` na sessão, podemos executar nossa instrução SQL. Veja o plano de execução mostrado na tela, conforme pode ser visto a seguir:

```
SQL> select * from emp, dept where emp.deptno = dept.deptno;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 615168685

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   812 |     5  (20)| 00:00:01 |
|*  1 |  HASH JOIN         |      |    14 |   812 |     5  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPT |     4 |    80 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP  |    14 |   532 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

Figura 7.1: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap07img01code.txt

Analisando esse plano de execução escolhido pelo otimizador para executar essa instrução SQL, podemos observar as seguintes características. Primeiro, o método de execução que o otimizador escolheu foi o método `HASH JOIN`. Ele deve ter escolhido esse método por não termos, por exemplo, índices criados para essas tabelas. Esse também foi o motivo pelo qual foram realizadas varreduras completas (aqui demonstrado como `TABLE ACCESS FULL`) nas duas tabelas para que fosse possível montar o resultado. Uma das características do `HASH JOIN` é utilizar como tabela de condução a tabela com menos registros, o que justificaria a tabela `dept` aparecer primeiro na leitura, conforme pode ser visto no plano. Através da leitura das duas tabelas, podemos ver que a tabela `dept` possui 4 registros e que a tabela `emp` possui 14 registros.

Agora vamos para a parte legal, vamos testar o otimizador de

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

consultas do Oracle para ver se ele escolhe caminhos diferentes para essa mesma execução. Vimos anteriormente que um dos meios de influenciarmos o otimizador é através da criação de índices ou da utilização de `hints`, por exemplo. Desta forma, vamos tentar influenciá-lo, criando um índice na tabela `dept`. Você se lembra de quando falei que, se existisse um índice criado na coluna `deptno` da tabela `dept` e não existisse um índice criado para a coluna `deptno` da tabela `emp`, o otimizador, provavelmente, escolheria a tabela `emp` como sendo a tabela de condução? Vamos tentar simular este cenário? Primeiro vamos criar um índice para a coluna `deptno` da tabela `dept`. Vamos fazer isso criando uma chave primária para esta tabela, indicando a coluna `deptno` como sendo a chave. Sabemos que, quando criamos uma chave primária, o Oracle cria automaticamente um índice para o mesmo conjunto de colunas. Nesse caso, ele vai criar um índice único para a coluna `deptno`, além de torná-la a chave primária da tabela `dept`. Para isso, o seguinte comando pode ser executado:

```
alter table dept add constraint deptno_pk primary key (deptno);
```

Observe que nomeamos como `deptno_pk` a chave primária. Agora vamos gerar um novo plano de execução. Para isso, basta executar, novamente, a nossa instrução SQL. O resultado é mostrado em seguida:

```
SQL> select emp.ename, dept.dname from emp, dept where emp.deptno
 = dept.deptno;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1121137472

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |           |    14 |   308 |     3   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |           |    14 |   308 |     3   (0)| 00:00:01 |
|   2 |   TABLE ACCESS (FULL)         | EMP       |    14 |   126 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS BY INDEX ROWID | DEPT      |     1 |    13 |     1   (0)| 00:00:01 |
|*  4 |    INDEX (UNIQUE SCAN)        | DEPTNO_PK |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

Figura 7.2: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap07img02code.txt

Observe como o plano de execução mudou, ou seja, como o otimizador foi influenciado pela mudança de cenário. Em primeiro lugar, o método de junção utilizado pelo otimizador foi o `NESTED LOOPS` , em vez do `HASH JOIN` , utilizado anteriormente. Isso deve ter ocorrido, pois vimos que uma das condições para que o otimizador escolha esse método é que existam índices em uma ou em mais colunas que fazem a junção entre as tabelas.

Outra condição é que as colunas que fazem parte desta junção, além de serem indexadas (pelo menos uma delas), possuam o mesmo nome. E, como era de se esperar, a tabela escolhida como tabela controle foi a tabela `emp` . Como nela não há índices, o otimizador sabia que teria que ler todos os seus registros, através de uma varredura completa (`FULL`). Isso fez com que ele percebesse que valia a pena ler a tabela `emp` primeiro e comparar cada registro encontrado com os demais registros da tabela `dept` usando o índice `deptno_pk` , que foi criado ao criarmos a chave

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

primária. Dessa forma, ele estaria economizando, inclusive, tempo de processamento, pois, como a coluna `deptno` da tabela `dept` está indexada, ainda mais por um índice único, ele não precisa ler todos os registros desta tabela. Quando ele acha o valor correspondente ou percebe que o valor a ser encontrado é maior que o valor corrente lido, ele já pode parar a leitura e ir para o próximo registro da tabela `emp` .

Esse é um típico caso onde o otimizador fugiu à regra geral, digamos assim, que diz que a tabela de condução escolhida pelo otimizador deverá ser aquela que possui menos registros. Ao contrário dessa regra, o otimizador escolheu a tabela `emp` como a tabela de condução, pois ele sabia que deveria ler todos os registros dela, pois não há predicados que limitem as linhas desta tabela. Já a tabela `dept` , como também não possui predicados limitando seus dados, mas, por outro lado, possui a coluna de relacionamento indexada, se torna uma forte candidata à tabela de junção. O índice dela, como já mencionado, permite que nem todos os registros precisem ser lidos para cada leitura da tabela `emp` no momento da junção, pois sabemos que índices são ordenados e, portanto, ao encontrar seu par na junção, o otimizador não precisará continuar sua busca. Como o índice se trata de um índice único, pois é derivado de uma chave primária, ele se torna ainda mais forte.

Agora, mais uma vez, vamos tentar influenciar o otimizador. Vamos criar um novo índice, agora para a coluna `deptno` da tabela `emp` . Contudo, vamos excluir o índice que existe na coluna `deptno` da tabela `dept` . Depois disso, vamos ver como o otimizador vai montar o plano de execução para essa nossa instrução SQL. Note que, até o momento, não modificamos a instrução. Ela permanece a mesma. Para a criação de um novo

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

índice e a exclusão do índice existente, podemos executar os seguintes comandos:

```
alter table dept drop constraint deptno_pk;

create index emp_idx_1 on emp(deptno);
```

Agora, vamos visualizar a geração do novo plano:

```
SQL> select emp.ename, dept.dname from emp, dept where emp.deptno
 = dept.deptno;

Execution Plan
----------------------------------------------------------
Plan hash value: 1565351509

-----------------------------------------------------------------------------------------
| Id  | Operation                      | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |           |    14 |   308 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS (BY INDEX ROWID) | EMP       |     4 |    36 |     1   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                 |           |    14 |   308 |     3   (0)| 00:00:01 |
|   3 |    TABLE ACCESS (FULL)         | DEPT      |     4 |    52 |     2   (0)| 00:00:01 |
|*  4 |    INDEX (RANGE SCAN)          | EMP_IDX_1 |     5 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

Figura 7.3: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap07img03code.txt

O otimizador continua utilizando o mesmo método de junção, o `NESTED LOOPS`, como no exemplo anterior. Contudo, podemos observar agora que a tabela controle mudou e não é mais a tabela `emp`, e sim, a tabela `dept`. O novo índice, o `emp_idx_1` que criamos, foi o meio utilizado pelo otimizador para acessar os dados da tabela `emp`. Quanto à tabela `dept`, esta foi lida por completo.

Esse plano é bem interessante, pois podemos observar que o otimizador primeiramente relaciona a tabela `dept` não com a

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

tabela `emp` , mas sim com seu índice. Como ele sabia que teria que ler todos os dados de `dept` , colocou esta tabela em primeiro, sendo a tabela de condução. Ao relacionar com o índice, e isso foi possível pois ele se refere à coluna que participa da junção (`emp.deptno`), ele já descarta as linhas que não satisfazem a relação. Fazendo isso, ele evita ler toda a tabela `emp` , o que poderia ocasionar mais custo. O índice em questão é composto por apenas uma coluna e, por sorte, talvez ela já esteja em memória. Do contrário, ele poderia ter que realizar mais leituras em disco, por exemplo, para montar a relação em `emp` e `dept` , caso tivesse que ler a tabela `emp` por completo. Depois de feita essa relação, ele já terá os `rowids` pertencentes à tabela `emp` que satisfazem a condição com `dept` . Através destes `rowids` adquiridos dos índices, ele pode acessar diretamente as linhas da tabela `emp` sem precisar ler todas as linhas da tabela. Nesse caso, o otimizador seguiu a regra da menor tabela sendo nomeada como tabela de condução.

Embora, em nosso primeiro exemplo, o otimizador tenha escolhido como tabela de condução uma tabela com mais registros, fugindo à regra geral, é muito comum a escolha por uma tabela com menos registros. Observe que nos exemplos que utilizamos não há predicados nas tabelas e quase todos os registros, senão todos, satisfazem a condição de relacionamento.

O que você deve estar se perguntando é por que é importante saber qual é a tabela de condução se esperamos que o otimizador determine da melhor forma qual será essa tabela, e vimos nos exemplos mostrados que ele certamente vai tentar tomar a melhor decisão.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nem sempre o otimizador pode acertar nessa escolha. Alguns fatores podem fazer com que a escolha dele não seja a melhor. Isso não quer dizer que ele escolheu um plano errado, mas que, mediante os cenários e cálculos que ele utilizou, talvez aquela opção tenha sido a mais sensata levando em consideração o modo como foi escrita a instrução, a memória da máquina naquele momento, o tráfego de acesso ao banco ou qualquer outro fator. Por isso, é importante que você conheça esses conceitos. Através desse conhecimento você poderá influenciar o otimizador a tomar outros rumos, seja dando dicas a ele, modificando nossos programas ou simplesmente ajudando-o a melhorar este caminho escolhendo outra tabela como tabela de condução. Agora vamos ver em mais detalhes alguns pontos interessantes que estão ao nosso alcance para ajudar o otimizador a tomar a melhor escolha.

7.1 ORDENANDO TABELAS E FILTROS

Como já vimos, o Oracle processa uma tabela de cada vez. Detalhando mais um pouco esse conceito, o Oracle inicia recuperando todos os dados referentes à tabela de condução (`Driving` ou `Outer Tables` , como podem ser chamadas). Uma vez que esses dados são recuperados, eles são utilizados para limitar o número de linhas processadas nas tabelas subsequentes (`Joins` , `Driven` ou `Inner Tables` , como podem ser chamadas). No caso de junções ou múltiplas junções, ou seja, onde há mais de duas tabelas se relacionando, a tabela controle acaba tendo suas linhas limitadas, por predicados de filtro, por exemplo, e com isso acaba limitando o processamento para a primeira tabela de junção.

Dessa forma, você deve considerar que uma tabela sob o efeito

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

de filtros pode acabar ficando com menos registros que as demais, considerando sua totalidade sem a aplicação desses filtros. Uma vez processado, esse conjunto de dados é passado para a segunda tabela de junção, que vai seguindo o fluxo em direção às demais tabelas da junção. O melhor a ser feito é primeiramente processar tabelas que recuperam um pequeno número de linhas e ir deixando as tabelas com maior número de linhas mais para o final. O otimizador com certeza conhece essa regra e vai tentar montar o plano de execução com base nessas premissas, contudo, alguns fatores podem influenciá-lo a agir de outra forma como também já vimos. Por isso, é importante se certificar de que o plano está adequado, principalmente quando forem detectados problemas de performance que já estão acontecendo. Feito isso, cuidando da forma como ordenamos os objetos e seus relacionamentos, podemos ajudar o otimizador a escolher o melhor caminho por essa ordenação informada, caso ele não consiga definir um melhor caminho através de outros meios.

Relembrando os conceitos, já sabemos que o uso de tabelas de condução é empregado tanto no modo baseado em custo, quanto no modo baseado em regras. Se estivermos trabalhando em um ambiente baseado em custo e o otimizador não conseguir definir qual a melhor tabela para ser a tabela de condução, ele assumirá os mesmos critérios utilizados pelo otimizador baseado em regras, onde o processamento ocorre a partir da extremidade da cláusula `from` em direção ao início dela. Para não correr o risco, devemos sempre colocar a tabela de condução no final da cláusula `from`. Quanto às outras tabelas de junção, devemos colocá-las na ordem, considerando as que recuperam maior número de linhas, deixando-as mais perto do início da cláusula `from`. Por outro lado, a cláusula `where` deve ser escrita em ordem inversa. As

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

condições referentes à tabela de condução devem estar em primeiro, logo no início da cláusula `where`. Já as condições relacionadas às tabelas de junção devem aparecer por último, seguindo o mesmo critério da cláusula `from` com relação à quantidade de registros recuperados. Veja o exemplo a seguir:

```
...
FROM  d, c, b, a
WHERE a.join_column = 12345
AND   a.join_column = b.join_column
AND   b.join_column = c.join_column
AND   c.join_column = d.join_column
```

Se queremos limitar as linhas da tabela `d`, fazemos da seguinte forma:

```
...
FROM  d, c, b, a
WHERE a.join_column = 12345
AND   a.join_column = b.join_column
AND   b.join_column = c.join_column
AND   c.join_column = d.join_column
AND   d.name = 'JONES'
```

Dependendo do número de linhas que a tabela `d` recupera agora, e talvez pela presença de índices, pode ser que o otimizador julgue que a tabela `d` seja uma melhor candidata à tabela de condução. Note que, agora, a tabela `d` possui mais critérios que permitem limitar ainda mais a quantidade de linhas. Se isso for verdade, essa instrução SQL seria melhor escrita desta forma:

```
...
FROM  c, b, a, d
WHERE d.name = 'JONES'
AND   d.join_column = 12345
AND   d.join_column = a.join_column
AND   a.join_column = b.join_column
AND   b.join_column = c.join_column
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Conforme vimos no capítulo sobre performance através da sintaxe SQL, falamos que o otimizador pode reescrever uma instrução SQL executada a fim de tornar mais performática sua execução. Assim, se escrevermos a instrução mais adequadamente, certamente isso será um trabalho a menos para o otimizador, e também, de forma mais eficaz, o orientaremos a fazer com que a tabela `d` retorne menos linhas e, com isso, a torne uma tabela de condução eficiente.

Depois, como já sabemos, o Oracle identifica a tabela de condução e a utiliza para a junção com outra tabela da instrução SQL. O resultado dessa junção, que será um conjunto de linhas, será utilizado para realizar a junção com a próxima tabela. Isso até que todas as tabelas da instrução SQL estejam devidamente relacionadas, passando, ao final, por todo o plano de execução escolhido. Quanto menor o resultado inicial, ou seja, quanto mais linhas forem eliminadas nessa primeira leitura, mais rápida será a execução da instrução SQL.

Em resumo, você deve se certificar de que a maior parte das linhas estão sendo eliminadas logo nos primeiros passos da execução do seu SQL. Para isso, você deve observar quais os predicados de cláusula `where` eliminam a maioria das linhas e, em seguida, certificar se tal tabela está sendo utilizada na parte mais interna da execução, através da leitura do plano. Tendo em mente essa regra, certamente você terá sucesso nos ajustes.

7.2 FORÇANDO A DRIVING TABLE POR HINT

Lá no capítulo referente ao otimizador de consultas, vimos como cada modo trabalha. Vimos que, quando estamos

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

trabalhando com banco de dados em modo baseado em regras, o otimizador usa uma receita para determinar como ele executa tal instrução, baseando-se em um rank para isso. Nesses casos, o processo soa mais simples, mas não é eficiente. O modo baseado em custo é mais complexo, mas traz resultados muito melhores e também formas mais flexíveis para se tratar questões de performance. Contudo, mesmo trabalhando no modo CBO, existem formas de influenciar o otimizador para que ele trabalhe como se estivesse operando em modo RBO. O `hint` é uma delas.

Você pode utilizar `hints` para determinar a ordem das tabelas e, assim, determinar qual será a tabela de condução através da disposição das tabelas na cláusula `from`. Nos exemplos anteriores, vimos que a ordem das tabelas em uma instrução SQL executada em um banco de dados baseado em custo nem sempre é a escolhida pelo otimizador. Isso porque ele considera também outros fatores. Mas se mesmo assim, por ventura, você quiser que ele execute sua instrução SQL como se estivesse em modo baseado em regras, você pode utilizar um dos `hints`, `ORDERED` ou `LEADING`, para que o otimizador avalie sua instrução considerando aspectos deste modo de operação.

O `hint` `ORDERED`, por exemplo, inibe operações de análise de custo. Com isso, o otimizador usará aquele rank, isto é, uma receitinha de bolo, para executar a instrução. Em alguns casos, isso pode acelerar a execução de uma determinada instrução SQL. Contudo, muito cuidado, pois em um primeiro momento, ele poderá ser uma ótima escolha, mas, futuramente, à medida que as tabelas utilizadas na instrução SQL forem ganhando mais dados e ficando mais volumosas, ou o modelo de dados se tornar mais complexo, continuar com esses `hints` pode não ser a escolha

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

mais acertada. Isto, porque quando você fixa a ordenação das tabelas, o otimizador pode passar a não considerar outras informações. Sabemos que, conforme as tabelas vão crescendo, outros planos de execução vão sendo considerados pelo otimizador. Montar outros planos de execução também envolve escolher novas candidatas à tabela de condução. Se a ordenação estiver fixada através deste `hint`, o otimizador não experimentará essas novas candidatas e isso pode gerar sérios problemas caso a candidata atual não seja mais tão eficiente quanto antes. Então, tenha bastante cuidado quando utilizar este ou outros `hints`. Veja os exemplos a seguir.

Com o `autotrace` habilitado na sessão, a instrução SQL foi executada assim:

```
select e.first_name
      ,d.department_name
      ,l.street_address
      ,c.country_name
      ,r.region_name
from   employees e
      ,departments d
      ,locations   l
      ,countries   c
      ,regions     r
where  e.department_id = d.department_id
and    d.location_id   = l.location_id
and    l.country_id    = c.country_id
and    c.region_id     = r.region_id
```

O otimizador apresentou o seguinte plano de execução:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 229053977

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                 |   106 |  9010 |    10  (20)| 00:00:01 |
|*  1 |  HASH JOIN             |                 |   106 |  9010 |    10  (20)| 00:00:01 |
|*  2 |   HASH JOIN            |                 |    27 |  2025 |     7  (15)| 00:00:01 |
|*  3 |    HASH JOIN           |                 |    23 |  1288 |     5  (20)| 00:00:01 |
|   4 |     NESTED LOOPS       |                 |    23 |   966 |     2   (0)| 00:00:01 |
|   5 |      TABLE ACCESS FULL| LOCATIONS       |    23 |   621 |     2   (0)| 00:00:01 |
|*  6 |      INDEX UNIQUE SCAN| COUNTRY_C_ID_PK |     1 |    15 |     0   (0)| 00:00:01 |
|   7 |     TABLE ACCESS FULL | REGIONS         |     4 |    56 |     2   (0)| 00:00:01 |
|   8 |    TABLE ACCESS FULL  | DEPARTMENTS     |    27 |   513 |     2   (0)| 00:00:01 |
|   9 |   TABLE ACCESS FULL   | EMPLOYEES       |   107 |  1070 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 7.4: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap07img04code.txt

Nesse plano, ele escolheu a tabela `locations` como sendo a candidata à tabela de condução, seguida pelas tabelas e acessos a índices: `country`, `regions`, `departments` e `employees`. Contudo, se analisarmos a ordem das tabelas na cláusula `from`, veremos que ele não a respeitou. Muito provavelmente, dentro dos diversos planos que ele utilizou para testar a melhor execução, obedecendo a tal ordenação, ele não deve ter chegado a um resultado satisfatório. Mas somos teimosos e vamos influenciá-lo a executar na ordem que definimos na instrução SQL. Veja como ficou a instrução e o plano de execução após a alteração.

```
select /*+ ORDERED */
       e.first_name
      ,d.department_name
      ,l.street_address
      ,c.country_name
      ,r.region_name
from   employees e
      ,departments d
      ,locations   l
      ,countries   c
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
      ,regions     r
where  e.department_id = d.department_id
and    d.location_id   = l.location_id
and    l.country_id    = c.country_id
and    c.region_id     = r.region_id
```

Utilizamos o `hint` `ORDERED` inserindo-o na instrução SQL, logo depois da cláusula `select`. A sintaxe do `hint` é da forma como está no exemplo, com todos estes elementos, barras, asteriscos e um sinal de positivo. No capítulo sobre `hints`, sua sintaxe é demonstrada com mais detalhes. Agora vamos ver o plano de execução.

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3454436918

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                 |   106 |  9010 |    10  (20)| 00:00:01 |
|*  1 |  HASH JOIN             |                 |   106 |  9010 |    10  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL    | REGIONS         |     4 |    56 |     2   (0)| 00:00:01 |
|   3 |   NESTED LOOPS         |                 |   106 |  7526 |     7  (15)| 00:00:01 |
|*  4 |    HASH JOIN           |                 |   106 |  5936 |     7  (15)| 00:00:01 |
|*  5 |     HASH JOIN          |                 |   106 |  3074 |     5  (20)| 00:00:01 |
|   6 |      TABLE ACCESS FULL| EMPLOYEES       |   107 |  1070 |     2   (0)| 00:00:01 |
|   7 |      TABLE ACCESS FULL| DEPARTMENTS     |    27 |   513 |     2   (0)| 00:00:01 |
|   8 |     TABLE ACCESS FULL | LOCATIONS       |    23 |   621 |     2   (0)| 00:00:01 |
|*  9 |    INDEX UNIQUE SCAN  | COUNTRY_C_ID_PK |     1 |    15 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 7.5: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap07img05code.txt

Vejam que agora o otimizador respeitou a ordem com a qual descrevemos na instrução SQL. Ele fez isso, pois o influenciamos através do `hint`. Contudo, isso não quer dizer que estamos certos em nossa decisão, apenas quer dizer que o otimizador também sabe seguir ordens caso seja necessário ;-).

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Como foram utilizadas tabelas com quantidades de dados bem pequenas, não conseguimos avaliar questões relacionadas a custo e tempo de execução. No entanto, ao trabalhar com fontes de dados volumosas, isso será facilmente notado.

Embora eu esteja aqui fazendo vários alertas sobre `hints` e seus usos, quero deixar claro que este recurso é muito poderoso e, se usado de maneira adequada, pode lhe trazer ganhos, seja melhorando performance, atuando como solução paliativa, corrigindo temporariamente um determinado problema ou até definitivamente dependendo dos casos. No próximo capítulo, serão mostrados os `hints` mais utilizados e a função de cada um deles.

7.3 DRIVING TABLES E OS MÉTODOS DE JUNÇÃO

Apesar da maior ênfase em métodos de junção, quando falamos sobre `Driving Tables` , ao ver, por exemplo, planos de execução que envolvem o método `NESTED LOOPS` , este recurso não se limita a apenas este tipo de método de junção. Você verá que, em muitas literaturas, quando o assunto é `Driving Tables` , a maioria dos exemplos envolve esse método, não por uma questão de pré-requisito ou limitação, mas sim, pelo fato de ser o cenário onde vemos um impacto maior quando uma boa ou má escolha, por uma tabela de condução, acontece. A escolha por uma tabela

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

de condução também acontece em outros métodos de junção, no entanto elas não causam tanto impacto assim.

A `Driving Table` também é utilizada em junções do tipo `HASH JOIN`, por exemplo. Nesse caso, devemos nos certificar de que o otimizador escolheu a tabela com menos quantidade de linhas, ou, no caso de estar utilizando o `hint ORDERED`, de que a tabela com menos quantidade de linhas é a primeira tabela na cláusula `from`, da esquerda para a direita.

Como já vimos em métodos de junção, no caso do `HASH`, o otimizador cria um espaço na memória para abrigar o conteúdo de uma das tabelas, para então realizar a junção com as demais tabelas da instrução SQL. A tabela que o otimizador escolhe para colocar em memória é a tabela de condução. Por isso, a importância de a tabela de condução ser a tabela com menos registros possível.

Já quando estamos falando do método `MERGE JOIN`, a tabela de condução será a primeira da leitura, mas, neste caso, não fará muita diferença qual será a tabela de condução, pois ambas são ordenadas pelos atributos que as relacionam. Depois disso, os registros retornados de cada tabela são comparados e mesclados, satisfazendo os critérios de relacionamento. Depois de processados, eles são passados para o próximo nível do plano de execução.

Apenas reforçando algumas informações referentes aos métodos de junção - e isso é importante para entender os planos de execução e, consequentemente, o porquê da escolha de uma `Driving Table` - vale dizer que, no caso de o otimizador estar trabalhando em modo CBO (baseado em custo), e o parâmetro `OPTIMIZER_MODE` estar configurado como `FIRST_ROWS`, o

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

otimizador procurará eliminar a maioria das linhas através de índices, caso existam. Muito provavelmente, com isso, ele tentará utilizar o método de junção `NESTED LOOPS`. No caso de o `OPTIMIZER_MODE` estar configurado para `ALL_ROWS`, o otimizador procurará pelo melhor rendimento, com possibilidade de usar o `MERGE JOIN` como método de junção. Desta forma, você pode avaliar qual tabela o otimizador escolheu como tabela de condução e procurar entender isso no contexto do método que ele escolheu para buscar os dados. O que eu quero dizer com isso é que você não deve se limitar somente a determinadas configurações ou métodos. Analise o contexto como um todo antes de sair modificando seus programas, seu banco de dados ou demais procedimentos.

Nota: Lembre-se de que não existe um modo de operação para o otimizador chamado `Custo`, conforme vimos no capítulo sobre o otimizador de consultas. Se o banco de dados estiver operando em modo baseado em custo, o `OPTIMIZER_MODE` poderá estar configurado como `FIRST_ROWS` ou `ALL_ROWS`, por exemplo.

7.4 DRIVING TABLES E OS PLANOS CARTESIANOS (CARTESIAN JOIN)

Nem sempre será possível montar um plano de execução onde todas as tabelas se relacionem de forma natural e ainda façam isso da melhor maneira possível. Quando estamos falando de um conjunto de tabelas que possuem relacionamentos híbridos, onde

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

uma tabela pode estar se relacionando com outras tabelas de forma direta na mesma instrução SQL, não caracterizando um relacionamento totalmente aninhado (forma natural), como `A` com `B`, `B` com `C` e `C` com `D`, a escolha pela tabela de condução mais adequada pode ser uma tarefa que exige do otimizador bem mais trabalho na hora de montar os planos de execução. Veja o exemplo:

```
select first_name, department_name, street
from   employees   e
      ,departments d
      ,locations   l
where e.department_id = d.department_id
and   d.location_id   = l.location_id;
```

Nessa instrução SQL, estamos selecionando dados de três fontes diferentes. Note que existem dependências entre as tabelas. A tabela de empregados (`employees`) depende da tabela de departamentos (`departments`), que depende da tabela de localização (`locations`). Existem dependências que podemos chamar de diretas e indiretas: a tabela de empregados está ligada diretamente à tabela de departamentos e indiretamente ligada à tabela de localização, pois se quisermos saber o endereço de onde o empregado trabalha, precisamos primeiro saber qual o seu departamento. Pois bem, considerando que o otimizador optasse por utilizar o método `NESTED LOOPS` para construir esse relacionamento, poderíamos representar o plano de execução da seguinte forma:

```
for tabela_empregados loop
  for tabela_departamentos loop
    for tabela_localizações
      resultado;
    end loop;
  end loop;
end loop;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Note que o loop mais externo (`outer join`) é o representado pela tabela de empregados. Os loops mais internos (`inner join`) são representados pelas tabelas de departamentos e localizações. Observe também que aqui nesse exemplo a tabela de empregados é a nossa tabela de condução, pois esse tipo de tabela sempre está presente no loop mais externo.

Este seria o melhor dos mundos em termos de relacionamento, pois estamos trabalhando com tabelas que estão se relacionando de forma natural como estão descritas na instrução SQL. Contudo, nem sempre será assim. Na verdade, quanto mais complexa for a nossa instrução SQL, menos chance o otimizador terá para gerar planos de execução que representem exatamente a forma natural de como estão descritos os relacionamentos na instrução.

É de fundamental importância saber dessa informação, pois muitas vezes o plano de execução acaba ficando confuso de ser lido, pois temos em mente os relacionamentos que estão descritos na instrução SQL e, de repente, olhamos para plano e vemos referências às tabelas sendo relacionadas ou acessadas de forma diferente ou aparecendo várias vezes dentro do mesmo plano execução, ou ainda, buscando dados onde tais relacionamentos não existem na íntegra, descritos na instrução. Mas, calma! Agora que você sabe disso, certamente vai olhar um plano de execução com outros olhos.

O otimizador, como já foi dito, vai analisar vários aspectos para gerar diversos planos de execução, inclusive modificar a forma como as tabelas estão relacionadas na instrução para se chegar a um melhor resultado. No capítulo sobre performance através da sintaxe SQL, mostramos como o otimizador pode transformar

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

uma instrução SQL, a fim de torná-la mais simples e performática. Pois bem, como ele precisa gerar diversos planos de execução para uma instrução a fim de buscar o melhor plano, ele também pode incluir ou modificar relacionamentos em prol de gerar planos de execução mais eficientes, mesmo que isso implique em juntar tabelas independentes (tabelas que não se relacionam diretamente na instrução SQL) gerando um `Cartesian join` (Plano Cartesiano) ou dar a impressão de relacioná-las mais de uma vez no mesmo plano de execução.

Agora você deve estar pensando: mas isso não se tornaria um retrabalho? E não poderia gerar um grande esforço por parte do otimizador? Ou seja, selecionar uma determinada tabela mais de uma vez em diferentes pontos do plano, sendo que ele poderia buscar tudo o que precisasse dela de uma única vez ou processar uma grande quantidade de dados derivados de um Plano Cartesiano, sabendo que o resultado disso é exponencial, não seria perda de tempo? Você tem razão, sim, existe todo esse esforço, mas para um fim: resolver conflitos nas buscas pelos dados com o objetivo de alcançar o menor custo e consequentemente um menor tempo no retorno do resultado, nem que para isso ele precise dar mais voltas.

Fazendo uma analogia, imagine que você está percorrendo uma distância de 5km carregando um peso de 20 quilos e, ao iniciar o percurso, você nota que com esse peso você conseguiria chegar no máximo a uma velocidade de 5km/h, levando 1 hora para concluir o percurso. Achando que seu tempo não está bom, em uma outra oportunidade, você tenta fazer algo diferente: em vez de percorrer o percurso de uma única vez, levando 20 quilos, você decide fazer duas viagens carregando 10 quilos em cada um

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

dos percursos. Para sua surpresa, você percebe que levando apenas 10 quilos, seu rendimento e ritmo são melhores e você consegue chegar a uma velocidade de 11km/h, levando 27 minutos em cada percurso, totalizando 6 minutos a menos se juntarmos as duas corridas em relação à primeira corrida de 1 hora realizada da primeira vez, mesmo que, nesse caso, você acabe percorrendo 1 km a mais. Contudo, se a meta era ser mais rápido em relação ao primeiro tempo, o objetivo foi alcançado. O mesmo ocorre com o otimizador, ele poderá escolher realizar mais tarefas se o tempo de resposta no final das contas for melhor. Mas lembre-se de que, para isso, seu otimizador precisará apresentar um bom rendimento e manter o ritmo. Traduzindo para `bits`, ele precisará de poder de processamento e de memória para cumprir o trajeto.

Tendo isso em vista, pode ser comum você visualizar planos de execução em que o otimizador, em um determinado momento, realize um acesso a uma determinada tabela ou a um índice criado para ela apenas buscando os `rowids` que satisfaçam um predicado informado na cláusula `where` e, em um outro momento, utilize esse conjunto de resultados para realizar uma junção com outra tabela e, talvez em um outro momento, acessar a tabela propriamente dita para buscar demais informações que estão sendo solicitadas na cláusula `select`. Em outros casos, ele poderá acessar a tabela várias vezes se ele julgar necessário. Veja o exemplo a seguir.

Nesta instrução SQL, estamos selecionando informações do empregado, como nome, salário e departamento. Vamos olhar o plano de execução que o otimizador escolheu para ela. Antes disso, note que um plano de execução pode não ser semelhante a este, se executarmos em um banco de dados diferente, com amostragem

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

de dados ou configurações que diferem das do exemplo em questão. A versão do banco de dados também pode influenciar principalmente se você estiver utilizando as configurações padrões de instalação, ou se você estiver utilizando versões de banco de dados que já instalam uma pré-configuração automaticamente. Portanto, se você estiver querendo reproduzir este exemplo, ele pode não estar mostrando o mesmo plano de execução executado aqui.

```
select e.first_name, e.last_name, e.salary, d.department_name
from   hr.employees e, hr.departments d
where  d.department_name in ('Marketing', 'Sales')
and    e.department_id = d.department_id;
```

Vejamos agora o plano de execução:

```
Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes|Cost(%CPU)|    Time|
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |    19|   722|    3  (0)| 00:00:01|
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |    10|   220|    1  (0)| 00:00:01|
|   2 |   NESTED LOOPS               |                   |    19|   722|    3  (0)| 00:00:01|
|*  3 |    TABLE ACCESS FULL         | DEPARTMENTS       |     2|    32|    2  (0)| 00:00:01|
|*  4 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |    10|      |    0  (0)| 00:00:01|
----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("D"."DEPARTMENT_NAME"='Marketing' OR "D"."DEPARTMENT_NAME"='Sales')
   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Figura 7.6: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap07img06code.txt

Se analisarmos o plano de execução, veremos que o otimizador escolheu como tabela de condução a tabela `departments`. Nesse exemplo, é a melhor candidata para essa função, pois é a tabela que trará a menor quantidade de registros. Vimos também que o método de acesso utilizado por ele foi o `NESTED LOOPS` (seleção

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

por loops aninhados). Em uma analogia de programação, dizemos que se trata de um loop dentro de outro loop. Provavelmente, ele foi usado pelo fato de existirem índices em colunas que participam da cláusula `where`. Se formos tentar visualizar a forma como essas tabelas serão relacionadas no método escolhido, traduzindo para `PL/SQL`, por exemplo, ficaria mais ou menos assim:

```
for departments loop -- tabela de condução (driving table ou  out
er table) / outer loop
  for employees loop -- tabela de junção (driven table ou inner t
able) / inner loop
    ...
  end loop;
end loop;
```

Vale lembrar que métodos `NESTED LOOPS` são mais eficazes quando estamos falando de tabelas com pequenas quantidades de dados e que possuem índices em suas colunas de junção. Se uma tabela possui apenas uma linha, como acontece no caso de pesquisas de igualdade, e esta coluna for uma chave primária, por exemplo, então se tratará de uma pesquisa simples. O otimizador sempre tenta colocar a tabela que retornará a menor quantidade de linhas, como sendo a tabela de condução.

Ok. Sabendo de todas essas informações, se voltarmos ao plano, podemos ver que:

- O otimizador fez um acesso completo na tabela `departments` (`TABLE ACCESS FULL` - passo 3 do plano) em busca de dados que obedeçam ao filtro que estamos solicitando na instrução SQL, ou seja, somente os departamentos que tenham como nome `'Marketing'` e `'Sales'`. Como não temos um índice criado para a coluna `department_name`, o otimizador precisou varrer

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

todas as linhas da tabela em busca desta condição. Com isso, ele retornou um conjunto de linhas (chamado de `Row Sources` ou `Result Set`) que foi utilizado na junção com a primeira tabela.

- Para cada linha encontrada na tabela `departments` que atendeu à condição anterior, ele realizou uma busca no índice da tabela `employees` (chamado `EMP_DEPARTMENT_IX` - passo 4). Neste caso, refere-se a um índice criado para a coluna `DEPARTMENT_ID` , conforme você pode verificar pela `view` de banco `user_ind_columns` ou `all_ind_columns` , que mostram tabelas e suas respectivas colunas indexadas, buscando `rowids` que satisfaçam os critérios de junção, estabelecendo o relacionamento entre as duas tabelas (`NESTED LOOPS` - passo 2). O método de acesso utilizado pelo otimizador foi o `INDEX RANGE SCAN` , ou seja, uma leitura por faixa de índice.
- Para finalizar, o otimizador, após ter obtido os `rowids` que entraram em concordância com os filtros e junções utilizados, acessa a tabela `employees` propriamente dita (`TABLE ACCESS BY INDEX ROWID` - passo 1) por meio dos `rowids` retornados anteriormente, para buscar as demais informações pertinentes aos empregados.

Por isso, falamos em interpretação do plano de execução. A grande sacada é entender os meios, os métodos, os acessos, isto é, de quais recursos o otimizador dispõe para montar estes planos e quais combinações são possíveis de serem realizadas. Conhecendo esses pontos, você conseguirá entender certos aspectos e propor mudanças que podem melhorar a forma como seus programas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

estão escritos. O bacana disso tudo é que, ao praticar essas técnicas, você acaba aprendendo a pensar como as coisas funcionam, até chegar a um nível em que, ao escrever novos programas, você já começa a deixar alguns vícios para trás sabendo o que eles podem acarretar no futuro.

Acabamos de ver que a escolha pelo otimizador em definir qual será a tabela de condução implica no tipo de método e no tipo de acesso que ele terá que utilizar para satisfazer a todas as condições. Note que este início, esta escolha de qual tabela será a condutora, aquela que iniciará todo o processo de busca pelas informações, é algo de fundamental importância e é o que vai nortear os próximos passos até a execução final do programa.

Eu diria que a escolha da tabela de condução é tão importante para definir qual o melhor plano de execução para uma instrução SQL que, inclusive, em alguns casos leva o otimizador a realizar certas junções que você julgaria "incorretas" ou inadequadas. Falei aqui, ainda há pouco, que o otimizador do banco de dados Oracle pode realizar junções entre tabelas que não estão sendo relacionadas de forma direta na instrução SQL. Talvez, quando você viu isso lá nas páginas anteriores, ficou com dúvida do que eu realmente queria dizer com aquilo. Pois bem, chegou o momento.

Uma das regras que o otimizador leva em conta no momento de relacionar as tabelas em um método `NESTED LOOPS` - isso após ter definido qual será a tabela de condução - é que, se o caminho de acesso para o loop interno não for dependente do loop externo, ou seja, se as tabelas escolhidas por ele, tanto para ser a tabela de condução, quanto para ser as tabelas de junção, na ordem em que devem ser acessadas, não estiverem naturalmente se relacionando,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

então o resultado pode ser um produto cartesiano. Nesse caso, para cada iteração do loop externo, o loop interno produzirá o mesmo conjunto de linhas.

Fazendo uma analogia, seria como se escrevêssemos uma instrução SQL contendo duas ou mais tabelas, sem a cláusula `where` definindo como elas estariam relacionadas. O que é mais irônico é que, quando esquecemos e fazemos isso ao escrevermos uma instrução SQL, isto é, não definimos um ou mais relacionamentos entre as tabelas de uma cláusula `from`, geralmente, isso se torna um problema de performance. Então, sim, se o otimizador julgar vantajoso, ele gerará planos cartesianos. Por outro lado, ele sabe que terá que resolver esse plano cartesiano mais tarde, de forma consistente, para que tais planos não afetem o resultado final. De todo modo, essa pode ser, sim, uma opção a ser utilizada pelo otimizador, como você pode ver no exemplo a seguir. Veja a instrução SQL:

```
SELECT /*+ORDERED*/ e.last_name, d.department_name, l.country_id,
 l.state_province
FROM   employees e, locations l, departments d
WHERE  e.department_id = d.department_id
AND    d.location_id = l.location_id
```

No exemplo, estamos selecionando dados de empregados, departamentos e os locais desses departamentos. Note que estamos relacionando todas as tabelas, sendo que a relação dos empregados acontece com os departamentos através do id do departamento, e a relação dos departamentos com as localizações através do id da localização. O que temos de diferente nessa instrução SQL é que estamos utilizando o `hint` `ORDERED`. Desta forma, o otimizador tomará como regra, para essa instrução SQL, exatamente a ordem como se segue. Isso quer dizer que a tabela de condução será a

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

primeira tabela da cláusula `from` (agora, seguindo a sequência da esquerda para a direita).

> **Nota**: Não confunda a sequência de execução normal com a execução que acontece da direita para a esquerda quando não estamos forçando através do `hint`. Então, se olharmos para a instrução SQL, vemos que a tabela de condução a ser utilizada é a tabela `employees`, seguida pela tabela `locations` e `departments`. Essa é uma forma de testarmos cenários onde o otimizador utiliza relacionamentos do tipo `CARTESIAN join` (Planos Cartesianos) no plano de execução.

Contamos com a ajuda deste `hint`, pois, caso contrário, poderíamos levar algum tempo para demonstrar a utilização deste tipo de relacionamento. Na verdade, teríamos que provocar um caso onde o otimizador julgasse necessário optar por este caminho por conta própria, o que poderia levar algum tempo. Também é preciso deixar claro que casos envolvendo o uso de `CARTESIAN join` aparecem com frequência em instruções SQL muito grandes, mesmo não havendo `hints` forçando a sequência de leitura das tabelas na cláusula `from`. Mesmo que exista uma ordem natural entre os relacionamentos das tabelas envolvidas, o otimizador pode escolher utilizar este método se ele achar conveniente.

Mas continuando, com base nesta instrução SQL, vamos ver o plano de execução escolhido pelo otimizador.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3738798407

-----------------------------------------------------------------------------
| Id  | Operation              | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |             |   106 |  4664 |    36  (3)| 00:00:01 |
|*  1 |  HASH JOIN             |             |   106 |  4664 |    36  (3)| 00:00:01 |
|   2 |   TABLE ACCESS FULL    | DEPARTMENTS |    27 |   513 |     2  (0)| 00:00:01 |
|   3 |   MERGE JOIN CARTESIAN |             |  2461 | 61525 |    33  (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL   | EMPLOYEES   |   107 |  1177 |     2  (0)| 00:00:01 |
|   5 |    BUFFER SORT         |             |    23 |   322 |    31  (0)| 00:00:01 |
|   6 |     TABLE ACCESS FULL  | LOCATIONS   |    23 |   322 |     0  (0)| 00:00:01 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID" AND
              "D"."LOCATION_ID"="L"."LOCATION_ID")
```

Figura 7.7: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap07img07code.txt

Olhando o exemplo, vemos o `CARTESIAN join` no plano de execução, como esperado. Obedecendo ao `hint` `ORDERED`, o otimizador utilizou a tabela `employees` como tabela de condução (passo 3). Esta tabela é relacionada com a tabela `locations` (passo 6). Como não há um relacionamento entre `employees` e `locations`, mas ele precisa respeitar a ordem imposta pelo `hint` definido, o otimizador precisou gerar um plano cartesiano das duas tabelas (você também pode ouvir falar em Produto Cartesiano), o que gerou 2.461 linhas, sendo 23 linhas da tabela `locations`, multiplicadas pelas 107 linhas da tabela `employees` (Produto Cartesiano = tabela a x tabela b x tabela c...). O próximo passo é relacionar a tabela `departments` com o resultado cartesiano anterior, através das colunas `department_id` e `location_id`, filtrando por essas colunas e satisfazendo aos

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

filtros derivados dos relacionamentos entre essas tabelas.

Você percebe como não é complicado a partir do momento em que você entende o conceito? Pois bem, sei que no princípio não é fácil. Eu mesmo, quando vi um plano em execução a primeira vez, não entendia muita coisa, mas com o tempo e conhecendo os conceitos por trás das técnicas, comecei a ver de forma natural como as coisas aconteciam. O mesmo pode acontecer com você. O segredo é se familiarizar, treinar e buscar cada vez mais contato com o assunto.

Caso o otimizador esteja escolhendo realizar operações `CARTESIAN join` mesmo não utilizando o recurso de `hints`, e o problema de performance seja exatamente derivado dessas operações, ele poderá ser evitado, utilizando-se de outros métodos de junção para juntar duas ou mais tabelas independentes ou para fazer com que outros relacionamentos com outras tabelas possam ser utilizados para evitar tais operações.

Então, falamos bastante sobre como o otimizador é influenciado e sobre a escolha da tabela de condução, na grande maioria das vezes, explanando exemplos em que o método de junção escolhido era o `NESTED LOOPS`. Isso não foi por acaso. Outros métodos também utilizam alguns destes conceitos, como já foi dito, por exemplo, o `HASH JOIN`, contudo, os maiores impactos e evidências ocorrem quando o otimizador resolve utilizar o `NESTED LOOPS` como método de junção para juntar duas ou mais tabelas. Como sua eficiência é nata, você verá também que, em grande parte, o otimizador buscará opções de utilizá-lo, e por isso a importância de entender seu funcionamento e características junto à montagem do plano, bem como isso possa

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

influenciar ou ser influenciado.

Comentamos também sobre como o otimizador faz a escolha por qual tabela será a tabela de condução e também que existe um entendimento comum, mencionado na maioria das literaturas encontradas, de que o otimizador sempre escolherá a tabela com menos registros (sendo pela quantidade total dessa tabela, ou pela quantidade após aplicados os filtros) para ser a escolhida como tabela de condução. No entanto, outros aspectos podem levar o otimizador a ignorar essas regras, como a forma como os dados são armazenados. Dependendo de como foi definida a estrutura de armazenamento dos dados, isso pode influenciar o otimizador na hora de fazer uma escolha.

Mas não se preocupe, na grande maioria das vezes a causa dos problemas em instruções SQL são ocasionados por desenvolvimentos ou uma modelagem de dados mal desenhada, ou por mau uso dos recursos do banco de dados. Garantindo essas duas pontas, o resto ficará bem.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 8

OTIMIZANDO ATRAVÉS DE HINTS

Até aqui, vimos diversas técnicas para melhorar a performance e escrever instruções SQL melhores. O conhecimento adquirido sobre o otimizador também trará benefícios hábeis no auxílio do processo de otimização. Embora o otimizador seja preciso na escolha do plano de execução, visando sempre a uma maior assertividade, em alguns casos, dependendo de vários fatores que vão desde a escrita da instrução SQL, modelagem do banco de dados e até questões de ambiente, ele pode não satisfazer os critérios de desempenho esperados. Portanto, você deve estar preparado para realizar otimizações de forma manual, principalmente quando se está usando a otimização baseada em custo, pois o otimizador pode assumir que os dados estejam uniformemente distribuídos ou estruturados da forma adequada, o que não necessariamente será verdade - se, por exemplo, as estatísticas do banco não estiverem sendo geradas, ou geradas de forma inadequada.

Se os dados não se encontram em um ambiente adequado e bem estruturado, o otimizador pode não conseguir escolher o melhor plano de execução para determinada instrução SQL. Vimos também que a forma como fazemos as declarações SQL, se não

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

forem bem escritas, pode confundir o otimizador e fazer com que ele trabalhe mais para chegar a determinado resultado. A solução seria, então, avaliar o banco de dados ou instruções específicas e propor melhorias em suas estruturas.

Falando de instrução SQL, em muitas delas, para resolver problemas de performance, o ideal seria reescrevê-las para obter um melhor resultado. No entanto, sabemos que os sistemas podem ser complexos e partes de alguns sistemas ou instruções deles podem ser enormes e cheias de meandros, inclusive a começar pela modelagem do banco de dados em questão. Além disso, temos o tempo e o custo de mexer nelas. Muitas vezes é preciso de rapidez na solução, outras vezes não há intenção de reescrevê-las. Com isso, precisamos achar outras alternativas para melhorar o cenário, apostando na criação de índices, como já vimos anteriormente, ou através de `hints`. Eu diria que, para melhorar a performance de um programa pensando apenas no escopo de desenvolvimento, essas seriam as duas formas de tentar resolver um problema de performance sem alterar, significativamente, uma instrução SQL problemática (isto, claro, depois de ter descartado outros possíveis problemas que possam estar relacionados a tráfego de rede, memória, processamento etc., ou seja, temas fora do escopo da aplicação propriamente dita). Sobre índices, já falamos anteriormente, agora veremos detalhes sobre os `hints` e suas principais caracterterísticas e formas de uso.

Um `hint` nada mais é que uma sugestão que é dada ao otimizador no momento em que estamos escrevendo uma instrução SQL, para que ele trabalhe de uma determinada forma. Sua principal função é influenciar no trabalho do otimizador, uma vez que são levados em conta no momento da geração dos planos

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

de execução. Por exemplo, se para uma determinada tabela existe um índice que poderia ser usado pelo otimizador, mas na instrução SQL foi especificado um `hint` `FULL TABLE SCAN` para essa mesma tabela, o otimizador vai ignorar este índice, aceitando a sugestão do `hint` de fazer uma varredura completa na tabela.

Através dos `hints`, é possível dar como sugestão ao otimizador:

- A abordagem para a execução da instrução SQL;
- O caminho para quando a abordagem é baseada em custo;
- O caminho de acesso para as tabelas acessadas pela instrução SQL;
- A ordem dos `joins` entre as tabelas na instrução SQL;
- As operações a serem efetuadas entre os `joins` na instrução SQL.

Quanto ao uso dos `hints`, somente se aplicam em blocos de comando. Um bloco de comando pode ser composto por instruções SQL ou partes dela, como um simples comando `select`, `update`, `delete`, uma subinstrução (subquery) de um comando complexo, ou ainda em parte de uma instrução composta.

O uso dos `hints` proporciona formas de trabalhar de acordo com o que foi especificado neles, sendo possível mudar a maneira como o otimizador montará o plano de execução e permitindo situações mais específicas, tais como:

- Cancelar o método corrente de otimização;
- Cancelar a escolha do método de junção pelo otimizador;
- Cancelar a escolha do método de acesso pelo otimizador.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

A sintaxe para a declaração dos `hints` em uma instrução SQL é muito semelhante a inserir um comentário, na linguagem SQL, pois usa os mesmos caracteres para tal, salvo pelo fato de utilizar o operador de soma (+) em conjunto com o nome e parâmetros do `hint`. A seguir, temos a sintaxe:

```
SELECT /*+ hint */ ou --+ hint
```

Veja um exemplo em que especificamos o `hint` `FIRST_ROWS` (não se preocupe, veremos mais à frente o que significa cada um deles):

```
SQL> select /*+ FIRST_ROWS */ first_name, salary
  2  from employees
  3  where department_id = 10;
```

Podemos utilizar múltiplos `hints` em uma mesma instrução SQL. Veja a sintaxe:

```
select /*+ FULL(nome_da_tabela) CACHE(nome_da_tabela) */ atributo
1,...
```

Segue outro exemplo com múltiplos `hints`, utilizando os `hints` `FULL` e `CACHE`:

```
SQL> select /*+ FULL(employees) CACHE(employees) */ employee_id,
first_name, department_id
  2  from employees
  3  where department_id = 10;
```

Uma observação importante para quando usamos alias em tabelas, e elas forem especificadas nos `hints`, é não usar o nome das tabelas e sim seus alias. Veja o exemplo:

```
select /*+ FULL(A) */ employee_id, first_name, department_id
from employees a
where department_id = 20;
```

Nesse exemplo, o caractere `'A'` entre parênteses em

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`FULL(A)` refere-se ao alias da tabela `employees` .

Caso um `hint` seja especificado de forma incorreta, o otimizador vai ignorá-lo, mas não retornará nenhum erro. Nesse caso, a instrução SQL será executada normalmente, porém o otimizador não considerará o `hint` no momento em que estiver gerando seus planos de execução. Você poderá perceber isso pelo fato de ele não estar obedecendo às características do `hint` . A combinação de vários `hints` também é permitida, mas deve-se tomar cuidado para não usar combinações de `hints` que não sejam compatíveis entre si, pois o otimizador também vai desconsiderá-los nesse caso. Por exemplo, ao mesmo tempo em que você está solicitando, através de `hints` , que ele utilize um determinado índice de uma tabela, você está pedindo também que ele faça a varredura completa dela. Sabendo o significado de cada `hint` , é possível fazer as combinações de forma correta.

Os `hints` podem ser divididos em duas principais categorias: diretivas de uso e diretivas de compilador. As diretivas de uso são aquelas que podem ser configuradas em nível de parâmetro, além de em nível de instrução, como as diretivas de compilador. Em capítulos anteriores, foi visto como configurar o modo de trabalho do otimizador (`OPTIMIZER_MODE`) através dessas diretivas. Existem `hints` específicos, separados em categorias, para várias situações, baseando-se no tipo da operação que está sendo modificada. Há `hints` para mudar o caminho de execução de uma instrução, outros para determinar o método de acesso, outros para transformar a instrução e também para operações de junção. A seguir, estão alguns destes `hints` separados por categorias.

- **hints de mudança de caminho de execução:** `ALL_ROWS` ,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CHOOSE , FIRST_ROWS(n) , RULE .

- **hints de método de acesso:** AND_EQUAL , CLUSTER , FULL , HASH , INDEX , INDEX_ASC , INDEX_COMBINE , INDEX_DESC , INDEX_FFS , INDEX_JOIN , NO_INDEX , ROWID .
- **hints de transformação de consulta:** REWRITE , NOREWRITE .
- **hints usados em execução paralela:** NOPARALLEL , NOPARALLEL .
- **hints de operação de junção:** DRIVING_SITE , LEADING , PUSH_SUBQ , USE_HASH , ORDERED , USE_MERGE , USE_NL .
- **outros hints:** APPEND , NOAPPEND , NOCACHE , CACHE , ORDERED_PREDICATES .

Além desses hints , existem outros que estarão disponíveis de acordo com a versão do banco de dados que você estiver utilizando. Os hints apresentados aqui, por exemplo, estão disponíveis a partir da versão do banco de dados Oracle 9i. Para maiores informações, você deve consultar a documentação que acompanha o banco de dados, no site ou manuais da Oracle.

Em seguida, vamos falar sobre alguns destes hints , os mais vistos em planos de execução, e também demonstrar exemplos para um melhor entendimento. Vale lembrar que cada ambiente de banco de dados pode ser diferente e possuir suas próprias características e peculiaridades, não havendo uma fórmula ou cenário padrão para que a técnica de hints seja utilizada. No entanto, gostaria de frisar que o uso de hints precisa ser muito consciente, principalmente se for pensando em algo a longo prazo. Pode ser que em um dado momento o uso de um determinado

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`hint` pode trazer bons ganhos, pela quantidade de registros, dos índices disponíveis ou por qualquer outro motivo, mas à medida que os cenários vão se modificando ao longo do tempo, o `hint` pode passar de solução a problema da história.

Eu gosto de imaginar os `hints` como formas rápidas para testar se um determinado cenário pode ou não dar certo. Quando estou analisando um programa, às vezes uso `hints` para tentar forçar um `FULL TABLE SCAN` ou o uso de um índice em específico, trocar um método de junção ou de acesso, por exemplo. Caso funcione, tento descobrir porque o otimizador não segue o mesmo caminho sem que, para isso, eu precise forçá-lo. Às vezes dá certo, e então altero a instrução SQL ou crio um índice, por exemplo, mas outras vezes não chego a uma solução.

Quando isso acontece, quando não consigo o resultado esperado sem forçá-lo, o jeito mesmo é manter o `hint` . Depois, quem sabe, em uma outra ocasião, caso os cenários de performance mudem novamente, eu tente suprimir os `hints` , caso eles existam na instrução, para ver se ainda é necessário o seu uso. Às vezes ele pode ser o problema devido à ação do tempo. Como disse, seu uso precisa ser bem consciente, pois com o passar do tempo ele poderá necessitar de manutenção.

Você também vai notar que muitos destes `hints` forçam situações cujos conceitos já foram vistos em capítulos anteriores, ou serão vistos mais à frente, como `hints` sobre uso de índices, métodos de junção ou métodos de acesso. Desta forma, não entraremos nos detalhes destes conceitos, apenas falaremos sobre os `hints` em si e seus objetivos. Veja agora detalhes sobre alguns `hints` que mais vemos em planos de execução e seus usos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

8.1 HINT CHOOSE

Este `hint` tem o objetivo de forçar a otimização baseada em custo. Ele pode ser especificado através do parâmetro `OPTIMIZER_MODE`, do arquivo de configuração do Oracle, `init.ora`, ou em nível de sessão. Veja o exemplo a seguir, em que, através deste `hint`, estamos forçando o otimizador a trabalhar em modo baseado em custo, mesmo que ele não tenha todas as informações necessárias para isso.

```
SQL> select /*+ CHOOSE */ employee_id, first_name, department_id
  2  from   employees
  3  where  department_id= 30;

EMPLOYEE_ID FIRST_NAME           DEPARTMENT_ID
----------- -------------------- -------------
        114 Den                             30
        115 Alexander                       30
        116 Shelli                          30
        117 Sigal                           30
        118 Guy                             30
        119 Karen                           30

6 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 2056577954

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                   |     6 |    84 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     6 |    84 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     6 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DEPARTMENT_ID"=30)
```

Figura 8.1: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img01code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Veja que uma das características do modo baseado em custo é mostrar a informação sobre o custo de cada operação do plano. Isso pode ser observado através da presença da coluna `Cost (%CPU)` no plano de execução.

> **NOTA:** Tendo em vista que esse `hint` utiliza a otimização baseada em custo e, para isso, é preciso que estatísticas sejam geradas, a tabela usada na consulta deve ter sido analisada. Se a instrução SQL contiver várias tabelas, havendo junções, será necessário que pelo menos uma das tabelas tenha sido analisada. Nesse caso, o otimizador utilizará a otimização baseada em custo para todas as outras tabelas.

8.2 HINT RULE

Seu objetivo é forçar o otimizador a trabalhar no modo baseado em regras. Os `hints`, na grande maioria das vezes, têm o papel de influenciar o otimizador em cenários de otimização baseada em custo. Já o `hint RULE`, pelo contrário, força a otimização baseada em regras, ou seja, as informações estatísticas mais apuradas e detalhadas sobre as tabelas não são mais consideradas, e sim, um conjunto de regras predefinidas no Oracle, como o uso do `rowid` ou de índices exclusivos, por exemplo. Isso faz com que a consulta em questão seja totalmente controlada pelo otimizador. Ele escolherá qual o melhor caminho, sem considerar outros aspectos. Com a especificação do `hint RULE`, também são ignorados todos os outros `hints`, exceto os `hints ORDERED` e `DRIVING_SITE`. Esse `hint` pode também ser

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

definido através do parâmetro `OPTIMIZER_MODE`, no arquivo de parâmetros do Oracle, `init.ora`. O exemplo a seguir mostra o uso desse `hint` e é possível observar que, no plano de execução, não são mostradas informações sobre o custo. Veja também que, em nota, o Oracle sugere que seja usado o modo baseado em custo (CBO).

```
SQL> select /*+ RULE */ employee_id, first_name, department_id
  2  from employees
  3  where department_id = 40;

EMPLOYEE_ID FIRST_NAME           DEPARTMENT_ID
----------- -------------------- -------------
        203 Susan                           40

Execution Plan
----------------------------------------------------------
Plan hash value: 2056577954

---------------------------------------------------------
| Id  | Operation                   | Name              |
---------------------------------------------------------
|   0 | SELECT STATEMENT            |                   |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |
|*  2 |   INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |
---------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DEPARTMENT_ID"=40)

Note
-----
   - rule based optimizer used (consider using cbo)
```

Figura 8.2: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img02code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

8.3 HINT ALL_ROWS

Seu objetivo é fazer com que haja vazão em retornar todas as linhas o mais rápido possível. Seu uso pode trazer ganhos quando estamos trabalhando com grandes retornos de dados, como relatórios em lotes onde o retorno de linhas é gigantesco. Já seu uso para um número pequeno de linhas ou até mesmo o retorno de uma única linha não é recomendado. Outra característica desse `hint` é que seu uso pode suprimir o uso de índices pelo otimizador, tendo prioridade nesses casos. Contudo, `hints` que forçam caminhos de acesso específicos têm prioridade sobre esse `hint`. Ele pode também ser definido através do parâmetro `OPTIMIZER_MODE`, no arquivo de parâmetros do Oracle, `init.ora`. Contudo, os `hints` em nível de consulta terão prioridade sobre a definição padrão configurada através deste arquivo.

```
SQL> select /*+ ALL_ROWS */ employee_id, first_name
  2  from   employees;

107 rows selected.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2832838249

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                  |   107 |  2675 |     2  (50)| 00:00:01 |
|   1 |  VIEW                  | index$_join$_001 |   107 |  2675 |     2  (50)| 00:00:01 |
|*  2 |   HASH JOIN            |                  |       |       |            |          |
|   3 |    INDEX FAST FULL SCAN| EMP_EMP_ID_PK    |   107 |  2675 |     0   (0)| 00:00:01 |
|   4 |    INDEX FAST FULL SCAN| EMP_NAME_IX      |   107 |  2675 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(ROWID=ROWID)
```

Figura 8.3: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img03code.txt

Nesse exemplo, não é possível ver características apenas olhando para o plano, pois seu objetivo é rapidez em trazer as primeiras linhas do resultado da instrução SQL, que, nesse caso, seriam o código do empregado e o primeiro nome dele.

8.4 HINT INDEX

Este `hint` força o uso de determinados índices que por ventura estejam disponíveis em tabelas que façam parte da instrução SQL. Em algumas situações, dependendo de como a instrução SQL foi escrita, ou pela quantidade de dados que ela retorna, conforme é mostrado no capítulo sobre índices, pode acontecer de o otimizador fazer escolhas por determinados índices de forma não muito adequada, ou até mesmo optar por não escolher tais índices para acessar os dados de uma ou mais tabelas. Nessa hora, a utilização deste `hint` pode ser uma saída. Forçando a utilização de um determinado índice, o otimizador vai usá-lo

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

baseando-se no melhor plano. No exemplo a seguir, estamos forçando o otimizador a utilizar o índice `EMP_DEPARTMENT_IX` da tabela `EMPLOYEES` ao buscar informações do empregado. Note que, no plano de execução, tal índice é acessado através do método de acesso `INDEX RANGE SCAN` .

```
SQL> select /*+ INDEX( employees EMP_DEPARTMENT_IX ) */
  2         employee_id, first_name, department_id
  3  from employees
  4  where department_id = 10;

EMPLOYEE_ID FIRST_NAME           DEPARTMENT_ID
----------- -------------------- -------------
        200 Jennifer                        10


Execution Plan
----------------------------------------------------------
Plan hash value: 2056577954

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                  |     1 |    14 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES        |     1 |    14 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DEPARTMENT_ID"=10)
```

Figura 8.4: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img04code.txt

Há também o `hint` `NO_INDEX` , com o qual você pode forçar o otimizador a não utilizar determinados índices que estariam disponíveis na instrução SQL. Nesse `hint` , você informa a tabela e os nomes dos índices que você quer que o otimizador ignore. Atente-se para conflitos entre utilizar `NO_INDEX` e `INDEX` na mesma instrução SQL. Caso estejam em conflito, o otimizador vai

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

ignorá-los. O `hint NO_INDEX` pode ser muito útil na análise da performance quando você precisar testar cenários forçando a não utilização de determinados índices em específico, fazendo com que o otimizador utilize outros índices disponíveis ou até mesmo nenhum deles.

8.5 HINT ORDERED

Sua função é especificar a ordem das tabelas na cláusula `from`. Utilizando este `hint`, é possível especificar que a primeira tabela informada na cláusula `from` será a tabela principal. Como vimos em capítulos anteriores, uma das tarefas do otimizador é escolher a ordem das tabelas em uma instrução SQL. Aplicando o `hint ORDERED`, poderemos influenciar essa escolha forçando-o a utilizar a ordem na qual colocamos as tabelas na cláusula `from`. O interessante desta sugestão é que ela pode ser usada tanto no modo baseado em custo como no modo baseado em regras. Nem todos os `hints` possuem essa flexibilidade. No exemplo a seguir, é possível ver que a primeira operação dentro do método de junção `NESTED LOOPS` é uma varredura por um índice da tabela `employees`, seguindo a ordem estabelecida na cláusula `from`, onde colocamos a tabela `employees` em primeiro, e a tabela `departments` na sequência.

```
SQL> select /*+ ORDERED */ employee_id, first_name,
  2         departments.department_name
  3  from employees, departments
  4  where employees.department_id = departments.department_id
  5  and departments.department_id = 80
  6  and employees.employee_id = 160;

EMPLOYEE_ID FIRST_NAME           DEPARTMENT_NAME
----------- -------------------- ------------------------------
        160 Louise               Sales
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2782876085

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    30 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |               |     1 |    30 |     2   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    14 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |     1 |    16 |     1   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN         | DEPT_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("EMPLOYEES"."DEPARTMENT_ID"=80)
   3 - access("EMPLOYEES"."EMPLOYEE_ID"=160)
   5 - access("DEPARTMENTS"."DEPARTMENT_ID"=80)
```

Figura 8.5: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img05code.txt

Se invertermos a ordem das tabelas, o otimizador utilizará outros meios de acesso referentes à nova ordenação imposta. Note que, no exemplo anterior, a tabela `employees` é a tabela principal, ou seja, a tabela controle. Já no exemplo a seguir, como invertemos a ordem, a tabela `departments` é a tabela principal agora.

```
SQL> select /*+ ORDERED */ employee_id, first_name,
  2         departments.department_name
  3  from departments, employees
  4  where employees.department_id = departments.department_id
  5  and departments.department_id = 80
  6  and employees.employee_id = 160;

EMPLOYEE_ID FIRST_NAME           DEPARTMENT_NAME
----------- -------------------- ------------------------------
        160 Louise               Sales
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 4225575861

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    30 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |               |     1 |    30 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |     1 |    16 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    14 |     1   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENTS"."DEPARTMENT_ID"=80)
   4 - filter("EMPLOYEES"."DEPARTMENT_ID"=80)
   5 - access("EMPLOYEES"."EMPLOYEE_ID"=160)
```

Figura 8.6: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img06code.txt

Seguindo a mesma lógica do exemplo anterior, veja que a primeira operação dentro de `NESTED LOOPS` foi um acesso por índice através da tabela `departments`.

8.6 HINT PARALLEL

Quando um `FULL TABLE SCAN` estiver acontecendo em uma instrução SQL, por exemplo, este `hint` permite determinar que esta instrução seja dividida em partes e que cada parte seja processada em um processo diferente, isto é, que ela seja executada de forma paralela em várias execuções. A execução paralela pode economizar tempo de execução, permitindo que vários processamentos sejam executados em paralelo em vez de um por vez.

Para trabalhar com instruções paralelas, a versão do banco de

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

dados precisa ter configurada a opção de particionamento (`Partitioning Option`). Esta é uma Option de banco de dados que geralmente está disponível em versões mais completas do banco de dados, como na versão `Enterprise Edition (EE)` . A partir do Oracle 9i, por exemplo, este `hint` pode ser também usado em declarações `insert` , `update` e `delete` . Então, para utilizar esta Option é necessário ver se o banco de dados com o qual você está trabalhando suporta esse recurso.

> **NOTA:** A versão `Enterprise Edition` fornece desempenho, disponibilidade, escalabilidade e segurança necessária para aplicações de missão crítica, tais como aplicações transacionais (OLTP), `data warehouses` e aplicações de internet de alta demanda. O `Oracle Database Enterprise Edition` contém todos os componentes do Oracle Database e pode ser melhorado ainda mais com a aquisição de novas opções e pacotes específicos para sua necessidade.

Além de especificar que a instrução será executada de forma paralela, também é possível definir um grau de paralelismo que será aplicado para cada operação em uma instrução SQL. Esse grau será usado para definir quantos processos serão utilizados para satisfazer tal execução.

```
SQL> select /*+ PARALLEL (employees) */ employee_id, salary
  2  from employees
  3  order by salary;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 879556509

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name      |Rows|Bytes| Cost(%CPU)| Time     | TQ   |IN-OUT|PQ Distrib|
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |           | 107| 2782|     3 (34)| 00:00:01|      |      |          |
|   1 |  PX COORDINATOR          |           |    |     |           |         |      |      |          |
|   2 |   PX SEND QC (ORDER)     | :TQ10001  | 107| 2782|     3 (34)| 00:00:01| Q1,01| P->S | QC(ORDER)|
|   3 |    SORT ORDER BY         |           | 107| 2782|     3 (34)| 00:00:01| Q1,01| PCWP |          |
|   4 |     PX RECEIVE           |           | 107| 2782|     2  (0)| 00:00:01| Q1,01| PCWP |          |
|   5 |      PX SEND RANGE       | :TQ10000  | 107| 2782|     2  (0)| 00:00:01| Q1,00| P->P | RANGE    |
|   6 |       PX BLOCK ITERATOR  |           | 107| 2782|     2  (0)| 00:00:01| Q1,00| PCWC |          |
|   7 |        TABLE ACCESS FULL| EMPLOYEES | 107| 2782|     2  (0)| 00:00:01| Q1,00| PCWP |          |
-----------------------------------------------------------------------------------------------------------

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Figura 8.7: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img07code.txt

Aqui é possível identificar execuções em paralelo através das evidências de operações `PX`, mostradas no plano. Nesse exemplo, conforme pode ser visto em nota, o otimizador trabalhou com dois níveis de paralelismo (`level=2`).

Este outro exemplo mostra como seria uma instrução SQL utilizando execução paralela, mas agora especificando o grau de paralelismo. Através do `hint` inserido na instrução SQL, sugerimos quatro processos em paralelo. Veja como fica a declaração do `hint`:

```
SQL> select /*+ PARALLEL (employees,4) */ employee_id, salary
  2  from employees
  3  order by salary;
```

Para anular um paralelismo que tenha sido previamente definido para uma determinada tabela, você poderá utilizar o `hint NOPARALLEL`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

8.7 HINT FIRST_ROWS

Geralmente força o uso de índices na busca dos dados. Utilizando esse `hint`, o otimizador trabalhará baseando-se na busca mais rápida da primeira linha do resultado, ou de N linhas caso seja especificado. Isso é bastante conveniente quando se deseja fazer uma consulta em que o retorno esperado é apenas um registro. Entretanto, na recuperação de várias linhas, o processo pode se tornar mais lento. Ele não é aplicado em `update` ou `delete` e destina-se até mesmo para tabelas em que as estatísticas não foram geradas, ou no modo baseado em custo (CBO). Esse `hint` pode também ser definido através do parâmetro `OPTIMIZER_MODE`, no arquivo de parâmetros do Oracle (`init.ora`).

```
SQL> select /*+ FIRST_ROWS */ first_name, hire_date
  2  from employees
  3  where employee_id = 190;

Execution Plan
----------------------------------------------------------
Plan hash value: 1833546154

--------------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |    34 |     0   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    34 |     0   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID"=190)
```

Figura 8.8: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img08code.txt

A coluna de custo (`Cost (%CPU)`) não mostra nenhuma

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

informação quando utilizamos esse tipo de `hint`. Agora um exemplo especificando o número de linhas para o `hint` `FIRST_ROWS`:

```
SQL> select /*+ FIRST_ROWS(10) */ first_name, hire_date
  2  from employees
  3  where employee_id = 70;

no rows selected

Execution Plan
----------------------------------------------------------
Plan hash value: 1833546154

---------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |    34 |     0   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    34 |     0   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
---------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID"=70)
```

Figura 8.9: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img09code.txt

O objetivo aqui é que o otimizador possa trazer as dez linhas do resultado, com mais rapidez possível.

8.8 HINT FULL

Este `hint` força uma varredura completa da tabela. Ele é utilizado quando se quer que o otimizador ignore qualquer índice que possa existir e que esteja disponível para determinada tabela em uma instrução SQL. Dependendo dos critérios que você utilizar na instrução SQL para filtrar os dados, poderá ocasionar que os

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

registros de uma determinada tabela da instrução sejam poucos e o acesso pelo índice seja mais lento do que se o otimizador fizesse um acesso completo à tabela. Para esses casos, mesmo que o otimizador insista em usar índices, o uso do `hint FULL` vai forçá-lo a não os utilizar. Outra curiosidade sobre este `hint` é o fato de que sua aplicação pode mudar a execução da ordem das tabelas declaradas na cláusula `from`, podendo usar uma tabela principal diferente.

```
SQL> select /*+ FULL(employees) */ employee_id, first_name
  2  from employees
  3  where employee_id = 60;

no rows selected

Execution Plan
----------------------------------------------------------
Plan hash value: 1445457117

---------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     4 |   100 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMPLOYEES |     4 |   100 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("EMPLOYEE_ID"=60)
```

Figura 8.10: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img10code.txt

Veja que estamos forçando o otimizador a fazer um acesso completo na tabela `employees` (`TABLE ACCESS FULL`), ignorando qualquer chance de uso de índice, que por ventura estaria elegível para a instrução SQL.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

8.9 HINT LEADING

Seu objetivo é especificar apenas a primeira tabela principal (tabela controle). Quando se trabalha com consultas complexas, pode ser interessante querer definir qual tabela deve ser acessada primeiro. Em geral, analisando os objetos, podemos até prever qual tabela será acessada primeiro, mas não é certo que o otimizador vai escolhê-la. Com o uso deste `hint`, é possível especificar uma tabela que será a responsável por controlar a consulta (`Driving Table`), deixando que o otimizador descubra qual usar depois dela. O `hint ORDERED`, que vimos anteriormente, é semelhante ao `LEADING`, sendo que o `LEADING` é cancelado pelo `hint ORDERED` caso ele exista na instrução SQL. Uma observação importante no caso do `LEADING` é que poderá ser especificada apenas uma tabela para este `hint`. Caso mais tabelas forem especificadas, elas serão ignoradas pelo otimizador.

```
SQL> select /*+ LEADING(e) */
  2         e.employee_id
  3        ,e.first_name
  4        ,d.department_name
  5        ,h.job_id
  6  from   employees    e
  7        ,departments  d
  8        ,job_history  h
  9  where  e.department_id = d.department_id
 10  and    e.employee_id   = h.employee_id
 11  and    d.department_id = 10;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3796347601

-----------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                         |     1 |    88 |     3   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN          |                         |     1 |    88 |     3   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                 |                         |       |       |            |          |
|   3 |    NESTED LOOPS                |                         |     1 |    58 |     3   (0)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES               |     1 |    38 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN          | EMP_DEPARTMENT_IX       |     1 |       |     1   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN           | JHIST_EMP_ID_ST_DATE_PK |     1 |       |     1   (0)| 00:00:01 |
|   7 |    TABLE ACCESS BY INDEX ROWID | JOB_HISTORY             |     1 |    20 |     1   (0)| 00:00:01 |
|   8 |   BUFFER SORT                  |                         |     1 |    30 |     2   (0)| 00:00:01 |
|   9 |    TABLE ACCESS BY INDEX ROWID | DEPARTMENTS             |     1 |    30 |     0   (0)| 00:00:01 |
|* 10 |     INDEX UNIQUE SCAN          | DEPT_ID_PK              |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("E"."DEPARTMENT_ID"=10)
   6 - access("E"."EMPLOYEE_ID"="H"."EMPLOYEE_ID")
  10 - access("D"."DEPARTMENT_ID"=10)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Figura 8.11: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img11code.txt

Na linha 4 do plano de execução, a primeira tabela a ser acessada, ou seja, a tabela controle, foi a tabela `employees`. Como na instrução SQL a tabela utiliza o alias `'e'`, foi por este alias que aplicamos o `hint`.

Utilizando o mesmo exemplo, vejamos como ficou o plano de execução ao forçarmos para que a tabela `job_history` fosse a tabela principal.

```
SQL> select /*+ LEADING(h) */
  2         e.employee_id
  3        ,e.first_name
  4        ,d.department_name
  5        ,h.job_id
  6  from   employees    e
  7        ,departments  d
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  8         ,job_history  h
  9  where   e.department_id = d.department_id
 10  and     e.employee_id   = h.employee_id
 11  and     d.department_id = 10;

Execution Plan
----------------------------------------------------------
Plan hash value: 1295177975

-------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                          |     1 |    88 |     3  (34)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN          |                          |     1 |    88 |     3  (34)| 00:00:01 |
|   2 |   NESTED LOOPS                 |                          |       |       |            |          |
|   3 |    NESTED LOOPS                |                          |     1 |    58 |     3  (34)| 00:00:01 |
|   4 |     VIEW                       | index$_join$_003         |    10 |   200 |     3  (34)| 00:00:01 |
|*  5 |      HASH JOIN                 |                          |       |       |            |          |
|   6 |       INDEX FAST FULL SCAN     | JHIST_EMP_ID_ST_DATE_PK  |    10 |   200 |     1   (0)| 00:00:01 |
|   7 |       INDEX FAST FULL SCAN     | JHIST_JOB_IX             |    10 |   200 |     1   (0)| 00:00:01 |
|*  8 |     INDEX UNIQUE SCAN          | EMP_EMP_ID_PK            |     1 |       |     0   (0)| 00:00:01 |
|*  9 |    TABLE ACCESS BY INDEX ROWID| EMPLOYEES                |     1 |    38 |     0   (0)| 00:00:01 |
|  10 |   BUFFER SORT                  |                          |     1 |    30 |     3  (34)| 00:00:01 |
|  11 |    TABLE ACCESS BY INDEX ROWID| DEPARTMENTS              |     1 |    30 |     0   (0)| 00:00:01 |
|* 12 |     INDEX UNIQUE SCAN          | DEPT_ID_PK               |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access(ROWID=ROWID)
   8 - access("E"."EMPLOYEE_ID"="H"."EMPLOYEE_ID")
   9 - filter("E"."DEPARTMENT_ID"=10)
  12 - access("D"."DEPARTMENT_ID"=10)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Figura 8.12: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img12code.txt

Nesse exemplo, o otimizador acabou criando uma `view` em memória (chamada `INDEX$_JOIN$_003`) para apoiá-lo na sugestão que lhe demos ao definirmos que a tabela `job_history` fosse a tabela principal, ou seja, a tabela controle. Esse é um exemplo que mostra como o otimizador pode modificar a instrução SQL em tempo de execução para atender às necessidades de busca dos dados, sempre tentando utilizar a melhor forma para isto.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

8.10 HINT CACHE

Este `hint` é utilizado para que seja possível carregar toda a tabela em memória. Assim, se outras instruções SQL utilizassem a mesma tabela, não seria necessário realizar acesso ao disco, pois ela já estaria em memória. Embora pareça óbvio que mantê-las em memória é a melhor coisa a ser feita, isso pode se tornar um problema. Uma quantidade enorme de dados sendo carregada em memória pode sobrecarregar esta área. No entanto, este recurso pode ser uma opção quando utilizado para tabelas pequenas.

```
SQL> select /* CACHE(regions) */ * from regions;

Execution Plan
----------------------------------------------------------
Plan hash value: 3077898360

---------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |     4 |    56 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| REGIONS |     4 |    56 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

Estatísticas
----------------------------------------------------------
         66  recursive calls
          0  db block gets
         63  consistent gets
          2  physical reads
          0  redo size
        731  bytes sent via SQL*Net to client
        523  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
          4  rows processed
```

Figura 8.13: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img13code.txt

Nesse exemplo, estamos aplicando o `hint` sobre a tabela

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

regions . Se olharmos para as informações mostradas no bloco "Estatísticas" deste exemplo, é possível ver que o maior processamento ocorreu em memória (buffer cache). Isso pode ser visto através do número de Consistent Gets , que neste caso, foi 63 leituras.

> **NOTA:** Consistent Gets é o número de vezes que uma leitura consistente foi requisitada para um bloco do buffer cache .

Caso uma tabela esteja especificada, em nível de banco de dados, para ser carregada em memória ao ser acessada, e você queira anular isso, pode ser utilizado o hint NOCACHE . Assim ela não será colocada em memória quando você for acessá-la.

8.11 HINT USE_NL

Este hint é utilizado com o intuito de se obter, com rapidez, o retorno de uma única linha. Ele é ideal para quando estamos trabalhando com tabelas indexadas. A instrução SQL passa a ser processada usando loops aninhados (NL = NESTED LOOPS), tomando a primeira linha de uma tabela, com base no resultado de outra tabela. Este processo é o oposto de um MERGE JOIN , o que levaria mais tempo para adquirir a primeira linha. Com sua aplicação, há um maior ganho de tempo de resposta. Para o retorno de uma grande quantidade de linhas, esse processo pode se tornar lento, contudo seu uso aliado a bons índices pode trazer ganhos muito satisfatórios para a performance.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select /*+ ORDERED USE_NL(e) */
  2         e.employee_id
  3        ,e.first_name
  4        ,d.department_name
  5  from   employees   e
  6        ,departments d
  7  where  e.department_id = d.department_id
  8  and    d.department_id = 30
  9  and    e.employee_id   = 116;

Execution Plan
----------------------------------------------------------
Plan hash value: 2782876085

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    68 |     0   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |               |     1 |    68 |     0   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    38 |     0   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |     1 |    30 |     0   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN         | DEPT_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("E"."DEPARTMENT_ID"=30)
   3 - access("E"."EMPLOYEE_ID"=116)
   5 - access("D"."DEPARTMENT_ID"=30)
```

Figura 8.14: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img14code.txt

Nesse exemplo, são recuperadas as linhas resultantes da tabela `employees` e, em seguida, essas linhas são processadas com as linhas combinadas da tabela `departments`, que é a tabela especificada no `NESTED LOOPS` através do `hint` `USE_NL`.

8.12 HINT USE_HASH

Normalmente, este será o modo mais rápido para unir muitas linhas de várias tabelas, pois força o uso da junção `HASH`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Entretanto, deve-se ter em mente que essa operação envolverá um uso considerável de memória. Seu processo é semelhante ao do uso da junção `NESTED LOOPS`, em que o resultado de uma tabela é percorrido através do resultado de outra tabela da junção. A diferença é que, entre duas tabelas, a segunda tabela, que será varrida, estará sendo colocada primeiro, em memória. A vantagem deste tipo de varredura é a rapidez na recuperação dessas informações, pois à medida que as fontes forem sendo processadas, os resultados já estarão em memória. Sendo assim, o retorno dos dados não precisará esperar pelo término do processo para começar a receber as linhas encontradas.

```
SQL> select /*+ USE_HASH (d) */
  2         e.employee_id
  3        ,e.first_name
  4        ,d.department_name
  5  from   employees   e
  6        ,departments d
  7  where  e.department_id = d.department_id
  8  and    e.employee_id   = 100;

Execution Plan
----------------------------------------------------------
Plan hash value: 861361734

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    68 |     4  (25)| 00:00:01 |
|*  1 |  HASH JOIN                   |               |     1 |    68 |     4  (25)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    38 |     0   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL          | DEPARTMENTS   |   409 | 12270 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   3 - access("E"."EMPLOYEE_ID"=100)
```

Figura 8.15: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img15code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Na linha Id 1 do plano de execução, é possível ver o uso do método de junção `HASH JOIN` pelo otimizador, conforme especificado no `hint`. Nesse exemplo, a tabela que será colocada em memória será a tabela `departments`, indicada pelo alias `d`.

8.13 HINT USE_MERGE

Este `hint` é utilizado para forçar o método de junção `SORT-MERGE`, com o qual duas fontes de dados são classificadas e mescladas para gerar um resultado final. Esse método, como vimos em capítulos anteriores, é melhor empregado quando precisamos varrer todas as linhas de uma determinada tabela, por exemplo. No caso de haver outras fontes que não estejam especificadas no `hint`, as linhas resultantes delas serão unidas com o resultado final da junção `MERGE`.

```
SQL> select /*+ USE_MERGE(e d) */ *
  2  from   employees   e
  3        ,departments d
  4  where  e.department_id = d.department_id;

106 rows selected.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1343509718

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |   106 |  9540 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |             |   106 |  9540 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |    27 |   567 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | DEPT_ID_PK  |    27 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |             |   107 |  7383 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | EMPLOYEES   |   107 |  7383 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Figura 8.16: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap08img16code.txt

Na linha Id 1 do plano de execução, é possível ver o uso do método de junção `MERGE JOIN` pelo otimizador, conforme especificado no `hint`. Nesse exemplo, as tabelas envolvidas foram a `employees` e `departments`, indicadas pelo aliases `e` e `d`, respectivamente.

8.14 CONSIDERAÇÕES SOBRE O USO DE HINTS

A sintaxe do uso de `hints` deve ser seguida à risca para que o otimizador possa considerá-lo. Mediante essa premissa, deve-se ter atenção para algumas situações que podem ocorrer durante sua declaração, pelas caraterísticas dos objetos envolvidos na instrução SQL, ou ainda, por determinadas configurações do próprio banco de dados, inviabilizando seu uso pelo otimizador. Algumas destas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

situações são:

- A sintaxe do `hint` estar incorreta;
- Haver uma ou várias tabelas não analisadas;
- Haver conflitos entre `hints` ;
- O modo de otimização não estar configurado para um tipo compatível com o `hint` ;
- O `hint` precisar de algum parâmetro de inicialização para que funcione;
- Utilizar o nome da tabela em vez do nome do alias especificado;
- Haver incompatibilidade entre o `hint` e a versão do banco de dados;
- Existir algum bug no software de banco de dados.

O uso dos `hints` proporciona um sistema de otimização maleável no sentido de que, dependendo da necessidade encontrada, é possível determinar como o otimizador vai trabalhar e qual o caminho mais adequado deve percorrer para executar suas instruções SQL. Portanto, seu uso, quando bem alinhado e afinado, poderá trazer ganhos de performance, sendo tanto para soluções rápidas e paliativas, quanto para soluções permanentes, embora este último não seja o uso mais adequado.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 9

PLANOS DE EXECUÇÃO (EXPLAIN PLAN)

Um plano de execução é uma sequência de passos que mostra qual o roteiro que o otimizador de consultas tomará para acessar as informações em um banco de dados, dada uma instrução SQL. Para cada tabela envolvida na instrução SQL, haverá, por exemplo, um método de acesso para a recuperação dos dados. Se a instrução SQL possuir mais de uma tabela, haverá métodos de junção para relacioná-las. Sendo assim, essa sequência de passos que determina como o otimizador chegará até os dados propriamente ditos, resultantes da execução da instrução SQL, chama-se plano de execução.

O plano de execução é gerado pelo próprio otimizador que, através de análises de estatísticas ou de regras, determina qual o plano mais adequado para a execução de uma instrução SQL. O otimizador é capaz de gerar vários planos, até achar aquele que ele julgar ser o mais adequado. Através da leitura e do entendimento das informações mostradas no plano de execução, é possível conhecer vários aspectos relacionados ao comportamento e à maneira como o otimizador toma decisões. Mas qual o intuito de conhecer o plano de execução de uma instrução SQL, ou saber como o otimizador age perante a execução de tal instrução? Pois

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

bem, entender os planos de execução será de grande valia no monitoramento e ajuste da performance e na solução de problemas, pois, sabendo interpretá-los, será possível determinar se o otimizador escolheu um plano adequado e até mesmo sugerir modificações neste plano. Em um plano de execução, podem ser mostradas diversas informações importantes, que podem nos dar um raio-x da nossa instrução SQL, de como os objetos são vistos pelo otimizador e de como chegar ao resultado que vemos. Através do plano de execução é possível visualizar, por exemplo:

- A ordenação das tabelas referenciadas pela instrução SQL;
- O método de acesso para cada tabela mencionada na instrução SQL;
- O método de junção utilizado para o relacionamento entre as tabelas da instrução SQL;
- Operações de dados, como filtros, ordenações ou agrupamentos;
- Dados referentes à otimização, tal como o custo e a cardinalidade de cada operação;
- Informações sobre particionamento, tal como o conjunto de partições acessadas;
- Informações sobre execuções paralelas, tal como o método de distribuição de junção de saídas.

De fato são muitas as informações que podem ser extraídas dos planos de execução. Contudo, é preciso aprender sobre essas informações e suas características, de modo que possam ser usadas para entender cenários, corrigir problemas de desempenho, buscar o melhoramento da performance e atingir os objetivos esperados.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

9.1 INTERPRETANDO PLANOS DE EXECUÇÃO

Para saber ler e interpretar um plano de execução é necessário saber a sequência de como fazer a leitura. Ler um plano de execução não é complexo, mas pode levar algum tempo, dependendo do tamanho do plano.

Não se assuste se em um primeiro momento o plano de execução parecer uma sopa de letrinhas sem sentido. Na verdade, acredito que, neste ponto do livro, se você leu os capítulos anteriores, você vai identificar muitas das operações que aparecem no plano. Depois que você se familiariza, fica mais fácil. Geralmente, junto das operações, o otimizador dá os tempos que levou para executá-las, informações sobre o custo, o número de execuções e quantidade de linhas retornadas. Uma dica é dar uma boa visualizada no plano de uma forma mais ampla, tentando localizar onde os maiores tempos ou custos aparecem, caso você esteja procurando um problema de performance. Mas vamos lá. A ideia aqui é demonstrar como fazer a interpretação e conhecer um pouco da estrutura de um plano de execução.

Cada operação do plano de execução, ou recupera linhas de tabelas ou índices, ou recebe linhas vindas a partir de uma ou mais operações antecessoras. Este conjunto de linhas é conhecido como `Row Source`. Cada operação executada devolve seu resultado para uma operação acima dela.

As informações de um plano de execução são mostradas de forma encadeada. Cada entrada no plano de execução é uma ação de processamento realizada pelo otimizador. Os planos de

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

execução devem ser lidos de dentro para fora, ou da operação mais interna para a operação mais externa. A operação mais interna é a operação que ocorreu primeiro, e a operação mais externa a que ocorreu depois. Cada linha do plano de execução possui um identificador numérico, visto no plano através da coluna Id (coluna da tabela `plan_table`).

Veja o exemplo a seguir, mostrando um plano de execução, e como faríamos a leitura e interpretação dele:

```
Plan hash value: 4225575861
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     1 |    30 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |               |     1 |    30 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |     1 |    16 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK    |     1 |       |     0   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    14 |     1   (0)| 00:00:01 |
|*  5 |   INDEX UNIQUE SCAN          | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

NÍVEL MAIS INTERNO (operação mais interna)

NÍVEL MAIS EXTERNO (operação mais externa)

IDs das linhas do Plano de Execução

Figura 9.1: Exemplo 1: leitura e interpretação de um plano de execução.

Observando o plano de execução, podemos notar que, visualmente, a operação de Id 5 é a operação mais interna, e a de Id 4 é a mais externa, então, nessa estrutura encadeada, podemos dizer que a operação 5 é filha da operação 4. É exatamente dessa forma então que devemos ler este plano de execução: começando de dentro para fora, da operação mais interna para a externa. Via de regra, a operação mais interna indica a primeira operação que foi realizada.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Você vai perceber que em uma estrutura encadeada, como em um plano de execução, pode haver muitas operações que estão em um mesmo nível. Você deve ir resolvendo cada nível de dentro para fora, almejando sempre subir para um nível mais acima (ou mais externo). Então primeiro serão lidos os níveis mais internos, os que estejam no mesmo nível, e depois os níveis mais externos.

O que é preciso ter em mente é que o resultado de uma determinada operação sempre será utilizado no processamento de outra operação mais externa, até chegarmos ao final do processamento de todas as operações do plano de execução, ou seja, ao resultado final do processamento da instrução SQL. Cada vez que uma operação é executada, os dados propriamente ditos são transformados ou eliminados, sendo pelas operações de comparação, pelos filtros ou por funções aplicadas nas colunas das tabelas que fazem parte da instrução SQL, ou ainda, pelas junções desses objetos.

O tamanho do plano de execução, ou o quão dispendiosa pode se tornar sua leitura, também dependerá da quantidade de métodos de junção ou métodos de acesso contidos nele, bem como outros aspectos relacionados à quantidade de tabelas envolvidas, ao uso de subconsultas, funções e outras características da instrução. Dependerá da forma como a instrução SQL foi escrita em termos de complexidade, não só de tamanho, mas de lógica idealizada.

No entanto, para ler um plano de execução, é preciso apenas decompô-lo em blocos básicos (pequenos trechos) e saber reconhecer a ordem de execução com base nas regras existentes para cada tipo de operação e como se relacionam. Conhecendo as

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

regras e como elas estão relacionadas, você conseguirá ler e interpretar um plano de execução, independente do seu tamanho. Por isso, é válido ressaltar que a dica é sempre analisar o plano de execução por pequenos trechos, pequenos blocos, que uma vez identificados, poderão ser lidos mais facilmente e, assim, você vai percorrendo todo o plano de execução gradualmente.

Para cada operação filha de cada operação combinada, um bloco é definido e, portanto, você pode isolá-lo e interpretá-lo, partindo para o próximo bloco. Veja um exemplo a seguir como podemos identificar estes blocos, tornando mais organizada a visualização do plano.

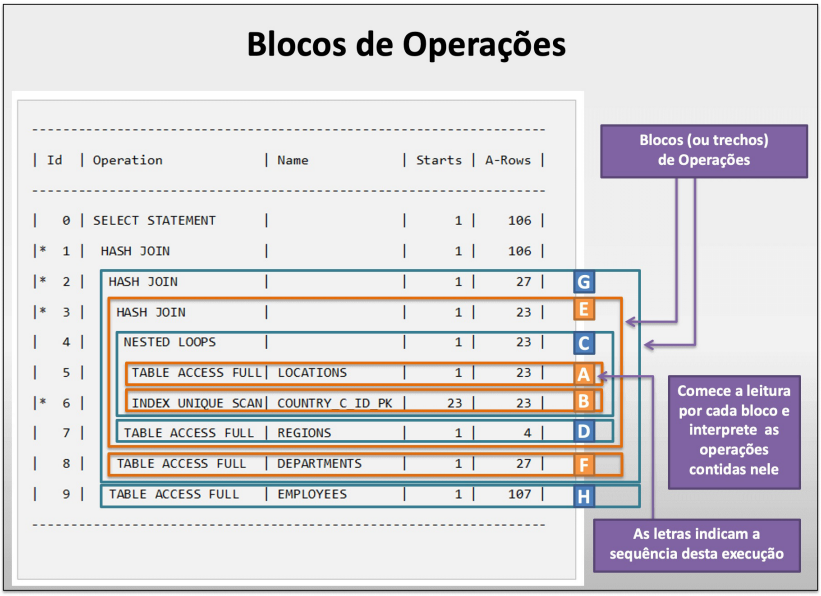

Figura 9.2: Exemplo 2: leitura e interpretação de um plano de execução.

Conforme pode ser visto no exemplo, os blocos estão indicados

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

pelos retângulos presentes na imagem, por exemplo, as operações `A`, `B` e `C` estão dentro de um bloco. O resultado das operações, juntamente com a operação `E`, formam o próximo bloco, e assim por diante, até que percorramos todo o plano.

No primeiro bloco, o otimizador resolve a operação `A` e faz uma varredura completa na tabela `locations`. Depois disso, resolve a operação `B`, acessando os dados da tabela `country` através do seu índice `country_c_id_pk`. Com isso, ele combina o resultado dessas duas operações utilizando o método `NESTED LOOPS`, como pode ser identificado pela letra `C`. Desta forma, finaliza-se a leitura do primeiro bloco.

O resultado deste primeiro bloco será combinado com o resultado da operação `D`, através do método `HASH JOIN`, combinação representada através da operação `E`. O resultado do segundo bloco então deriva do resultado do primeiro bloco, do resultado da operação da `C`, combinada com a operação `D`, na qual o otimizador realizou uma operação de varredura completa na tabela `regions`. Assim, são finalizadas as operações do segundo bloco, com a operação `E`. Se olharmos mais externamente, veremos que seguindo a mesma lógica existirão outros blocos a serem resolvidos, até chegarmos ao final do plano de execução.

Um dos motivos da análise do plano de execução será para identificar problemas de performance. Se for este o seu objetivo, é mais um motivo para sempre isolar pequenos trechos do plano de execução, buscando por partes em que exista um custo (`Cost %CPU`) ou tempo de execução (`Time`) mais elevado. Geralmente, essas duas informações têm relação uma com a outra. Nesses casos,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

tente identificar as tabelas e relacionamentos que se referem a estes pontos no plano de execução e tente descobrir os porquês da baixa de desempenho. Veja o exemplo a seguir:

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3454436918

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                  |   106 |  9010 |    10  (20)| 00:00:01 |
|*  1 |  HASH JOIN             |                  |   106 |  9010 |    10  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL    | REGIONS          |     4 |    56 |     2   (0)| 00:00:01 |
|   3 |   NESTED LOOPS         |                  |   106 |  7526 |     7  (15)| 00:00:01 |
|*  4 |    HASH JOIN           |                  |   106 |  5936 |     7  (15)| 00:00:01 |
|*  5 |     HASH JOIN          |                  |   106 |  3074 |     5  (20)| 00:00:01 |
|   6 |      TABLE ACCESS FULL| EMPLOYEES        |   107 |  1070 |     2   (0)| 00:00:01 |
|   7 |      TABLE ACCESS FULL| DEPARTMENTS      |    27 |   513 |     2   (0)| 00:00:01 |
|   8 |     TABLE ACCESS FULL | LOCATIONS        |    23 |   621 |     2   (0)| 00:00:01 |
|*  9 |    INDEX UNIQUE SCAN   | COUNTRY_C_ID_PK  |     1 |    15 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 9.3: Custo elevado no plano de execução.

Olhando para esse exemplo, vamos supor que detectássemos que a operação 4 do plano estivesse com um custo muito elevado (coluna `COST (%CPU)`). No exemplo, seu valor está como 7, indicando o custo total de todas as operações contidas em um nível mais interno, inclusive a própria operação. Com isso, poderíamos isolar o bloco correspondente às operações abaixo da operação 4, no caso, as operações 6, 7, 8, 5, incluindo a 4. Agora, sabemos que o resultado da operação 4 vem do processamento das tabelas `employees`, `departments` e `locations`. Olhando para o plano, também é possível detectar os métodos de acesso e de junção que foram utilizados.

Poderíamos trabalhar neste bloco específico a fim de tentar melhorá-lo, talvez verificando a existência ou não de índices nas tabelas envolvidas e que poderiam ser utilizados, por exemplo. Talvez melhorar o filtro destes dados analisando os critérios da

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

cláusula `where` , ou seja, analisar todos os pontos envolvidos neste trecho do plano de execução. Após as alterações, verifique outros pontos que por ventura estejam ainda com o custo elevado. É muito importante que seja gerado um novo plano após as alterações na instrução SQL, pois as alterações realizadas para um bloco de operações, mesmo específico, podem influenciar em todo o plano de execução.

Outra forma de visualizar a estrutura de plano de execução, no sentido de melhor entendê-la, utilizada inclusive por alguns desenvolvedores para demonstrá-la de forma gráfica, é através de diagramas. Neles, é mostrada uma visão de cada operação do plano como sendo um nó, fornecendo outra perspectiva de como podemos interpretar estas leituras.

A seguir, um exemplo de como o plano de execução é representado no formato de diagrama. Os números dentro dos círculos correspondem aos IDs de cada operação no plano de execução. Através dessa representação fica mais legível e claro o tamanho do plano, suas ramificações e é mais fácil identificar pequenos trechos (grupos) de operações. Mas lembre-se, o diagrama não nos diz exatamente a sequência de execução do plano de execução, apenas nos mostra como ele está estruturado. O que vai nos mostrar o direcionamento de como fazer a leitura será conhecer os tipos de operações que compõem o plano e como elas se relacionam, conforme é visto mais à frente.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

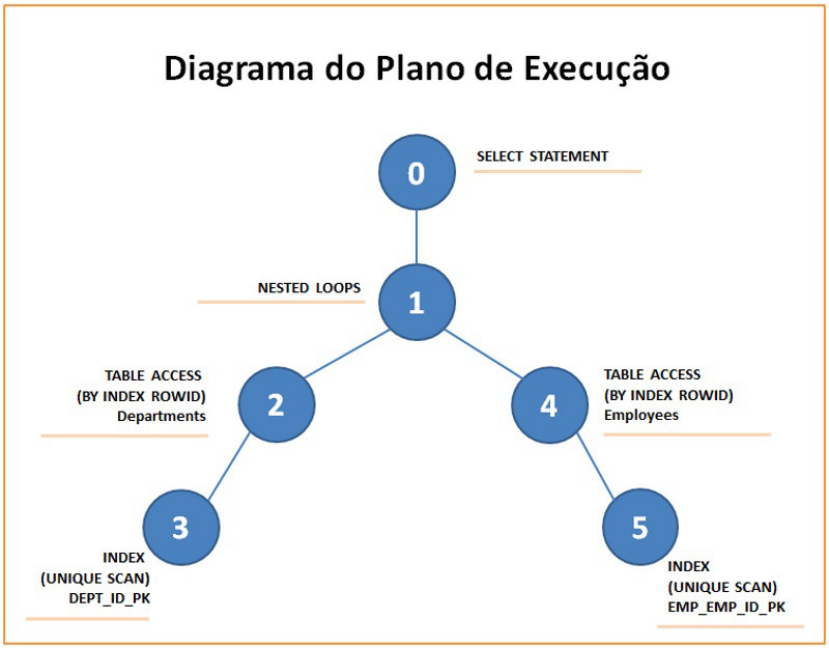

Figura 9.4: Diagrama do Plano de Execução.

Em uma interpretação bem sucinta desta estrutura, podemos dizer que as operações `TABLE ACCESS` (Id 2) e (Id 4) aceitam os `rowids` obtidos pelos `SCANS INDEX` (Id 3) e (Id 5). Isso resulta em um conjunto de `rowids` que permitem que as operações `TABLE ACCESS` (Id 2) da tabela `departments` e `TABLE ACCESS` (Id 4) da tabela `employees` retornem suas respectivas linhas para a sua operação `NESTED LOOPS` (Id 1), que, por sua vez, retorna as linhas que satisfazem a instrução SQL (Id 0).

Agora que já falamos um pouco sobre os planos de execução e suas características, vamos entrar nos detalhes mostrando as regras de como devemos proceder na leitura e interpretação, levando em consideração a forma de relacionamento entre as operações e quais

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

tipos de operações e comportamentos podem ser vistos.

Via de regra, o que vai nortear a leitura e permitir a interpretação correta do plano de execução será conhecer quais tipos de operação existem, seus funcionamentos e como se relacionam entre si. Em um plano de execução, podem existir várias formas de ler e de interpretar uma operação, dependendo do seu tipo e da situação como se encontra no plano. São essas regras e características que veremos agora e que vão lhe dar o conhecimento para interpretar a maioria dos planos de execução que comumente vemos no dia a dia. Antes de começar, recomendo que você já tenha lido os capítulos referentes ao métodos de junção de tabelas e métodos de acesso. Esse conhecimento vai lhe ajudar a melhor compreender o funcionamento das operações.

Relacionamento `Parent-Child`

A lógica que rege o funcionamento desses relacionamentos obedece a critérios e regras de como as operações foram ou serão executadas pelo otimizador. Ela é uma representação de uma série de passos que aconteceram até se alcançar os resultados. Esses passos, ou melhor, essas operações, estão relacionadas entre si e, na maioria das vezes, são dependentes umas das outras. Além de conhecer a lógica de como as operações são executadas, para ler e interpretar um plano de execução é necessário entender como elas se relacionam. Aprendemos em outros capítulos a função de cada operação e como elas funcionam. Agora vamos entender melhor a forma como elas se relacionam e como isso pode ser visualizado no plano de execução. O tipo de relacionamento entre operações podemos chamar de `Parent-Child` .

Sabendo disso, você conseguirá entender como o otimizador

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

está fazendo seu trabalho e, com esta análise, decidir ou não influenciar nesta condução. Perceba que entender esses relacionamentos é de suma importância e até mesmo o primeiro passo para se conseguir interpretar um plano de execução. A seguir, estão listadas as características e regras que definem como são e como devemos interpretar os relacionamentos entre as operações em um plano de execução.

Características do relacionamento `Parent-Child` :

- Um plano de execução é apresentado como uma estrutura em árvore;
- Cada nó desta árvore é referenciado por uma operação no plano de execução;
- Entre as operações (chamadas aqui de nós) existe uma relação pai e filho, chamada de `Parent-Child` .

Regras do relacionamento `Parent-Child` :

- Um nó pai possui um ou vários nós filhos;
- Um nó filho possui um único nó pai;
- A única operação sem um nó pai é a operação raiz da árvore;
- Quando um plano de execução é exibido, seus nós filhos aparecem recuados em relação ao seu nó pai;
- Um nó pai aparece antes de seus nós filhos (IDs dos nós pais são menores do que os IDs dos nós filhos).

Veja a seguir um exemplo de plano de execução onde podemos visualizar os relacionamentos entre nós pais e nós filhos. Para melhor entendimento, segue o diagrama em forma de árvore:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
-----------------------------------------------------------------------------
| Id  | Operation                    | Name              | Starts | A-Rows |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |      1 |      0 |
|   1 |  NESTED LOOPS                |                   |      1 |      0 |
|   2 |   NESTED LOOPS               |                   |      1 |      0 |
|*  3 |    TABLE ACCESS FULL         | DEPARTMENTS       |      1 |      1 |
|*  4 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |      1 |      0 |
|*  5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |      0 |      0 |
|   6 |    SORT AGGREGATE            |                   |      0 |      0 |
|   7 |     TABLE ACCESS FULL        | EMPLOYEES         |      0 |      0 |
|   8 |    SORT AGGREGATE            |                   |      0 |      0 |
|   9 |     TABLE ACCESS FULL        | EMPLOYEES         |      0 |      0 |
-----------------------------------------------------------------------------
```

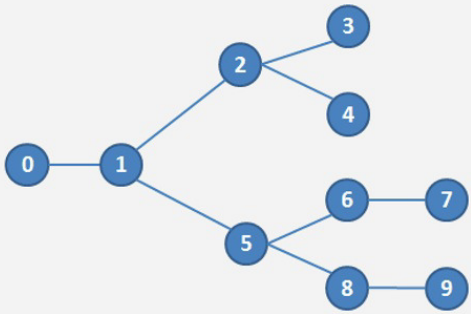

Figura 9.5: Diagrama em forma de árvore.

Com base nesse exemplo, podemos interpretar o plano de execução da seguinte forma:

- A operação 0 é a raiz da árvore. Ela possui apenas 1 nó filho, o nó 1.
- A operação 1 possui 2 nós filhos, os nós 2 e 5;
- A operação 2 possui 2 nós filhos, os nós 3 e 4;
- As operações 3 e 4 não possuem filhos;
- A operação 5 possui 2 nós filhos, os nós 6 e 8;
- A operação 6 possui 1 nó filho, o nó 7;
- A operação 7 não possui nós filhos;
- A operação 8 possui 1 nó filho, o nó 9;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- A operação 9 não possui nós filhos.

Com isso, vimos a forma como as operações se relacionam no plano de execução. Isso ajuda a entender como o plano de execução será lido e interpretado. Agora veremos como podem ser classificadas as operações, conhecendo seus tipos e suas características. Veremos que, independente do tipo da operação, a forma de leitura será sempre a mesma (com raras exceções). Podemos classificar as operações em: `Stand-Alone` , `Unrelated-Combine` e `Related-Combine` .

Operações do tipo Stand-Alone

As operação do tipo `Stand-Alone` são as mais comuns vistas nos planos de execução, pela forma simples de relacionamento. Elas são comumente visualizadas quando estamos trabalhando com instruções SQL sem muita complexidade. Contudo, elas podem estar presentes também em planos de execuções maiores ou derivados de instruções SQL mais complexas. Seguem as características desse tipo de operação:

O que são e quais são:

- Todas as operações que têm no máximo um nó são chamadas de operações autônomas (`Stand-Alone operations`);
- A maioria das operações são deste tipo e são comumente vistas em plano de execução, independente da complexidade da instrução SQL.

Regras de funcionamento deste tipo de operação:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Nós filhos são executados antes de seus nós pais (ver nota a seguir).
- Cada nó filho é executado no máximo uma vez;
- Todo nó filho gera um `Row Source` para seu nó pai.

> **Nota:** Embora a regra básica seja que os nós filhos sejam executados antes de seus nós pais, existem duas técnicas de otimização que fogem a esta regra: as operações Stand-Alone `COUNT STOPKEY` e `FILTER`, que veremos logo na sequência.

Veja o exemplo de um plano de execução com operações `Stand-Alone` a seguir:

```
SQL> select /*+ GATHER_PLAN_STATISTICS INDEX(employees EMP_JOB_IX)*/
  2         department_id
  3        ,count(*)
  4  from   employees
  5  where  job_id like '%CLERK'
  6  and    salary < 4000
  7  group by department_id;

DEPARTMENT_ID   COUNT(*)
------------- ----------
           30          5
           50         37

SQL> select *
  2  from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST -NOTE -ROWS'));

-----------------------------------------------------------------------
| Id  | Operation                    | Name       | Starts | A-Rows |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |      1 |      2 |
|   1 |  HASH GROUP BY               |            |      1 |      2 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES  |      1 |     42 |
|*  3 |    INDEX FULL SCAN           | EMP_JOB_IX |      1 |     45 |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("SALARY"<4000)
   3 - filter("JOB_ID" LIKE '%CLERK')
```

Hint para geração de estatísticas

SQL para visualizar as estatísticas

Diagrama do Plano de Execução para Operações Stand-Alone

Cada nó pai, tem apenas um nó filho

Nr. de execuções de cada operação

Nr. de linhas retornadas pela operação

Figura 9.6: Operações Stand-Alone.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Além das características do plano de execução, nos blocos de códigos, também são mostradas algumas informações adicionais como o uso do `hint GATHER_PLAN_STATISTICS` para a coleta de estatísticas, e também o uso do pacote `DBMS_XPLAN.DISPLAY_CURSOR` para visualização dessas estatísticas coletadas.

Esse plano de execução consiste apenas em operações do tipo `Stand-Alone`, isto é, autônomas. Aplicando as regras vistas anteriormente, podemos ver como são executadas as operações no plano de execução:

1. As operações Id 0, 1 e 2 têm um único nó filho cada uma (Id 1, 2 e 3, respectivamente). Essas três operações não podem ser as primeiras a serem executadas. Portanto, inicia-se com a execução da operação 3.
2. A operação 3 varre o índice `EMP_JOB_IX` aplicando o predicado de acesso `job_id like '%CLERK'`. Ao fazê-lo, ele extrai 45 `rowids` a partir do índice (valor visto na coluna `A-Rows`) e as passa para a sua operação principal (Id 2), sua operação pai.
3. A operação 2 acessa a tabela `employees` através dos 45 `rowids` passados pela operação Id 3. Para cada `rowid`, é realizada a leitura de 1 linha. Em seguida, o predicado de filtro `salary < 4000` é aplicado. Este filtro leva à exclusão de 3 linhas. As 42 linhas restantes são passadas para a operação principal (Id 1).
4. A operação Id 1 executa um `group by` nas linhas recebidas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

da operação Id 2. O resultado é reduzido para 2 linhas. A última operação é a operação 0, onde os dados são retornados como resultado da instrução SQL (`select statement`).

Operações `Stand-Alone` : `COUNT STOPKEY`

Esses tipos de operações aparecem no plano de execução quando tentamos limitar o resultado a um determinado número de registros em uma instrução SQL. Geralmente, essas limitações são utilizadas de forma genérica para constatar a existência ou não de ocorrências de registros em uma ou mais tabelas, não importando o retorno, mas sim, se retornam algo ou não, dado um ou mais critérios. No Oracle, utilizamos a pseudo-coluna chamada ROWNUM. Quando utilizamos o `rownum` em uma instrução SQL, a operação `COUNT STOPKEY` aparece no plano de execução.

- A operação `COUNT STOPKEY` é comumente utilizada para executar consultas top-n;
- O seu objetivo é realizar a parada do processamento assim que o número necessário de linhas for retornado ao chamador.

Observe o plano de execução a seguir:

```
SQL> select /*+ GATHER_PLAN_STATISTICS */ *
  2  from   departments
  3  where  rownum <= 15;

DEPARTMENT_ID DEPARTMENT_NAME              MANAGER_ID LOCATION_ID
------------- ---------------------------- ---------- -----------
           10 Administration                      200        1700
           20 Marketing                           201        1800
      ...
          140 Control And Credit                             1700
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
          150 Shareholder Services                          1700

15 rows selected.

SQL> select *
  2  from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST
-NOTE -ROWS'));

Execution Plan
----------------------------------------------------------------
| Id  | Operation          | Name        | Starts | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |      1 |     15 |
|*  1 |  COUNT STOPKEY     |             |      1 |     15 |
|   2 |   TABLE ACCESS FULL| DEPARTMENTS |      1 |     15 |
----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(ROWNUM<=15)
```

Figura 9.7: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img01code.txt

Neste plano de execução, é importante notar que o número de linhas retornadas pela operação 2 é limitado a 15. Isso é verdadeiro mesmo se a operação 2 for uma `FULL TABLE SCAN` de uma tabela que contenha mais de 15 linhas. O que acontece é que a operação 1 realiza a parada do processamento da operação 2 logo que o número de linhas necessárias for alcançado com base no critério da instrução SQL, ou seja, recuperar as 15 primeiras linhas.

Um exemplo muito comum, e que acaba pegando o pessoal de surpresa, é achar que, ao utilizar o `rownum` combinado com cláusula `where`, o resultado será primeiro a ordenação dos dados com base nos critérios de seleção, para depois o otimizador aplicar o `rownum`. Quem já teve essa experiência sabe que isso não ocorre

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

assim, as primeiras linhas que ele ler serão as linhas que ele retornará. Veja a seguir o plano de execução da mesma instrução SQL, agora acrescentando a cláusula `order by`.

```
SQL> select /*+ GATHER_PLAN_STATISTICS */ *
  2  from   departments
  3  where  rownum <= 15
  4  order by department_name;

Execution Plan
----------------------------------------------------------
| Id  | Operation           | Name        | Starts | A-Rows |
----------------------------------------------------------
|   0 | SELECT STATEMENT    |             |      1 |     15 |
|   1 |  SORT ORDER BY      |             |      1 |     15 |
|*  2 |   COUNT STOPKEY     |             |      1 |     15 |
|   3 |    TABLE ACCESS FULL| DEPARTMENTS |      1 |     15 |
----------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(ROWNUM<=15)
```

Figura 9.8: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img02code.txt

Note que a operação de ordenação só ocorre depois que o otimizador já recuperou os 15 registros da tabela `departments`. Para esse plano, a sequência de execução seria: Ids 3, 2, 1 e 0.

Operações `Stand-alone`: `FILTER`

A operação `FILTER` aplica um filtro quando o seu nó filho passa dados para essa operação. Dependendo dos dados, ele pode decidir evitar completamente a execução do nó filho e todas as suas operações dependentes. Nesses casos, a operação pai acaba sendo, de certa forma, não executada, mas avaliada a ponto de ser

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

ou não validada, para que o otimizador decida aplicá-la ou não após o processamento dos seus nós filhos.

No exemplo a seguir, note que no filtro `1 <> 1`, que é constituído de valores fixos, fica claro que, independente do que as operações possam retornar, quando chegar a vez deste filtro, ele anulará todos os possíveis resultados. Com isso, o otimizador acaba não considerando a execução dos seus filhos, evitando desperdício de execução. Este se trata de um caso onde a exceção foge à regra, onde uma operação pai acaba sendo executada antes da sua operação filha.

```
SQL> select /*+ GATHER_PLAN_STATISTICS INDEX(employees EMP_JOB_IX
) */
  2         first_name
  3        ,last_name
  4        ,job_id
  5  from   employees
  6  where  job_id = 'SALESMAN'
  7  and    1 <> 1;

no rows selected

SQL> select *
  2  from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST
-NOTE -ROWS'));
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------------------
| Id  | Operation                    | Name       | Starts | A-Rows |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |      1 |      0 |
|*  1 |  FILTER                      |            |      1 |      0 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES  |      0 |      0 |
|*  3 |    INDEX RANGE SCAN          | EMP_JOB_IX |      0 |      0 |
----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(NULL IS NOT NULL)
   3 - access("JOB_ID"='SALESMAN')
```

Figura 9.9: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img03code.txt

Vale ressaltar que, conforme as regras de relacionamento `Parent-Child`, a leitura do plano de execução desse exemplo deveria ser iniciada pela operação Id 3. Contudo, se analisarmos a coluna `Starts`, podemos ver que somente a operação Id 1 foi executada. O otimizador, simplesmente evitou o processamento das operações Id 2 e Id 3 porque os dados não têm nenhuma chance de passar pelo filtro aplicado pela operação 1.

Operações do tipo Unrelated-Combine

Ao contrário das operações `Stand-Alone`, as operações do tipo `Unrelated-Combine` possuem vários nós filhos, ou seja, nós presentes em um mesmo nível. Nesse caso, cada operação é executada de forma independente e na ordem em que aparecem. Elas tendem a aparecer em situações mais específicas, como no caso de agrupamento de dados, visualizações de forma hierárquica, grandes volumes de processamento ou união de várias instruções SQL. Seguem as características deste tipo de operação:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

O que são e quais são:

- Todas as operações que possuem vários nós filhos e que são executados de forma independente são operações `Unrelated-Combine`.
- As seguintes operações deste tipo são: `BITMAP AND`, `BITMAP OR`, `BITMAP MINUS`, `CONCATENATION`, `CONNECT BY WITHOUT FILTERING`, `HASH JOIN`, `INTERSECTION`, `MERGE JOIN`, `MINUS`, `MULTI-TABLE INSERT`, `SQL MODEL`, `TEMP TABLE TRANSFORMATION` e `UNION-ALL`

Regras de funcionamento deste tipo de operação:

- Os nós filhos são executados antes de seus nós pais.
- Os nós filhos são executados de forma sequencial.
- Todo nó filho é executado no máximo uma vez e de forma independente dos outros.
- Todo nó filho envia dados ao seu nó pai.

```
SQL> select /*+ GATHER_PLAN_STATISTICS */
  2         first_name
  3  from   employees
  4  union all
  5  select department_name
  6  from   departments
  7  union all
  8  select /*+ FULL(locations) */city
  9  from   locations;

SQL> select *
  2  from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST -NOTE -ROWS'));

-----------------------------------------------------------
| Id  | Operation          | Name        | Starts | A-Rows |
-----------------------------------------------------------
|   0 | SELECT STATEMENT   |             |      1 |    157 |
|   1 |  UNION-ALL         |             |      1 |    157 |
|   2 |   TABLE ACCESS FULL| EMPLOYEES   |      1 |    107 |
|   3 |   TABLE ACCESS FULL| DEPARTMENTS |      1 |     27 |
|   4 |   TABLE ACCESS FULL| LOCATIONS   |      1 |     23 |
-----------------------------------------------------------
```

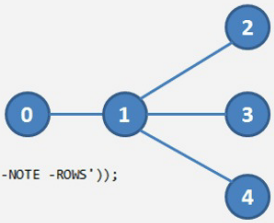

Figura 9.10: Operação Unrelated-Combine.

Nesse plano de execução, a operação Unrelated-Combine é a

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

cláusula `union all`. As outras três operações são operações autônomas (`Stand-Alone`). Ao aplicar as regras, você pode ver que o plano de execução realiza as operações da seguinte forma:

1. A operação 1 tem três nós filhos, dentre os quais a operação 2 é a primeira em ordem crescente. Desta forma, a execução se inicia com a operação 2.
2. A operação 2 varre a tabela `employees` e retorna 107 linhas para a sua operação pai (Id 1).
3. Quando a operação 2 é finalizada, a operação 3 é iniciada.
4. A operação 3 varre a tabela `departments` e retorna 27 linhas para a sua operação pai (Id 1).
5. Quando a operação 3 é finalizada, a operação 4 é iniciada.
6. A operação 4 varre a tabela `locations` e retorna 23 linhas para a sua operação pai (Id 1).
7. Operação 1 constrói um único conjunto de resultados de 157 linhas com base em todos os dados recebidos a partir dos seus nós filhos e os retorna como resultados da instrução SQL.

Observe como a coluna `Starts` mostra claramente que cada operação é executada apenas uma vez. A seguir, temos um segundo exemplo, mas agora envolvendo operação `Unrelated-Combine` e operações `Stand-Alone` com nós filhos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select /*+ GATHER_PLAN_STATISTICS */
  2         first_name
  3  from   employees
  4  where  department_id = 30
  5  union all
  6  select /*+ GATHER_PLAN_STATISTICS */
  7         first_name
  8  from   employees
  9  where  department_id = 90
 10  union all
 11  select /*+ GATHER_PLAN_STATISTICS */
 12         first_name
 13  from   employees
 14  where  department_id = 110;

SQL> select *
  2  from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST -NOTE -ROWS'));

-----------------------------------------------------------------------
| Id  | Operation                    | Name              | Starts | A-Rows |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |      1 |     11 |
|   1 |  UNION-ALL                   |                   |      1 |     11 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |      1 |      6 |
|*  3 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |      1 |      6 |
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |      1 |      3 |
|*  5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |      1 |      3 |
|   6 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |      1 |      2 |
|*  7 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |      1 |      2 |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENT_ID"=30)
   5 - access("DEPARTMENT_ID"=90)
   7 - access("DEPARTMENT_ID"=110)
```

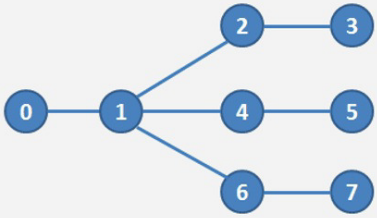

Figura 9.11: Operação Unrelated-Combine e operações Stand-Alone com nós filhos.

Nesse plano de execução, a operação `Unrelated-Combine` também é a cláusula `union all` . As outras três operações, com seus respectivos filhos, são operações autônomas (`Stand-Alone`). Ao aplicar as regras, você pode ver que o plano de execução realiza as operações da seguinte forma:

1. A operação 1 tem três nós filhos, dentre os quais a operação 2 é a primeira em ordem crescente. Portanto, a execução se inicia com a operação 2.
2. A operação 2 tem 1 filho. Portanto, este filho, operação 3, é executado antes do seu pai.
3. A operação 3 varre o índice `EMP_DEPARTMENT_IX` aplicando o predicado de acesso `department_id=30` . Ao fazê-lo, ele

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

extrai 6 `rowids` a partir do índice (valor visto na coluna `A-Rows`), e as passa para a sua operação principal (Id 2), sua operação pai.

4. A operação 2 acessa a tabela `employees` através dos 6 `rowids` passados pela operação Id 3. Para cada `rowid`, é realizada a leitura de 1 linha. As 6 linhas são passadas para a operação principal (Id 1).
5. Quando a operação 2 é completada, a operação 4 é iniciada.
6. A operação 4, também, tem 1 filho. Portanto, este filho, operação 5, é executado antes do seu pai.
7. A operação 5 varre o índice `EMP_DEPARTMENT_IX` aplicando o predicado de acesso `DEPARTMENT_ID=90`. Ao fazê-lo, ele extrai 3 `rowids` a partir do índice (valor visto na coluna `A-Rows`), e as passa para a sua operação principal (Id 4), sua operação pai.
8. A operação 4 acessa a tabela `employees` através dos 3 `rowids` passados pela operação Id 5. Para cada `rowid`, é realizada a leitura de 1 linha. As 3 linhas são passadas para a operação principal (Id 1).
9. Quando a operação 4 é completada, a operação 6 é iniciada.
10. A operação 6 também tem 1 filho. Portanto, este filho, operação 7, é executado antes do seu pai.
11. A operação 7 varre o índice `EMP_DEPARTMENT_IX` aplicando o predicado de acesso `DEPARTMENT_ID=110`. Ao fazê-lo, ele extrai 2 `rowids` a partir do índice (valor visto na coluna `a-rows`), e as passa para a sua operação principal (Id 6), sua operação pai.
12. A operação 6 acessa a tabela `employees` através dos 2 `rowids` passados pela operação Id 7. Para cada `rowid`, é realizada a leitura de 1 linha. As 2 linhas são passadas para a

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

operação principal (Id 1).

13. Operação 1 constrói um único conjunto de resultados de 11 linhas com base em todos os dados recebidos a partir dos seus nós filhos e os retorna como resultado da instrução SQL.

Observe novamente que na coluna `Starts` é mostrado que cada operação é executada apenas uma vez.

Operações do Tipo Related-Combine

Para este tipo de operação, o *modus operandi* de leitura do plano é o mesmo. Contudo, existe uma particularidade: caso a operação seja de um determinado tipo e possua vários nós filhos no mesmo nível, então o primeiro filho na sequência será o responsável por determinar a execução dos demais filhos desta operação. Então, diferente das operações do tipo `Unrelated-Combine`, onde seus filhos são executados de forma independente, nas operações do tipo `Related-Combine` a execução do primeiro filho determinará a forma como serão executadas as operações referentes aos demais filhos desta operação. A seguir mais detalhes e características sobre este tipo de operação:

O que são e quais são:

- Todas as operações que possuem vários filhos, sendo que um deles controla a execução de todos os outros filhos, são consideradas operações do tipo `Related-Combine`.
- As operações deste tipo são: `NESTED LOOPS`, `UPDATE`, `FILTER`, `CONNECT BY WITH FILTERING` e `BITMAP KEY ITERATION`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Regras de funcionamento deste tipo de operação:

- Os nós filhos são executados antes de seus nós pais.
- O filho com o menor ID controla a execução dos outros filhos.
- Os filhos não são executados sequencialmente. Em vez disso, um tipo de intercalação é executado e, nessa execução, o resultado de alguns filhos podem influenciar na execução de outros.
- O primeiro filho, o que controlará a execução dos demais, é executado no máximo uma vez. Todos os outros filhos podem ser executados várias vezes ou não serem executados.
- Nem todo filho retorna `ROW SOURCE` para seus pais.

As operações deste tipo seguem, na essência, as mesmas características. Contudo, cada uma delas, além de seguir o mesmo *modus operandi*, poderá, até certo ponto, ter seu próprio comportamento. Veremos agora exemplos de planos de execução em que estes tipos de operações são mostrados.

Operações Related-Combine: `NESTED LOOPS`

Neste exemplo de operação `Related-Combine` de `NESTED LOOPS` , note que o método de junção aparece duas vezes no plano de execução, e, em cada uma das vezes, podemos constatar o funcionamento e aplicação das regras para este tipo de operação.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select /*+ GATHER_PLAN_STATISTICS ORDERED USE_NL(d) FULL(e) INDEX(d DEPT_ID_PK) */
  2         e.first_name
  3        ,e.job_id
  4        ,d.department_name
  5  from   employees e
  6        ,departments d
  7  where  d.department_name <> 'Sales'
  8  and    e.commission_pct is null
  9  and    e.department_id = d.department_id;

SQL> select *
  2  from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST -NOTE -ROWS'));

-----------------------------------------------------------------------
| Id  | Operation                    | Name        | Starts | A-Rows |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |      1 |     72 |
|   1 |  NESTED LOOPS                |             |      1 |     72 |
|   2 |   NESTED LOOPS               |             |      1 |     72 |
|*  3 |    TABLE ACCESS FULL         | EMPLOYEES   |      1 |     72 |
|*  4 |    INDEX UNIQUE SCAN         | DEPT_ID_PK  |     72 |     72 |
|*  5 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     72 |     72 |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("E"."COMMISSION_PCT" IS NULL)
   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   5 - filter("D"."DEPARTMENT_NAME"<>'Sales')
```

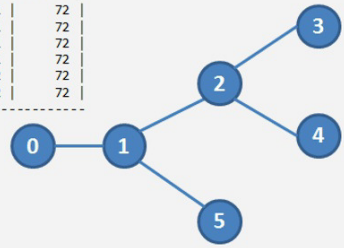

Figura 9.12: Operação `Related-Combine` de `NESTED LOOPS`.

O plano de execução realiza as operações da seguinte forma:

1. A operação 1 tem dois nós filhos (2 e 5), e, entre eles, a operação 2 é a primeira em ordem crescente. Portanto, a execução se inicia por esta operação.
2. A operação 2 também tem dois filhos (3 e 4), e, entre eles, a operação 3 é a primeira em ordem crescente. Desta forma, a execução se inicia por esta operação.
3. A operação 3 varre a tabela `employees`, aplica o filtro `e.commission_pct is null` e retorna os dados de 72 linhas à sua operação principal (2).
4. Para cada linha retornada pela operação 3, o segundo nó da operação `NESTED LOOPS`, operação 4, é executado uma vez. Isso é confirmado pela comparação da coluna `A-Rows` da operação 3 com a coluna `Starts` da operação 4.
5. A operação 4 varre o índice `DEPT_ID_PK` aplicando o

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

predicado de acesso `e.department_id=d.department_id`. Ao fazer isso, ele extrai 72 `rowids` do índice ao longo das 72 execuções e os passa para a sua operação pai (2).

6. Acompanhando o plano de execução, podemos observar que ele possui operações `NESTED LOOPS` encadeadas. Até aqui, resolvemos a operação `NESTED LOOPS` mais interna (Id 2). Agora, vamos para a operação `NESTED LOOPS` mais externa, seguindo a sequência de execução. Com base nas regras que se aplicam a operações `Related-Combine`, a próxima operação a ser executada é a operação 2, pois, pela hierarquia, a operação 2 é a dominante, por se tratar do primeiro nó e do tipo de operação. A operação 2 então envia os dados de 72 linhas para sua operação pai (Id 1).
7. Na sequência, a próxima operação a ser executada é a operação 5. A operação 5 acessa a tabela `departments` através dos 72 `rowids` passados pela operação 2. Para cada `rowid`, uma linha é lida. Em seguida, aplica-se o predicado de filtro `d.department_name<>'Sales'`. Este filtro acaba não implicando na exclusão de linhas, pois todas satisfazem o critério do filtro. Ele passa os dados, 72 linhas, para a sua operação principal (Id 1).
8. A operação 1 envia os dados de 72 linhas para a operação 0, que por sua vez as retorna como resultado da instrução SQL.

Operações Related-Combine: `FILTER`

A operação `FILTER` é comumente vista quando estamos utilizando operações que envolvem subconsultas com cláusulas `exists` ou `not exists`, por exemplo. No plano de execução, ela pode ser vista seguida de um ou mais filhos. Quando possui mais de um filho, seu funcionamento é semelhante à operação

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`Related-Combine` de `NESTED LOOPS`, mas quando possui apenas um filho, ela funciona como uma operação `Stand-Alone` e trabalha de forma independente. Da mesma forma que em `NESTED LOOPS`, o primeiro filho controla as execuções dos demais filhos. Veja um exemplo a seguir:

```
SQL> select /*+ GATHER_PLAN_STATISTICS RULE */ *
  2  from   employees e
  3  where  1=1
  4  and   not exists (select /*+ no_unnest */ 1
  5                    from  departments d
  6                    where  1=1
  7                    and    d.department_name = 'Sales'
  8                    and    d.department_id = e.department_id)
  9  and   not exists (select /*+ no_unnest */ 1
 10                    from   job_history j
 11                    where  1=1
 12                    and    to_char(j.employee_id) = to_char(e.employee_id));

SQL> select * from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST -NOTE -ROWS'));

-----------------------------------------------------------------------
| Id  | Operation                     | Name        | Starts | A-Rows |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |      1 |     67 |
|*  1 |  FILTER                       |             |      1 |     67 |
|   2 |   TABLE ACCESS FULL           | EMPLOYEES   |      1 |    107 |
|*  3 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     12 |      1 |
|*  4 |    INDEX UNIQUE SCAN          | DEPT_ID_PK  |     12 |     11 |
|*  5 |   TABLE ACCESS FULL           | JOB_HISTORY |     73 |      6 |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(( IS NULL AND  IS NULL))
   3 - filter("D"."DEPARTMENT_NAME"='Sales')
   4 - access("D"."DEPARTMENT_ID"=:B1)
   5 - filter(TO_CHAR("J"."EMPLOYEE_ID")=TO_CHAR(:B1))
```

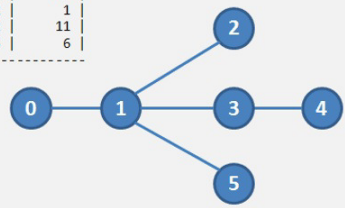

Figura 9.13: Operação Related-Combine de FILTER.

O plano de execução executa as operações da seguinte maneira:

1. A operação 1 tem três filhos (2, 3 e 5), e a operação 2 é a primeira delas em ordem crescente. Desta forma, a execução começa com a operação 2.
2. A operação 2 varre a tabela `employees` e retorna 107 linhas para sua operação pai (Id 1).
3. Para cada linha retornada pela operação 2, o segundo e

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

terceiro filhos da operação `FILTER` devem ser executados uma vez. O retorno da operação 2 foi de 107 linhas. Desta forma, pela regra, cada um dos demais filhos da operação 1 deve executar essa mesma quantidade de vezes. No entanto, um tipo de cache é implementado para reduzir esse número de execuções. A operação 3 é executada somente 12 vezes, uma vez para cada valor distinto encontrado na coluna `department_id` na tabela `employees`. A operação 5 é executada 73 vezes, uma vez para cada valor distinto encontrado na coluna `employee_id` na tabela `employees`, depois de aplicado o filtro imposto pela operação 3. Para melhor entendimento, veja como se chegou a estes números de execuções para as operações 3 e 5:

```
SQL> select count(*)
  2  from   (select distinct department_id
  3          from   employees);

COUNT(*)
----------
        12
```

Essa instrução SQL mostra o retorno de 12 linhas correspondentes aos departamentos distintos na tabela `employees`. Agora vamos para a instrução referente à operação 5:

```
SQL> select count(distinct nvl(e.employee_id,0)) count_dis
tinct
  2  from   employees e
  3        ,(select department_id
  4          from   departments
  5          where  department_name <> 'Sales') d
  6  where  1=1
  7  and    e.department_id(+) = d.department_id;

COUNT_DISTINCT
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
--------------
             73
```

Já essa instrução SQL mostra o retorno de 73 linhas correspondentes aos empregados distintos na tabela `employees`, depois de aplicado o filtro imposto pela operação 3, conforme já mencionado.

4. De acordo com as regras para operações independentes, a operação 4, executada antes da operação 3, verifica o índice `DEPT_ID_PK` aplicando o predicado de acesso `"D"."DEPARTMENT_ID"=:B1`. Aqui vemos algo novo no plano, a presença de uma variável `bind`, já vista em outros capítulos. Note que o otimizador também utiliza este tipo de notação para indicar passagem de valores. Neste exemplo, a variável `bind` `B1` é utilizada para passar o valor a ser verificado pela subconsulta. Como pode ser visto, nas 12 execuções, esta operação extraiu 11 `rowids` do índice `DEPT_ID_PK` e os passou para a sua operação pai (Id 3).
5. A operação 3 acessa o departamento de tabela através dos `rowids` passados de sua operação filha (4) e aplica o predicado de filtro `"D". "DEPARTMENT_NAME" = 'Sales'`.
6. A operação 5 varre a tabela `job_history` e aplica o seguinte predicado de filtro: `TO_CHAR("J"."EMPLOYEE_ID")=TO_CHAR(:B1)`. Observação: neste exemplo foi utilizada a função `to_char` para fazer com que o índice existente para a coluna `employee_id` não fosse utilizado pelo otimizador. Aqui, mais uma vez, aparece outra variável `bind`. Esta variável, por sua vez, também é utilizada para passar o valor a ser verificado pela subconsulta. Outra observação importante é que, como essa operação é usada apenas para aplicar uma

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

restrição, ela não retorna nenhum dado para seu pai (Id 1). Contudo, é importante observar que, para esta operação, 6 linhas atenderam ao predicado de filtro. Logo, por se tratar de um `not exists`, 6 linhas são descartadas.

7. A operação 1, depois de aplicar o predicado de filtro implementado com as operações 3 e 5, envia os dados de 67 linhas como resultado da instrução SQL. Note que neste passo o predicado de filtro aparece da seguinte forma: (`IS NULL AND IS NULL`). Isso se dá pela limitação da função `dbms_xplan`. Contudo, se utilizarmos o comando `explain plan` para mostrar o plano de execução, verá que ele traz a descrição correta do predicado de filtro referente a operação 1:

```
SQL> explain plan for
  2  select /*+ GATHER_PLAN_STATISTICS RULE */ *
  3  from   employees e
  4  where  1=1
  5  and   not exists (select /*+ no_unnest */ 1
  6                   from  departments d
  7                   where  1=1
  8                   and    d.department_name = 'Sales'
  9                   and    d.department_id = e.department_id)
 10  and   not exists (select /*+ no_unnest */ 1
 11                   from   job_history j
 12                   where  1=1
 13                   and    to_char(j.employee_id) = to_char(e.
employee_id));

Explained.

SQL> select * from table(dbms_xplan.display);
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------
| Id  | Operation                    | Name        |
----------------------------------------------------
|   0 | SELECT STATEMENT             |             |
|*  1 |  FILTER                      |             |
|   2 |   TABLE ACCESS FULL          | EMPLOYEES   |
|*  3 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |
|*  4 |    INDEX UNIQUE SCAN         | DEPT_ID_PK  |
|*  5 |   TABLE ACCESS FULL          | JOB_HISTORY |
----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( NOT EXISTS (SELECT /*+ NO_UNNEST */ 0 FROM "DEPARTMENTS"
               "D" WHERE "D"."DEPARTMENT_ID"=:B1 AND "D"."DEPARTMENT_NAME"='Sales')
               AND  NOT EXISTS (SELECT /*+ NO_UNNEST */ 0 FROM "JOB_HISTORY" "J" WHERE
               TO_CHAR("J"."EMPLOYEE_ID")=TO_CHAR(:B2)))
   3 - filter("D"."DEPARTMENT_NAME"='Sales')
   4 - access("D"."DEPARTMENT_ID"=:B1)
   5 - filter(TO_CHAR("J"."EMPLOYEE_ID")=TO_CHAR(:B1))
```

Figura 9.14: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img04code.txt

Operações Related-Combine: UPDATE

Esta operação aparece em instruções SQL de `update`, ou seja, quando estamos atualizando informações em um banco de dados. Igualmente à operação `FILTER`, ela pode vir acompanhada de vários filhos, mas, na grande maioria das vezes, ela terá apenas um filho. Os casos em que mais de um filho aparece são aqueles em que subconsultas são utilizadas em sua cláusula `set`. Tendo apenas um filho, ela será considerada uma operação `Stand-Alone`. Havendo mais um filho, o primeiro controlará a execução dos demais. Veja o exemplo a seguir:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> update /*+ GATHER_PLAN_STATISTICS */ employees e
  2  set    salary = (select /*+ FULL(j) */ avg(min_salary + max_salary)
  3                   from   jobs j
  4                   where  j.job_id = e.job_id)
  5        ,commission_pct = (select round((avg((min_salary + max_salary))*0.002)/100,2)
  6                           from   jobs);

107 rows updated.

SQL>
SQL> select * from table(dbms_xplan.display_cursor(format=>'ALLSTATS LAST -NOTE -ROWS'));

-----------------------------------------------------------
| Id  | Operation            | Name      | Starts | A-Rows |
-----------------------------------------------------------
|   0 | UPDATE STATEMENT     |           |      1 |      0 |
|   1 |  UPDATE              | EMPLOYEES |      1 |      0 |
|   2 |   TABLE ACCESS FULL  | EMPLOYEES |      1 |    107 |
|   3 |   SORT AGGREGATE     |           |     19 |     19 |
|*  4 |    TABLE ACCESS FULL| JOBS      |     19 |     19 |
|   5 |   SORT AGGREGATE     |           |      1 |      1 |
|   6 |    TABLE ACCESS FULL| JOBS      |      1 |     19 |
-----------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter("J"."JOB_ID"=:B1)
```

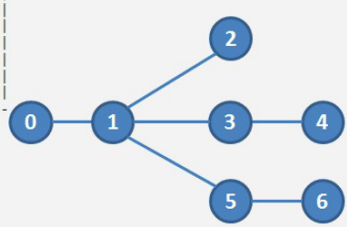

Figura 9.15: Operação Related-Combine de update.

O plano de execução executa as operações da seguinte forma:

1. A operação 1 tem três filhos (2, 3 e 5), e a operação 2 é a primeira das três em ordem crescente. Desta forma, a execução começa com a operação 2.
2. Operação 2 varre a tabela `employees` e retorna 107 linhas para sua operação pai (Id 1).
3. O segundo e o terceiro filhos (3 e 5) podem ser executados várias vezes, respeitando a quantidade máxima retornada pela sua operação controladora, no caso, a operação 2. Como essas duas operações são independentes (`Stand-Alone`) e cada uma delas possui um único filho, sua execução começa com as operações filho (Id 4 e Id 6).
4. Para cada valor distinto na coluna `job_id` retornado pela operação 2, a operação 4 varre a tabela `jobs` e aplica o predicado de filtro `"J". "JOB_ID" =: B1` . Ao fazer isso ao

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

longo das 19 execuções, ele extrai 19 linhas e as passa para sua operação pai (Id 3).

5. A operação 3 calcula a média de salários, utilizando a soma das colunas `min_salary` e `max_salary`, das linhas passadas para ela a partir da operação 4, e retorna o resultado para sua operação pai (Id 1).
6. Operação 6 varre a tabela `jobs`, extrai 19 linhas e as passa para sua operação pai (Id 5). Observe que esta subconsulta é executada apenas uma vez porque não está correlacionada à consulta principal. Note que ela não tem relacionamento com a tabela `employees`, executando de forma independente.
7. Operação 5 calcula a comissão média, calculando um pequeno percentual em cima da média da soma das colunas `min_salary` e `max_salary` das linhas passadas para ela a partir da operação 6, e retorna o resultado para sua operação pai (Id 1).
8. Operação 1 atualiza cada linha passada pela operação 2 com o valor retornado por seus outros filhos (Id 3 e Id 5). Observe que, mesmo a instrução `update` atualizando as 107 linhas, a coluna `A-Rows` mostrará 0 de retorno para esta operação.

Exceções à regra

Dentre os tipos de operações mencionados aqui, as regras se aplicam a quase todos os planos de execução que você vai vivenciar no seu dia a dia monitorando e *performando* instruções SQL. Contudo, existem algumas exceções, situações especiais que podem ocorrer. Observando bem o plano de execução, ao verificar quais predicados estão sendo aplicados, as tabelas nas quais eles

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

são executados e seu comportamento em tempo de execução, principalmente analisando as colunas `Starts` e `A-Rows`, você será capaz de constatar essas nuances e melhor interpretar da forma correta o plano de execução. A seguir são mostrados alguns destes exemplos em que a execução do plano foge às regras discutidas em tópicos anteriores.

Instrução SQL com subconsulta na cláusula `select`

Observe o plano de execução a seguir, onde são selecionados os nomes de todos os empregados e seus respectivos nomes de departamento. Note que não foi utilizada a prática convencional de utilizar o relacionamento entre tabelas utilizando a cláusula `from` da instrução SQL principal, e sim, um relacionamento construído na cláusula `select`. Note que neste plano de execução a operação 0 tem vários filhos (Id 1 e Id 3), o que não é comum, pois, como pode ser visto nos casos convencionais, a operação 0 sempre tem apenas um filho. Note também que, embora as operações 1 e 2 sejam executadas 12 vezes (coluna `Starts`), a operação 3 é executada apenas 1 vez, o que indica que ela não está sendo controlada pela operação 1, como caracterizaria uma operação `Related-Combine`. Ainda com relação às operações 1 e 2, note que elas referenciam a tabela `departments`, portanto, são as responsáveis por implementar a subconsulta.

```
SQL> select /*+ GATHER_PLAN_STATISTICS */
  2         last_name, (select department_name
  3                     from   departments d
  4                     where  d.department_id = e.department_id)
  5  from   employees e;

107 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(format=>'ALLST
ATS LAST -NOTE -ROWS'));
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------------------------------------------------------------------
| Id  | Operation                    | Name        | Starts | A-Rows |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |      1 |    107 |
|   1 |  TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     12 |     11 |
|*  2 |   INDEX UNIQUE SCAN          | DEPT_ID_PK  |     12 |     11 |
|   3 |  TABLE ACCESS FULL           | EMPLOYEES   |      1 |    107 |
----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("D"."DEPARTMENT_ID"=:B1)
```

Figura 9.16: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img05code.txt

Agora vamos ver a interpretação deste plano de execução e qual a sequência de execução das operações contidas nele.

1. A operação 3, que é a primeira a ser executada, varre a tabela `employees` e retorna todas as linhas para seu pai (Id 0). Aqui podemos observar que a ordem de execução muda em relação à regra. E como percebemos isso? Veja que as operações 1 e 2 executam 12 vezes. Este número é a quantidade de dados distintos de departamento (`department_id`) que existem na tabela `employees` . Logo, para que as operações 1 e 2 soubessem quantas vezes executar, a operação 3 teria que ser executada primeiro. Veja a instrução SQL a seguir, mostrando os valores distintos. Vale lembrar que o motor do banco de dados trabalha com `caches` de informações, conforme visto nos exemplos anteriores.

```
SQL> select count(distinct nvl(department_id,0)) qtde_dist
intos
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  2  from   employees;

QTDE_DISTINTOS
--------------
            12
```

2. Para cada linha retornada pela operação 3, a subconsulta deve ser executada uma vez. No entanto, também neste caso, os resultados são armazenados em cache e, portanto, a subconsulta é executada apenas uma vez para cada valor distinto na coluna `department_id` , no total 12 vezes.

3. Para executar a subconsulta, a operação 2 varre o índice `DEPT_ID_PK` aplicando o predicado de acesso `"D". "DEPARTMENT_D" =:B1` , extraindo os `rowids` e os retornando ao seu pai (Id 1). A variável `bind` (`B1`) é utilizada para passar o valor a ser verificado para a subconsulta. Em seguida, a operação 1 acessa a tabela `departments` com os `rowids` retornados e passa os dados para seu pai (Id 0).
4. Operação 0 retorna os dados como resultado da instrução SQL.

Instrução SQL com subconsulta na cláusula `where`

Neste tópico, veremos um exemplo com o uso de subconsulta na cláusula `where` . Através deste exemplo, também será possível observar como o otimizador tem a capacidade de transformar uma instrução SQL, ou seja, alterá-la, para traçar um plano de execução visando menor esforço. Veremos também como podemos influenciar o otimizador para usar features diferentes de otimização, dependendo da versão do banco.

Conforme pode ser visto a seguir, este plano de execução

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

contém duas operações do tipo `Stand-Alone` (operações independentes). Em um primeiro momento, você já notará que a operação filha (Id 2) é executada 27 vezes, enquanto a operação pai (Id 1) é executada apenas 1 vez.

```
<< QUADRO 1>>

SQL> select /*+ GATHER_PLAN_STATISTICS optimizer_features_enable(
'10.2.0.5') */ department_id
  2  from   departments
  3  where  department_id not in (select department_id
  4                               from   employees);

no rows selected

SQL> select * from table(dbms_xplan.display_cursor(format=>'ALLST
ATS LAST -NOTE -ROWS'));

Execution Plan
----------------------------------------------------------------
| Id  | Operation          | Name       | Starts | A-Rows |
----------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |      1 |      0 |
|*  1 |  INDEX FULL SCAN   | DEPT_ID_PK |      1 |      0 |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES  |     27 |     27 |
----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( IS NULL)
   2 - filter(LNNVL("DEPARTMENT_ID"<>:B1))
```

Figura 9.17: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img06code.txt

Aqui podemos notar duas características importantes. A primeira é o uso do `hint` `OPTIMIZER_FEATURES_ENABLE` , que serve para induzir o otimizador a utilizar features de uma

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

determina versão do banco de dados para otimizar a instrução SQL. Nesse caso, estamos dizendo ao otimizador que ele deve usar as features da versão 10.2.0.5. Esse exemplo foi executado na versão de banco 11.2.0.2.0 (Oracle XE). Sem indicar esse `hint`, o otimizador daria preferência por utilizar as `features` desta versão, que tendem a ser mais novas e com mais recursos otimizados, como o uso de `anti-join` com reconhecimento de valores nulos (`NULL-aware anti-join`). Uma operação `anti-join` consiste em termos uma subconsulta não correlacionada, com uma cláusula `NOT IN` ou `NOT EXISTS`. Nesse caso, qualquer linha encontrada nesta subconsulta não é incluída no resultado da instrução SQL. Veremos um exemplo disso mais à frente.

Outra característica tem relação com o uso de `dbms_xplan`, que, conforme vimos anteriormente, não pode trazer a informação do filtro corretamente. Podemos observar isso pelo predicado de filtro `is null` na operação 1. Neste exemplo, essa informação correta do filtro vai ser imprescindível para notarmos que o otimizador fez transformações em nossa instrução SQL. Veja novamente a mesma instrução, mas agora mostrando o predicado de filtro corretamente. Note que utilizamos outro procedimento da package `dbms_xplan` para conseguir uma formatação diferente do mesmo plano e assim enxergar o detalhe do filtro 1.

```
<< QUADRO 2>>

SQL> explain plan for
  2  select /*+ GATHER_PLAN_STATISTICS optimizer_features_enable(
'10.2.0.5') */ department_id
  3  from   departments
  4  where  department_id not in (select department_id
  5                               from   employees);

Explained.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select * from table(dbms_xplan.display);

Execution Plan
----------------------------------------
| Id  | Operation          | Name       |
----------------------------------------
|   0 | SELECT STATEMENT   |            |
|*  1 |  INDEX FULL SCAN   | DEPT_ID_PK |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES  |
----------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( NOT EXISTS (SELECT 0 FROM "EMPLOYEES" "EMPLOYEES" WHERE
               LNNVL("DEPARTMENT_ID"<>:B1)))
   2 - filter(LNNVL("DEPARTMENT_ID"<>:B1))
```

Figura 9.18: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img07code.txt

Observe que o predicado de filtro 1, referente à operação 1, mostra uma cláusula `not exists` em vez de uma cláusula `not in` como a existente na instrução SQL original. Isso indica que o otimizador transformou a instrução SQL. Para melhor entendimento da sequência de execução das operações, considere a instrução SQL modificada como se segue:

```
SQL> select department_id
  2  from   departments d
  3  where  not exists (select 1
  4                     from   employees e
  5                     where  e.department_id <> d.department_id
);
```

Agora vamos dar um olhada na execução das operações do plano de execução (<>).

1. A operação 1 é a primeira a ser executada e varre o índice `DEPT_ID_PK` . Para cada valor na coluna `department_id` , o

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

otimizador executa a operação 2. Conforme mostrado pelos predicados de filtro, esta operação é realizada devido à subconsulta `NOT EXISTS (SELECT 0 FROM "EMP" "EMP" WHERE LNNVL ("DEPTNO" <>: B1))` , mostrada no <>. Observe que o otimizador de consulta transformou o `not in` em `NOT EXISTS` . A variável de `bind B1` é usada para passar o valor a ser verificado para a subconsulta. Além dessas transformações, o otimizador ainda utiliza a função `LNNVL` . Essa função é usada para avaliar uma condição em que um dos operandos dessa condição pode ter valor `NULL` , tomando como verdadeira a condição caso tenha. Além disso, se essa condição for falsa ou desconhecida, a função retornará verdadeiro. Se a condição for verdadeira, ela retornará falso.

2. Operação 2 varre a tabela `employees` , aplica o predicado de filtro `LNNVL ("DEPARTMENT_ID" <>: B1)` e retorna os dados para seu pai (Id 1).
3. Para cada linha que satisfaz a operação do predicado de filtro 1, são passados seus dados para seu pai (Id 0).
4. Operação 0 retorna os dados como resultado da instrução SQL.

Vale ressaltar que as informações mostradas referentes aos predicados (área *Predicate Information*) foram de fundamental importância para o entendimento na interpretação do plano de execução. Sem este tipo de informação, a interpretação do plano poderia ser dificultada. Conforme mencionado anteriormente, veja a mesma instrução SQL sendo executada adiante, agora, sem a indução realizada através do `hint OPTIMIZER_FEATURES_ENABLE` em uma base com a versão do

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

banco de dados 11.2.0.2.0 (Oracle XE).

```
SQL> select /*+ GATHER_PLAN_STATISTICS */ department_id
  2  from   departments
  3  where  department_id not in (select department_id
  4                               from   employees);

no rows selected

SQL> select * from table(dbms_xplan.display_cursor(format=>'ALLST
ATS LAST -NOTE -ROWS'));

Execution Plan
----------------------------------------------------------------------
| Id  | Operation                | Name               | Starts | A-Rows |
----------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                    |      1 |      0 |
|*  1 |  FILTER                  |                    |      1 |      0 |
|   2 |   NESTED LOOPS ANTI SNA|                    |      0 |      0 |
|   3 |    INDEX FULL SCAN       | DEPT_ID_PK         |      0 |      0 |
|*  4 |    INDEX RANGE SCAN      | EMP_DEPARTMENT_IX  |      0 |      0 |
|*  5 |   TABLE ACCESS FULL      | EMPLOYEES          |      1 |      1 |
----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( IS NULL)
   4 - access("DEPARTMENT_ID"="DEPARTMENT_ID")
   5 - filter("DEPARTMENT_ID" IS NULL)

/*
   Comentário do autor:

   1 - filter( IS NULL) foi igual a (saída gerando pelo comando explain):

   1 - filter( NOT EXISTS (SELECT 0 FROM "EMPLOYEES" "EMPLOYEES" WHERE
             "DEPARTMENT_ID" IS NULL))
*/
```

Figura 9.19: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap09img08code.txt

Vamos dar uma olhada nesse plano de execução:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

1. A operação 1 tem dois filhos (2 e 5), e a operação 2 é a primeira delas em ordem crescente. Desta forma, ela deveria ser a primeira operação a ser executada se não estivéssemos falando de exceções. No entanto, a operação 5 é a primeira a ser executada. O que reforça essa tese são os números de execuções vistos nas operações 2, 3 e 4. Note que eles não foram executados nenhuma vez no plano. Isso leva a crer que a operação 5 tenha sido eficaz de tal forma, que não fez sentido o otimizador executar as operações 2, 3 e 4.
2. Operação 5 varre a tabela `employees` , aplica o predicado de `"department_id" is null` e retorna os dados para seu pai (Id 1).
3. Com relação às operações 2, 3 e 4, o otimizador simplesmente evitou o processamento das operações, porque os dados não têm nenhuma chance de passar pelo filtro aplicado pela operação 1, pois a subconsulta sempre retornará um registro, invalidando a condição `not exists` . Veja a instrução SQL a seguir, simulando a subconsulta alterada pelo otimizador.

```
 SQL> select 0 from "employees" "employees" where "departme
nt_id" is null;

         0
 ----------
         0

 SQL>
```

4. Para cada linha que satisfaz a operação do predicado de filtro 1, são passados seus dados para seu pai (Id 0).
5. Operação 0 retorna os dados como resultado da instrução SQL.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Além das operações que aparecem no plano de execução, alguns indicadores também podem aparecer no plano, como o indicador `[CHOOSE]`, que indica `OPTIMIZER_GOAL` (o modo do otimizador) para a consulta. Há também o indicador `[:Q65001]` que indica que esta parte específica da consulta está sendo executada em paralelo e, por fim, o indicador `[ANALYZED]` que indica que o objeto em questão foi analisado e atualmente existem estatísticas disponíveis para o CBO usar.

O objetivo deste capítulo foi demonstrar como os planos de execução podem ser lidos e interpretados. Vimos as características e regras que são utilizadas para a interpretação e como isso pode ajudar no monitoramento, análise e identificação de possíveis problemas. Vale ressaltar que é muito importante sempre verificarmos, de forma preventiva, o plano de execução dos programas e instruções SQL, não só quando problemas ocorrem, mas também quando estamos desenvolvendo tais programas a fim de evitar futuros incômodos. Agora que você já tem conhecimento sobre essas estruturas e mecanismos de otimização, de como funcionam e quais suas características e regras, veremos como melhor gerar, extrair e visualizar essas informações. O próximo capítulo tratará disso.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 10

GERANDO ESTATÍSTICAS PARA O OTIMIZADOR

Até o momento, falamos bastante sobre a importância dos dados estatísticos para que o otimizador do banco de dados Oracle possa reunir informações suficientes para nos ajudar na escolha do plano de execução mais adequado para uma instrução SQL. Em capítulos anteriores, mostramos uma prévia de como utilizar, por exemplo, o comando `analyze` , que tem como objetivo colher informações e gerar tais estatísticas e que foi o primeiro a fazer este papel. Agora, chegou o momento de falarmos especificamente deste trabalho. Falaremos sobre qual a melhor forma de colher essas estatísticas, dar subsídios para o otimizador determinar qual o melhor método de junção, o tipo de junção mais eficiente e quais índices serão mais performáticos para se executar uma determinada instrução SQL.

Antes de mais nada, vamos refrescar o conceito de estatísticas. As estatísticas geradas e utilizadas pelo otimizador são uma coleção de dados que descrevem detalhes sobre o banco de dados e os objetos pertencentes a ele. Você verá também que, além de informações sobre os objetos do banco, o otimizador poderá ter acesso às estatísticas do sistema. Todas essas estatísticas poderão ser utilizadas pelo otimizador de consulta para escolher o melhor

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

plano de execução para cada instrução SQL que enviamos para o banco de dados. Para isso, vamos utilizar um pacote que já vem instalado no banco de dados Oracle chamado `dbms_stats` .

Se você consultar as documentações mais recentes do banco de dados Oracle, você vai perceber que a Oracle é enfática em dizer que o mais recomendado é que seja usado o pacote `dbms_stats` em vez do comando `analyze` para coletar estatísticas. O pacote `dbms_stats` possui muito mais recursos, pois ele trabalha com o que há de mais novo no que diz respeito às versões novas dos bancos de dados. O pacote permite coletar estatísticas em paralelo, estatísticas globais para objetos particionados, de várias formas.

Quanto à coleta através do `analyze` , ela permanece mesmo nas versões mais novas do banco de dados, pela política da Oracle de sempre manter a compatibilidade e também por conta dos casos em que são necessárias a coleta de estatísticas não relacionada ao otimizador baseado em custo.

Na seção *O Modo baseado em Custos (CBO)* deste livro, falo sobre o `analyze` . Neste capítulo, eu passo alguns exemplos de como utilizá-lo.

Nota: Você pode ainda encontrar o comando `analyze` sendo usado em formato de pacote, através da referência `dbms_utility` . A única diferença em usar essa package é a possibilidade de coletar estatísticas de um schema ou do banco de dados por completo.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

10.1 O PACOTE DBMS_STATS

O pacote `dbms_stats` está disponível desde a versão do banco de dados Oracle 8i. De lá para cá, ele vem sendo aprimorado com o intuito de cada vez mais melhorar a coleta das estatísticas. Dessa forma, pode haver algumas nuances entre as versões do banco de dados. As estatísticas geradas podem estar armazenadas no dicionário de dados do Oracle ou podem ser armazenadas em uma tabela específica, criada em algum esquema do usuário, por exemplo. Contudo, apenas as estatísticas armazenadas no dicionário de dados do Oracle são as que afetam o otimizador baseado em custo (CBO).

Através deste pacote, você poderá coletar, modificar, visualizar, exportar, importar e excluir estatísticas. O pacote `dbms_stats` pode gerar estatísticas referentes a índices, tabelas, colunas e partições específicas do banco de dados, bem como gerar estatísticas de um schema ou banco de dados completo.

Ele possui algumas limitações com relação à geração de estatísticas para `clusters` ou geração de estatísticas em paralelo quando se trata de índices, mas elas são contornáveis.

Além das estatísticas coletadas referentes aos objetos do banco de dados, o pacote `dbms_stats` também possibilita a coleta de estatísticas do sistema. Embora não seja o foco do livro, é bom você saber que isso é possível. A coleta de estatísticas do sistema permite que o otimizador considere a eficiência e utilização de I/O e CPU do sistema. Para cada plano de execução em potencial, o otimizador calcula estimativas para os custos de I/O e CPU. Por isso, é importante o otimizador conhecer as características do sistema para escolher o plano mais eficiente em relação também ao

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

que se refere a custos de I/O e de CPU.

> **NOTA:** Use a rotina `dbms_stats.gather_system_stats` para coletar estatísticas do sistema. Para maiores informações, consulte a documentação da Oracle ou o DBA responsável pelo banco de dados.

Em resumo, as funcionalidades incluídas no pacote `dbms_stats` permitem gerar tais estatísticas:

- Estatísticas de tabelas: número de linhas, número de blocos e comprimento médio de linha.
- Estatísticas de colunas: número de valores distintos (`NDV`) na coluna, número de valores nulos na coluna e distribuição de dados (histograma).
- Estatísticas de índices: número de `Leaf Blocks` , níveis e fator de agrupamento (`Clustering`).
- Estatísticas de sistema: desempenho e utilização de I/O, desempenho e utilização da CPU.

Vamos agora conhecer algumas formas como podemos utilizar esse pacote para gerir as estatísticas do banco de dados e seus objetos:

Para gerar a estatística de uma determinada tabela, utilize o comando a seguir:

```
SQL> exec dbms_stats.gather_table_stats('TTUN', 'EMPLOYEES');
PL/SQL procedure successfully completed.
```

Nesse exemplo, estamos gerando a estatística para a tabela

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`'EMPLOYEES'` que pertence ao schema TTUN (`Schema` exemplo para o livro). Para facilitar, entenda Schema como se fosse o usuário dono da tabela.

Caso você queira gerar as estatísticas somente em cima de X% das linhas da tabela, utilize o comando a seguir:

```
SQL> exec dbms_stats.gather_table_stats('TTUN', 'DEPT', estimate_
percent => 15);
PL/SQL procedure successfully completed.
```

Já nesse exemplo, estamos solicitando que as estatísticas sejam geradas apenas em cima de 15% das linhas da tabela `'DEPT'` . Isso é informado através do parâmetro `estimate_percent` .

Você também pode informar se as estatísticas também devem ser geradas para os índices pertencentes à tabela fruto da geração:

```
SQL> exec dbms_stats.gather_table_stats('TTUN', 'EMP', estimate_p
ercent => 50, cascade => TRUE);
PL/SQL procedure successfully completed.
```

Nessa execução, estamos solicitando a geração das estatísticas da tabela `emp` , mas para 50% da tabela, e estamos solicitando que as estatísticas dos índices também sejam geradas.

Nota: O parâmetro `cascade` deve ser considerado para todas as coletas de estatística, para todos os índices da tabela que está sendo analisada. Antes do Oracle 10g, o padrão era `FALSE` , mas a partir do 10g o padrão é `auto_cascade` , ou seja, caso seja necessário, o Oracle coletará as estatísticas dos índices da tabela que está sendo analisada.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Para gerar estatísticas apenas de índices, utilize um dos procedimentos a seguir, informando o nome do índice que deve ser gerado (uma execução para cada índice) por meio do procedimento `gather_index_stats` :

```
SQL> exec dbms_stats.gather_index_stats('TTUN', 'EMP_IDX_1');
PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_index_stats('TTUN', 'DEPT_ID_PK', est
imate_percent => 20);
PL/SQL procedure successfully completed.
```

Veja na sequência outros exemplos de execução do pacote `dbms_stats` .

Gerando estatísticas de um determinado esquema:

```
SQL> exec dbms_stats.gather_schema_stats('TTUN');
PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_schema_stats('TTUN', estimate_percent
 => 10);
PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_schema_stats('TTUN', estimate_percent
 => 30, cascade => TRUE);
PL/SQL procedure successfully completed.
```

Nesse exemplo, estatísticas serão geradas para o schema `TTUN` , que foi criado especificamente para armazenar todas as tabelas utilizadas nos exemplos deste livro.

Gerando estatísticas do banco de dados:

```
SQL> exec dbms_stats.gather_database_stats;
PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_database_stats(estimate_percent => 25
);
PL/SQL procedure successfully completed.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> exec dbms_stats.gather_database_stats(estimate_percent => 35
, cascade => TRUE);
PL/SQL procedure successfully completed.
```

Nesse exemplo, basta apenas executar o comando estando conectado ao banco de dados do qual se quer coletar as estatísticas.

Atualizando estatísticas do dicionário de dados:

```
SQL> exec dbms_stats.gather_dictionary_stats;
PL/SQL procedure successfully completed.
```

Esse procedimento apenas atualiza as estatísticas já existentes, ou seja, estatísticas geradas anteriormente para determinados objetos. Caso tenham sido criados outros objetos e eles não tenham dados estatísticos, gerados em algum momento, eles não serão colhidos utilizando este procedimento, apenas as estatísticas já colhidas serão atualizadas.

Gerando estatísticas do sistema:

```
SQL> exec dbms_stats.gather_system_stats;
PL/SQL procedure successfully completed.
```

Como já foi mencionado, além da geração de estatísticas, você também pode excluí-las. Veja na sequência como fazer isso.

Excluindo as estatísticas de uma tabela, no exemplo, da tabela `'DEPT'`:

```
SQL> exec dbms_stats.delete_table_stats('TTUN', 'DEPT');
PL/SQL procedure successfully completed.
```

Excluindo as estatísticas de uma coluna de tabela, no exemplo, da coluna `'STATE_PROVINCE'` da tabela `'LOCATIONS'`:

```
SQL> exec dbms_stats.delete_column_stats('TTUN', 'LOCATIONS', 'ST
ATE_PROVINCE');
PL/SQL procedure successfully completed.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Excluindo as estatísticas de um índice, no exemplo, do índice `'REG_ID_PK'` (tabela `'REGIONS'`):

```
SQL> exec dbms_stats.delete_index_stats('TTUN', 'REG_ID_PK');
PL/SQL procedure successfully completed.
```

Excluindo as estatísticas de um schema inteiro, no exemplo, o esquema `'TTUN'`:

```
SQL> exec dbms_stats.delete_schema_stats('TTUN');
PL/SQL procedure successfully completed.
```

Excluindo as estatísticas de um banco de dados inteiro, no exemplo, o banco de dados no qual o comando está sendo executado:

```
SQL> exec dbms_stats.delete_database_stats;
PL/SQL procedure successfully completed.
```

Excluindo as estatísticas do dicionário de dados de um banco de dados, no exemplo, o dicionário do banco de dados no qual o comando está sendo executado:

```
SQL> exec dbms_stats.delete_dictionary_stats;
PL/SQL procedure successfully completed.
```

10.2 QUANDO EU DEVO GERAR ESTATÍSTICAS?

Como os objetos do banco de dados podem mudar constantemente, é muito importante que as estatísticas sejam atualizadas de tempos em tempos. Esse trabalho pode ser realizado de forma manual, através dos comandos aqui apresentados, bem como de forma automática, com a qual você poderá configurar para que o Oracle as atualize de tempos em tempos. É

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

extremamente importante que os dados estatísticos sejam precisos e que reflitam a situação real dos objetos do banco de dados para que o otimizador possa trabalhar com dados fiéis e, assim, ser assertivo na escolha do plano de execução.

O uso da coleta automática de estatísticas é bastante recomendado pelo fato de tornar esse trabalho regular, pois você garante que sempre será realizada a tarefa de atualização nos dias e horários predeterminados, reduzindo as chances de estatísticas ausentes ou obsoletas no banco de dados. Além disso, estatísticas incorretas ou não existentes são ruins para o otimizador e consequentemente para o funcionamento do banco de dados.

10.3 GERAÇÃO AUTOMÁTICA DE ESTATÍSTICAS

As estatísticas poderão ser geradas automaticamente através do `job gather_stats_job` nas versões a partir do banco de dados Oracle 10g. Este `job` executará a geração para todos os objetos do banco de dados que não possuírem estatísticas coletadas anteriormente ou que estejam obsoletas, ou seja, quando o objeto em questão for modificado de forma significativa (modificações em mais de 10% das linhas).

Ele faz isso através da chamada ao procedimento `dbms_stats.gather_database_stats_job_proc`, que é muito semelhante a `dbms_stats.gather_database_stats`, com uma principal diferença: ele prioriza os objetos de banco de dados que requerem estatísticas, para que os objetos que mais precisam desses dados atualizados sejam processados primeiro. Isso garante que as estatísticas mais necessárias sejam reunidas antes que a janela de

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

manutenção seja fechada. O banco de dados Oracle sabe quais estatísticas deve priorizar na geração, pois ele documenta todas as ações de DML ocorridas na SGA e armazena essas informações. A instrução a seguir mostra como podemos visualizar esses dados documentados pelo banco:

```
SQL> exec dbms_stats.flush_database_monitoring_info;
PL/SQL procedure successfully completed.

SQL> select table_owner, table_name, inserts, updates, deletes, t
imestamp
  2  from   dba_tab_modifications
  3  where  table_name = 'EMPLOYEES';

TABLE_OWNER          TABLE_NAME        INSERTS    UPDATES    DELE
TES TIMESTAMP
-------------------- -------------- ---------- ---------- -------
--- ---------
TTUN                 EMPLOYEES               0        107
  0 26-SEP-18

SQL>
```

Nota: As informações de DML são preenchidas apenas para tabelas com o atributo `monitoring`. Por motivos de desempenho, o banco de dados Oracle não preenche essas informações imediatamente quando ocorrem as modificações reais de DML. Para visualizá-las, execute o procedimento `flush_database_monitoring_info` no pacote `dbms_stats` para o preenchimento e exibição das informações mais recentes. O privilégio do sistema `analyze_any` é necessário para executar este procedimento.

É possível verificar o percentual modificado das ações de DML

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

em uma tabela, através da seguinte execução:

```
SQL> select dbms_stats.get_prefs('STALE_PERCENT', 'TTUN', 'EMP')
stale_percent
  2  from   dual;

STALE_PERCENT
--------------------------------
10

SQL>
```

Também é possível verificar se existem estatísticas obsoletas por meio das informações contidas na `dba_tab_statistics`, mais precisamente através da coluna `stale_stats`. Se o valor é `"NO"`, significa que as estatísticas estão atualizadas; se o valor for `"YES"`, indica que são obsoletas e, se o valor for vazio, indica que as estatísticas nunca foram coletadas. Veja a instrução SQL a seguir:

```
SQL> select stat.owner
  2        ,stat.table_name
  3                     ,stale_stats
  4  from   dba_tab_statistics stat
  5  where  table_name = 'EMP';
OWNER                          TABLE_NAME                     STA

---------------------------- ------------------------------ ----

TTUN                           EMP                            NO


SQL>
```

Pois bem, voltando ao `job`, ele é criado de forma automática junto da criação do banco de dados e é gerenciado pelo programa `Scheduler` do banco de dados. Ele também já vem com uma programação padrão de funcionamento quanto aos dias e horários.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Para mais informações, consulte a documentação do banco de dados Oracle. Lá, você obterá mais informações sobre as tarefas do `Scheduler` e das janelas de manutenção.

Para verificar se a tarefa relacionada ao `'GATHER_STATS_JOB'` está ativa, execute a instrução SQL a seguir:

```
select * from dba_scheduler_jobs where job_name = 'GATHER_STATS_J
OB';
```

Para desabilitar ou habilitar a execução do `job`, execute um dos comandos a seguir respectivamente:

```
begin
  dbms_scheduler.disable('GATHER_STATS_JOB');
end;
/

begin
  dbms_scheduler.enable('GATHER_STATS_JOB');
end;
/
```

Para ver a última vez em que o `job` `GATHER_STATS_JOB` foi executado:

```
SQL> select last_start_date
  2  from   dba_scheduler_jobs
  3  where  job_name = 'GATHER_STATS_JOB';

no rows selected
SQL>
```

Uma forma bem primária para verificar se as estatísticas estão atualizadas é através da instrução SQL a seguir:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select last_analyzed from dba_tables where table_name = 'EMP
';
LAST_ANAL
————
25-SEP-18
```

Outras consultas que podem ser utilizadas para verificar informações de última coleta de estatísticas:

Buscando informações da última coleta de uma determinada tabela:

```
SQL> select stat.owner
  2        ,stat.table_name
  3        ,stat.object_type
  4        ,stat.num_rows
  5        ,to_char(stat.last_analyzed, 'dd/mm/rrrr hh24:mi:ss')
last_analyzed
  6  from   dba_tab_statistics stat
  7  where  table_name = 'EMP';

OWNER           TABLE_NAME  OBJECT_TYPE   NUM_ROWS LAST_ANALYZED

--------- ---------------- ------------ ---------- --------------
-----
TTUN                   EMP        TABLE         14 25/09/2018 00:
37:29

SQL>
```

Buscando informações da última coleta de um determinado índice:

```
SQL> select stat.owner
  2        ,stat. table_name
  3        ,stat.object_type
  4        ,stat.num_rows
  5        ,to_char(stat.last_analyzed, 'dd/mm/rrrr hh24:mi:ss')
last_analyzed
  6  from   dba_ind_statistics stat
  7  where  index_name = 'REG_ID_PK'
  8  order  by last_analyzed;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
OWNER          TABLE_NAME  OBJECT_TYPE    NUM_ROWS LAST_ANALYZED

------- ---------------- ------------ ---------- ---------------
----
HR                REGIONS        INDEX          4 25/09/2018 23:0
3:55
TTUN              REGIONS        INDEX          4 25/09/2018 23:0
4:44

SQL>
```

A partir da versão 11g do bando de dados Oracle, o `job` `gather_stats_job` não existe mais, sendo substituído então por `jobs` de estatística que ficam como tarefas automáticas de manutenção. Para consultar essas informações, você poderá utilizar a instrução SQL a seguir:

```
SQL> select client_name, status from dba_autotask_client;
CLIENT_NAME
STATUS
-----------------------------------------------------------------
____
auto optimizer stats collection
ENABLED
auto space advisor
ENABLED
sql tuning advisor
DISABLED

SQL>
```

Para desabilitar, execute a instrução:

```
SQL> begin
  2    dbms_auto_task_admin.disable ( client_name  => 'AUTO OPTIM
IZER STATS COLLECTION'
  3                                  ,operation    => null
  4                                  ,window_name  => null );
  5  end;
  6  /
PL/SQL procedure successfully completed.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select client_name, status from dba_autotask_client;
CLIENT_NAME                                      STATUS
---------------------------------------------- ----------
auto optimizer stats collection                  DISABLED
auto space advisor                               ENABLED
sql tuning advisor                               DISABLED

SQL>
```

Para habilitar, execute a instrução:

```
SQL> begin
  2    dbms_auto_task_admin.enable ( client_name  => 'AUTO OPTIMI
ZER STATS COLLECTION'
  3                                ,operation    => null
  4                                ,window_name  => null );
  5  end;
  6  /
PL/SQL procedure successfully completed.
SQL> select client_name, status from dba_autotask_client;
CLIENT_NAME                                      STATUS
---------------------------------------------- ---------
auto optimizer stats collection                  ENABLED
auto space advisor                               ENABLED
sql tuning advisor                               DISABLED

SQL>
```

Para ver o histórico das coletas de todas as estatísticas, sendo de forma manual ou via `job`, use a seguinte instrução SQL:

```
SQL> select operation
  2        ,target
  3        ,to_char(start_time,'dd/mm/rrrr hh24:mi:ss') start_tim
e
  4        ,to_char(end_time,'dd/mm/rrrr hh24:mi:ss') end_time

  5  from   dba_optstat_operations
  6  where  operation <> 'copy_table_stats'
  7  and    target like '%DEPT';

OPERATION            TARGET               START_TIME           EN
D_TIME
-------------------- -------------------- -------------------- --
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
-----------------
gather_table_stats   TTUN.DEPT            20/09/2018 22:50:53  20
/09/2018 22:50:54
delete_table_stats   TTUN.DEPT            24/09/2018 22:27:05  24
/09/2018 22:27:06
delete_table_stats   TTUN.DEPT            24/09/2018 22:28:01  24
/09/2018 22:28:01

SQL>
```

> **Nota:** Caso você não queira utilizar os `jobs` automáticos que já vêm como padrão do banco de dados Oracle, você poderá estruturar sua forma de geração de estatísticas montando os comandos de geração de acordo com sua necessidade, com base no que você aprendeu aqui, e utilizar o pacote `dbms_job` para auxiliá-lo na automatização das gerações.

Como já mencionei, as estatísticas ficam armazenadas no dicionário. Contudo, por curiosidade, se você quiser visualizar as estatísticas no dicionário de dados, consulte as seguintes `views` do banco de dados Oracle: `dba_tables`, `dba_object_tables`, `dba_tab_statistics`, `dba_tab_col_statistics`, `dba_tab_histograms`, `dba_indexes`, `dba_ind_statistics`, `dba_clusters`, `dba_tab_partitions`, `dba_tab_subpartitions`, `dba_ind_partitions`, `dba_ind_subpartitions`, `dba_part_col_statistics`, `dba_part_histograms`, `dba_subpart_col_statistics`, `dba_subpart_histograms`. Algumas poderão ser visualizadas através dos prefixos `user` e `all`, já outras somente através do prefixo DBA.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Neste capítulo, vimos as formas para a geração de estatísticas através do pacote `dbms_stats` . Vimos como gerar estatísticas dos objetos do banco de dados e a importância deste trabalho. Verificamos também como podemos automatizar as gerações, bem como formas de configurar e gerar essas estatísticas. Falamos um pouco das opções que o pacote `dbms_stats` oferece para trabalhar com as estatísticas e como isso pode nos auxiliar em vários aspectos. Nos próximos capítulos, abordaremos as ferramentas que utilizam essas estatísticas e nos mostram como o otimizador chegou a tal plano de execução através do uso delas. São ferramentas que nos trazem o plano de execução real e estimado, com base nas estatísticas armazenadas. Veremos também as formas de visualização e quais informações temos à disposição para melhor interpretar o plano de execução, no qual essas estatísticas são a base de tudo, e entender como o otimizador utiliza tais informações. Tudo isso, utilizando os próprios recursos que já vêm com o banco de dados.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 11

HISTOGRAMAS

Quando conversamos sobre índices, falamos que, mesmo existindo um índice para uma determinada coluna de uma tabela, mesmo que ela esteja sendo utilizada pela instrução SQL como predicado de filtro, nem sempre o otimizador vai utilizá-lo para recuperar os dados. Um dos motivos disso pode ser o fato de ele não conhecer como os dados estão realmente distribuídos nesta coluna, levando-nos a crer que não valeria a pena utilizá-lo. Por padrão, o otimizador assume uma distribuição uniforme de linhas para os valores distintos de uma coluna. Os histogramas vêm para ajudar o otimizador nesta tarefa, fornecendo a ele informações mais precisas para tomar melhores decisões.

Histograma é um tipo especial de estatística de coluna que fornece informações mais detalhadas sobre a forma como estão distribuídos os dados em uma coluna de determinada tabela. Anteriormente ao Oracle 9i, o trabalho de detectar quais colunas eram candidatas a ter histogramas era feito de forma manual, através da análise de alguns fatores, como a verificação das colunas mais usadas, nas quais o nível de segmentação dos dados pudessem causar variações bem acentuadas no plano de execução gerando planos bem diferentes dependendo da informação armazenada. A partir da versão 9i, o banco de dados ganhou a capacidade de procurar e detectar colunas que devem ter histogramas e criá-los

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

automaticamente através do pacote `dbms_stats`.

O objetivo de gerarmos histogramas para uma ou mais colunas de uma tabela é podermos prever a cardinalidade e o número de linhas retornadas de seus conjuntos de resultados para predicados de filtro e junções que envolvem essas colunas, o que pode fazer a diferença na escolha ou não pelo uso de um índice. Esses histogramas fornecem estimativas precisas da distribuição dos dados da coluna. Também fornecem estimativas de seletividade mais detalhadas quando na presença de distorção de dados, resultando em planos de execução mais precisos quando há distribuições de dados não uniformes. Esse seria o motivo para utilizarmos histogramas.

Basicamente, os histogramas são utilizados em duas ocasiões pelo otimizador: na ordem da junção de tabelas e nos métodos de acesso à tabela:

- Ordem de junção de tabela (*Table Join Order*): o otimizador terá que saber o tamanho dos conjuntos de resultados intermediários (cardinalidade entre as tabelas) para determinar corretamente a ordem de junção mais adequada para a união de várias tabelas.
- Método de acesso à tabela (*Table Access Method*): o otimizador terá que saber informações sobre as colunas das cláusulas SQL, em que o valor da coluna é distorcido (não uniforme), de modo que uma varredura de tabela completa (*Full Table Scan*) possa ser mais rápida do que uma varredura de índice. O Oracle utiliza essas informações de distorção (desuniformidades) em conjunto com as colunas `clustering_factor` da `view` `dba_indexes`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Se voltarmos um pouco e relembrarmos do conceito de cardinalidade, veremos que existem vários tipos, um dos quais é a **cardinalidade distinta** (*Distinct Cardinality*), que se refere à quantidade de valores distintos em uma coluna. A cardinalidade distinta de um conjunto de linhas é calculada com base nos dados da coluna, ou seja, se existem dados distintos e quantos tipos de dados distintos temos em determinada coluna. Esse é um dos fatores que podem levar o otimizador a escolher fazer acesso através de um índice ou realizar uma varredura completa da tabela (FTS - `FULL TABLE SCAN`).

O outro fator é a seletividade do índice. Quando o otimizador analisa a instrução SQL, ele determina a viabilidade do uso de determinados índices ao verificar se esses índices existem para as tabelas da instrução e se estão sendo utilizados como filtro de dados, por exemplo. Ao encontrar cenários como esse, ele levará em conta o quanto este índice é seletivo. A seletividade de um índice é determinada pelo seu valor de uso, que é baseado na consulta e nos dados. Esse valor é determinado de várias formas, uma delas é pelo número de chaves distintas no índice, através da análise da tabela ou do índice. Por exemplo, quando uma determinada coluna utilizada em uma cláusula `where` tem uma quantidade considerável de valores desproporcionais, para o otimizador, a verificação por FTS poderá ser menos custosa do que o acesso ao índice.

Com esses conceitos em mente e sabendo como o otimizador trabalha por padrão, sem o uso de histogramas, o otimizador fará uma conta considerando a quantidade total das linhas da tabela dividida pelo número de valores distintos (`NDV`). Com base nisso, o otimizador pode achar que não valerá a pena utilizar o índice,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

fazendo um `FULL TABLE SCAN` na tabela em vez de utilizar um determinado índice para a busca. Ou ele pode ver que sim, vale a pena fazer um acesso através daquele índice.

> **Nota:** NDV = Número de valores distintos. O NDV é importante na geração de estimativas de cardinalidade.

Vamos para um exemplo prático. Vamos imaginar que uma pesquisa em nível nacional avaliou a preferência das pessoas com relação ao que elas mais assistem. Veja o resultado da pesquisa a seguir:

```
SQL> select count(*) pessoas, assistem from pesquisa group by ass
istem order by 1 desc;

PESSOAS ASSISTEM
------- -------------------
350000  Series
300000  Filmes
200000  Novelas
150000  Telejornais
    10  Programa do Bolinha

SQL>
```

Agora vamos imaginar que o otimizador de consultas, depois de consultar as informações da cardinalidade distinta da coluna `assistem`, chegasse à mesma conclusão que temos mostrada neste exemplo. O resultado está dividido em cinco conjuntos de dados: `'Series'`, `'Filmes'`, `'Novelas'`, `'Telejornais'` e `'Programa do Bolinha'`. Desta forma, é possível ver que possuímos cinco tipos de dados distintos.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Olhando o mesmo resultado, se pedíssemos para o otimizador que trouxesse as pessoas que assistem o `'Programa do Bolinha'` , levando em conta que há um índice para a coluna `assistem` , na qual faríamos o filtro, provavelmente diríamos que ele usaria tal índice, pois, dada a quantidade de registros que podemos observar nos outros conjuntos de dados, o filtro escolhido traria pouquíssimas ocorrências. Agora, vamos ver qual o plano de execução desta busca:

Antes de executar a instrução SQL, vamos gerar as estatísticas para a tabela `PESQUISA` :

```
SQL> begin
  2    dbms_stats.gather_table_stats( 'TTUN','PESQUISA'
  3                                  ,cascade => true
  4                                  ,estimate_percent => 10
  5                                  ,degree => 4);
  6  end;
  7  /

PL/SQL procedure successfully completed.
```

Agora vamos executar a instrução SQL. Note que, antes disso, executamos o comando `autotrace` para visualizarmos o plano de execução utilizado pelo otimizador logo após a execução.

```
SQL> set autotrace on

SQL> select assistem, idade, genero from pesquisa where assistem
= 'Programa do Bolinha';

ASSISTEM                       IDADE G
------------------------- ---------- -
Programa do Bolinha                20 F
Programa do Bolinha                99 F
Programa do Bolinha                80 F
Programa do Bolinha                63 M
Programa do Bolinha                77 M
Programa do Bolinha                22 F
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Programa do Bolinha                94 F
Programa do Bolinha                81 M
Programa do Bolinha                24 M
Programa do Bolinha                20 F

10 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 2652078278

-------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          | 74975 |   951K|   799   (2)| 00:00:12 |
|*  1 |  TABLE ACCESS FULL| PESQUISA | 74975 |   951K|   799   (2)| 00:00:12 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
1 - filter("ASSISTEM"='Programa do Bolinha')
```

Figura 11.1: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap11img01code.txt

Embora a tabela `pesquisa` tenha 1 milhão e 10 registros, e estejamos apenas solicitando os 10 registros - não esquecendo que estamos utilizando uma coluna indexada -, o otimizador escolheu realizar uma `FTS` nessa tabela em vez de optar pelo índice existente. Seguindo o que vimos até aqui, muito provavelmente a opção por `FTS` deve ter se dado pelo fato de o otimizador não ter achado muito atraente a seletividade do índice. Se olharmos para o cálculo, teremos:

```
Total de linhas da tabela: 1.000.010
Seletividade (NDV): 5
Jogando no cálculo: 1.000.010 / 5 = 200.002
```

Através do resultado do cálculo, o otimizador entende que há aproximadamente 200 mil registros para cada tipo distinto. Nós

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

sabemos que não há 200 mil registros de cada tipo, mas, se fosse o caso, poderíamos até interpretar o cenário da mesma forma como o otimizador: se fosse para ler 20% dos registros de um índice, convinha então ler diretamente a tabela. Contudo, percebemos que a interpretação do otimizador não foi adequada e que, neste caso, o índice poderia ser melhor utilizado se o otimizador conhecesse a quantidade de linhas que cada tipo possui. É neste ponto que o uso de histogramas faz sentido. Como não temos os dados distribuídos de forma uniforme na coluna `assistem`, o otimizador, com as informações que ele tinha, acabou sendo influenciado de uma forma não muito adequada.

Agora, vamos utilizar o mesmo cenário, mas gerando um histograma para a coluna `assitem`, permitindo que o otimizador conheça verdadeiramente como essas linhas estão distribuídas nesses tipos. Para gerar histograma para a coluna `assistem`, vamos utilizar o procedimento `gather_table_stats` do pacote `dbms_stats`, utilizando a opção `method_opt` na chamada:

```
SQL> begin
  2    dbms_stats.gather_table_stats( 'TTUN','PESQUISA'
  3                                  ,estimate_percent => 10
  4                                  ,method_opt => 'for columns
assistem size 10'
  5                                  ,cascade => true
  6                                  ,degree => 4 );
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

Agora vamos refazer a consulta:

```
SQL> select assistem, idade, genero from pesquisa where assistem
= 'Programa do Bolinha';

ASSISTEM                        IDADE G
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
------------------------- ---------- -
Programa do Bolinha                20 F
Programa do Bolinha                99 F
Programa do Bolinha                80 F
Programa do Bolinha                63 M
Programa do Bolinha                77 M
Programa do Bolinha                22 F
Programa do Bolinha                94 F
Programa do Bolinha                81 M
Programa do Bolinha                24 M
Programa do Bolinha                20 F

10 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 3484195155

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    30 |   390 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| PESQUISA    |    30 |   390 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | ASSISTEM_IX |    32 |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
2 - access("ASSISTEM"='Programa do Bolinha')
```

Figura 11.2: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap11img02code.txt

Veja que, para a mesma instrução SQL executada anteriormente, na qual o otimizador optou por um `FULL TABLE SCAN` , agora, depois de criado o histograma para a coluna do índice e a instrução executada mais uma vez, o otimizador acabou optando por utilizar um acesso via índice. Isso porque, ao criar um histograma, geramos estatísticas que demonstram ao otimizador o quão não uniforme são os dados que a coluna possui, e como eles estão distribuídos. Nesse caso, ele pode perceber que o predicado "Programa do Bolinha" representa apenas 0,0009999900001% do

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

total de registros e, por isso, vale a pena realizar uma varredura pelo índice.

> **NOTA:** Nesse exemplo, em que estamos falando de uma quantidade de dados enorme sendo varrida, se o índice não fosse utilizado, dados como os de acesso à memória (`CONSISTENT GETS`) e leituras físicas de disco (`PHYSICAL READS`) poderiam ser muito diferentes e até bem elevados, mudando de forma bem significativa o cenário com relação ao desempenho.

11.1 GERANDO HISTOGRAMAS

Os histogramas são gerados através do pacote `dbms_stats` quando utilizamos a opção do uso do parâmetro `method_opt` na chamada da procedure. A Oracle recomenda utilizarmos o valor `'for all columns size auto'` para esse parâmetro. Através desta opção, o Oracle determinará automaticamente quais colunas são melhores candidatas para ter histogramas e o número dos chamados `buckets` para cada histograma criado. Contudo, mais à frente, veremos a importância de analisar alguns fatores antes de decidirmos optar pelo modo automático. Veja a seguir um exemplo de geração de histograma, mas agora utilizando a opção configurada para o modo automático.

```
SQL> begin
2    dbms_stats.gather_table_stats( 'TTUN','PESQUISA’
3                                   ,method_opt => 'for all colum
ns size auto’
4                                   ,cascade => true
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
5                                   ,degree => 4 );
6  end;
7  /

PL/SQL procedure successfully completed.
SQL>
```

Quando não utilizamos o modo automático, precisamos definir para qual coluna desejamos que o histograma seja criado e qual deverá ser o seu tamanho. Veja um exemplo a seguir, o mesmo já demonstrado anteriormente.

```
SQL> begin
  2    dbms_stats.gather_table_stats( 'TTUN','PESQUISA'
  3                                  ,estimate_percent => 10
  4                                  ,method_opt => 'for columns
assistem size 10'
  5                                  ,cascade => true
  6                                  ,degree => 4 );
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

Aqui estamos criando um histograma para a coluna `assistem` e estamos dizendo ao Oracle que o tamanho do histograma será de 10 buckets.

Para criar histogramas para várias colunas ao mesmo tempo, informe no mesmo parâmetro as colunas desejadas. Veja o exemplo dos histogramas sendo criados para as colunas `assistem` e `IDADE`, onde são informados os nomes das colunas, com os respectivos tamanhos de histograma desejados.

```
SQL> begin
  2    dbms_stats.gather_table_stats( 'TTUN','PESQUISA'
  3                                   ,method_opt => 'for columns
 assistem size 10, idade size 254'
  4                                   ,cascade => true
  5                                   ,degree => 4 );
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  6  end;
  7  /

PL/SQL procedure successfully completed.
SQL>
```

Buckets são recipientes que o banco de dados Oracle destina para guardar, de forma organizada, todas as informações pertencentes aos grupos de valores distintos. O número de buckets pode ser definido pelo número de valores distintos. Contudo, não é bom definir o `size` manualmente, pois, caso entre um valor distinto na jogada, pode haver distorções. Portanto, é bom prever um aumento na quantidade de dados distintos futuramente ou deixar que o Oracle defina este tamanho de forma automática. Assim, ao entrar um valor novo, o Oracle poderá definir um bucket novo para ele.

Vamos passar rapidamente pela estrutura utilizada para gerar um histograma a partir do parâmetro `method_opt`. Basicamente, a estrutura permite que você informe quais colunas terão histogramas criados ou se será para todas as colunas de uma só vez, e como o Oracle deverá calcular os tamanhos dos histogramas. Veja a sintaxe:

```
for all [indexed|hidden] columns size [size_value]

for columns column size [size_value] [,column size [size_value]…]
```

Nela, temos:

- `for all columns`: este valor é o mais utilizado entre as gerações. Ele é o padrão para a geração de estatísticas e é utilizado para gerar estatísticas para todas as colunas da tabela.
- `for all indexed columns`: esta opção é para configurar

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

a geração das estatísticas somente para as colunas que possuírem índice atrelado a elas.

- `for all hidden columns` : esta opção permite coletar estatísticas somente para as colunas virtuais.
- `for columns` : esta opção é para quando você for especificar as colunas para as quais deseja que o Oracle gere histogramas.

Agora vamos aos tamanhos. Este valor é definido pela diretriz `size` e pode ser configurado da seguinte forma:

```
size {integer | repeat | auto | skewonly}
```

Nessa sintaxe:

- `integer` : número de buckets do histograma. Isso pode ser definido em um intervalo entre 1 e 254 buckets.
- `repeat` : gera histogramas apenas para colunas que já possuem histogramas. Esta opção ocupará menos recursos, pois apenas analisará novamente os histogramas existentes. É uma forma de apenas atualizar os histogramas existentes.
- `auto` : escolhe as colunas para gerar histogramas com base na distribuição de dados e carga de trabalho da coluna. Nesse caso, o Oracle determina o melhor número de buckets que ele achar necessário.
- `skewonly` : escolhe as colunas para gerar histogramas com base na distribuição de dados das colunas. O Oracle criará, de forma automática, histogramas em colunas que possuam maior incidência de distorção de dados. Esta opção é bastante interessante e pode ser uma alternativa mais conservadora em relação à opção `auto` , principalmente se você está começando a trabalhar com

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

histogramas.

Veja alguns exemplos de execução utilizando as combinações das quais falamos.

Utilizando a opção `'for all columns size skewonly'`:

```
SQL> begin
   2   dbms_stats.gather_schema_stats(ownname          => 'TTUN',

   3                                  estimate_percent => dbms_st
ats.auto_sample_size,
   4                                  method_opt       => 'for al
l columns size skewonly',
   5                                  degree           => 7);
   6  end;
   7  /

PL/SQL procedure successfully completed.
SQL>
```

Utilizando a opção `'for all columns size auto'`:

```
SQL> begin
 2     dbms_stats.gather_schema_stats(ownname          => 'TTUN',

 3                                    estimate_percent => dbms_st
ats.auto_sample_size,
 4                                    method_opt       => 'for al
l columns size auto',
 5                                    degree           => 7);
 6  end;
 7  /

PL/SQL procedure successfully completed.
SQL>
```

Utilizando a opção `'for all columns size repeat'`:

```
SQL> begin
 2     dbms_stats.gather_schema_stats(ownname          => 'TTUN',

 3                                    estimate_percent => dbms_st
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
ats.auto_sample_size,
 4                                    method_opt       => 'for al
l columns size repeat',
 5                                    degree           => 7);
 6  end;
 7  /

PL/SQL procedure successfully completed.
SQL>
```

11.2 QUANDO E COMO UTILIZAR OU NÃO HISTOGRAMAS?

Quando utilizamos a opção por geração automática do histograma pelo Oracle, o que é recomendado pela Oracle (usando a opção `'for all columns size auto'`), serão automaticamente determinadas as colunas potencialmente interessantes para terem histogramas, como já foi mencionado anteriormente, com base nas informações que o banco coleta da `dbms_stats.report_col_usage` . Para isso, são analisadas as informações segundo as instruções SQL que envolvem o objeto (`library cache`), levando em consideração situações em que tais colunas estão presentes em cláusulas `where` , sendo igualadas ou utilizadas como filtros por faixa de valores (`range values`), sem contar, é claro, o nível de uniformidade ou distinção de dados nas colunas. O Oracle também levará em conta a análise das chaves estrangeiras para determinar a cardinalidade de um resultado proveniente de uma junção de tabelas. A recomendação da geração de histogramas de forma automática vem a partir do Oracle 11g em diante, pois a ideia é aproveitar o recurso *Adaptive Cursor Sharing*.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

NOTA: `Adaptive Cursor Sharing` é um recurso disponível a partir da versão 11g do Oracle, com o qual é possível que o otimizador compare a eficácia dos planos de execução entre execuções com diferentes valores de variáveis `bind`. Quando uma instrução SQL utiliza uma variável `bind` em igualdade com uma coluna, pelo fato de o otimizador talvez já ter executado essa mesma instrução SQL anteriormente, ele será inclinado a utilizar o mesmo plano de execução. Contudo, quando estamos falando de colunas com dados não uniformes, isso pode acarretar em uma má escolha. Dessa forma, através do `Adaptive Cursor Sharing`, o otimizador pode comparar várias execuções anteriores, nas quais outros valores de `bind` tiveram outros resultados.

O processo funciona mais ou menos da seguinte forma:

- Você executa `dbms_stats` para gerar estatísticas de uma determinada tabela utilizando o parâmetro `method_opt` definido como `size auto`, o padrão remendado pela Oracle.
- Depois, uma instrução SQL, utilizando esta tabela, é executada por você no banco de dados.
- O banco de dados anota os predicados da instrução SQL e atualiza a tabela do dicionário de dados `sys.col_usage$`.
- Logo após, você executa o `dbms_stats` novamente, fazendo com que ele consulte a `sys.col_usage$` para determinar quais colunas requerem histogramas, com base na carga de trabalho da instrução anterior.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Como consequências da geração automática de histogramas (usando a opção `size auto`) temos:

- À medida que as instruções SQL mudam com o passar do tempo, o `dbms_stats` pode alterar as estatísticas geradas. Por exemplo, mesmo que os dados em uma tabela não sofram alterações, as instruções SQL e as operações `dbms_stats` poderão fazer com que os planos de execução para tais instruções, que fazem referência a essas tabelas, sejam alterados.
- Se você gerar estatísticas para uma determinada tabela e não a executar em alguma instrução SQL, o banco de dados não criará histogramas para colunas desta tabela. Para o banco de dados criar os histogramas de forma automática, é necessário executar uma ou mais instruções SQL para que as informações de uso da coluna possam ser preenchidas na `sys.col_usage$` .

No caso de colunas com histograma que estão sendo utilizadas em instruções SQL, sendo colocadas em igualdade ou em range com variáveis `bind` (variável de ligação), o otimizador poderá substituir estes valores de `bind` pelos valores literais, ou consultar outras execuções anteriores para produzir uma análise mais detalhada, a fim de escolher o melhor plano para a instrução. Contudo, ele poderá levar mais tempo para analisar os histogramas.

Vamos ver um exemplo na prática do uso da `sys.col_usage$` pelo banco de dados, e como isso reflete na geração de histogramas quando estamos utilizando a configuração automática (`size auto`):

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Para verificar as informações da `sys.col_usage$` , podemos utilizar a instrução SQL:

```
SQL> select r.name owner
2        ,o.name table1
3        ,c.name column1
4        ,equality_preds
5        ,equijoin_preds
6        ,nonequijoin_preds
7        ,range_preds
8        ,like_preds
9        ,null_preds
10        ,timestamp
11  from   sys.col_usage$ u
12        ,sys.obj$       o
13        ,sys.col$       c
14        ,sys.user$      r
15  where  o.obj#   = u.obj#
16  and    c.obj#   = u.obj#
17  and    c.col#   = u.intcol#
18  and    o.owner# = r.user#
19  and    r.name   = 'TTUN'
20  and    o.name   = 'PESQUISA';

no rows selected

SQL>
```

Note que não temos nenhum registro na `sys.col_usage$` referente à tabela `PESQUISA` . Se já existissem valores, poderíamos utilizar o comando a seguir para excluir esses dados. Contudo, não recomendo a execução deste comando em um ambiente de produção, principalmente sem as orientações e permissões do administrador do banco de dados.

```
SQL> delete sys.col_usage$ where obj# in (select object_id from a
ll_objects where object_name = 'PESQUISA');
4 rows deleted

SQL> commit;
Commit complete
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Podemos verificar também as informações sobre as estatísticas referentes aos histogramas da tabela `PESQUISA` :

```
SQL> select table_name
2        ,column_name
3        ,histogram
4        ,num_distinct
5        ,num_buckets
6        ,sample_size
7        ,last_analyzed
8  from  all_tab_col_statistics
9  where  table_name = 'PESQUISA'
10  order  by 1,2;

no rows selected

SQL>
```

Aqui também não há informações sobre as estatísticas da tabela `PESQUISA` . Se já existissem estatísticas geradas, poderíamos utilizar o comando adiante para excluí-las:

```
SQL> exec dbms_stats.delete_table_stats('TTUN', 'PESQUISA');
PL/SQL procedure successfully completed

SQL>
```

Podemos verificar também a tabela `user_histograms` para confirmar a existência ou não de dados de histogramas para a tabela `PESQUISA` .

```
SQL> select column_name
2        ,endpoint_number
3        ,endpoint_value,endpoint_number - nvl(lag(endpoint_numbe
r) over (order by endpoint_value),0) qdt_rows
4  from   all_histograms
5  where  table_name = 'PESQUISA'
6  and    column_name = 'ASSISTEM'
7  order  by 1;

no rows selected
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL>
```

Aqui, também podemos ver que não há informações referentes a histogramas para a tabela `PESQUISA` .

Agora vamos gerar as estatísticas para tabela `PESQUISA` e utilizar a opção de gerar histogramas de forma automática, deixando que o Oracle decida qual a melhor maneira de realizar isso.

```
SQL> begin
2    dbms_stats.gather_table_stats( 'TTUN'
3                                   ,'PESQUISA'
4                                   ,method_opt => 'for all colum
ns size auto'
5                                   ,cascade => true
6                                   ,degree => 4
7                                   ,estimate_percent => 100
8                                   );
9  end;
10  /

PL/SQL procedure successfully completed.

SQL>
```

Iniciamos por gerar as estatísticas para a tabela `PESQUISA` com a opção de gerar histogramas de forma automática. Na parte dos histogramas, configuramos para que o Oracle realize a melhor escolha dos tamanhos do buckets, com base no nível de uniformidade que encontrar.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Veja que para essa execução utilizamos o parâmetro `estimate_percent` igual a 100. Isso indica que o Oracle deve analisar 100% dos dados da tabela. Como queremos realizar algumas verificações e comparações, vamos deixá-lo computar todos os registros para que possamos ver as estatísticas e compará-las com outros resultados. Se não informarmos o parâmetro `estimate_percent`, o Oracle acaba computando apenas uma amostra dos registros, o que para nós, para efeito de exemplo, não facilitaria entender alguns resultados que vamos mostrar mais adiante. Contudo, essa amostragem que o Oracle realiza em nada desabona seu funcionamento e pode ser usada normalmente, principalmente se estivermos falando de um banco de dados de grande porte. Nesse exemplo, essa amostragem foi utilizada para exemplificar e tornar os resultados mostrados mais fáceis de serem entendidos.

Agora, consultando novamente os dados de estatística e histogramas para a tabela.

```
SQL> select table_name
2          ,column_name
3          ,histogram
4          ,num_distinct
5          ,num_buckets
6          ,sample_size
7          ,last_analyzed
8    from   all_tab_col_statistics
9    where  table_name = 'PESQUISA'
10   order  by 1,2;

TABLE_NAME      COLUMN_NAME      HISTOGRAM  NUM_DISTINCT NUM_BUCK
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
ETS SAMPLE_SIZE LAST_ANAL
--------------- ----------- --------------- ------------ --------
--- ----------- ---------
PESQUISA        ASSISTEM                NONE            5
  1     1000010 16-JAN-19
PESQUISA        CODIGO                  NONE       350000
  1     1000010 16-JAN-19
PESQUISA        GENERO                  NONE            2
  1     1000010 16-JAN-19
PESQUISA        IDADE                   NONE           84
  1     1000010 16-JAN-19

SQL>
```

Note que o Oracle criou as estatísticas para as colunas, mas em nenhuma delas ele gerou informações de histograma, coerentes com a realidade. Isso pode ser visto pela coluna `histogram`, na qual, para todas as linhas, aparece o valor `NONE`, e também pelo número de buckets representado. Agora vamos consultar a tabela de histogramas, porém, para que a instrução SQL não gere distorções, vamos consultar os dados apenas da coluna `ASSISTEM`. É possível pesquisar as outras colunas separadamente para verificar as informações de cada uma delas.

```
SQL> select column_name
2        ,endpoint_number
3        ,endpoint_value
4        ,endpoint_number - nvl(lag(endpoint_number) over (order
by endpoint_value),0) qdt_rows
5  from   all_histograms
6  where  table_name = 'PESQUISA'
7  and    column_name = 'ASSISTEM'
8  order  by 1;

COLUMN_NAME          ENDPOINT_NUMBER          ENDPOINT_VALUE QDT_
ROWS
-------------------- --------------- ----------------------- ----
----
ASSISTEM                           0     3,65599023603884E35
   0
ASSISTEM                           1     4,38210047515104E35
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  1

SQL>
```

Aqui é possível ver que o Oracle criou referências de histograma para a coluna `ASSISTEM` , mas veja que as informações de bucket e quantidade de linhas estão totalmente fora da realidade com base nas informações que já conhecemos sobre a tabela e suas colunas.

Agora podemos verificar na `sys.col_usage$` para ver como estão os dados para a tabela `PESQUISA` .

```
SQL> select r.name owner
2        ,o.name table1
3        ,c.name column1
4        ,equality_preds
5        ,equijoin_preds
6        ,nonequijoin_preds
7        ,range_preds
8        ,like_preds
9        ,null_preds
10        ,timestamp
11  from   sys.col_usage$ u
12        ,sys.obj$       o
13        ,sys.col$       c
14        ,sys.user$      r
15  where  o.obj#   = u.obj#
16  and    c.obj#   = u.obj#
17  and    c.col#   = u.intcol#
18  and    o.owner# = r.user#
19  and    r.name   = 'TTUN'
20  and    o.name   = 'PESQUISA';

no rows selected

SQL>
```

Como não poderia ser diferente, ainda não há dados de uso para esta tabela. Então, agora executando algumas instruções SQL

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

em cima da tabela e das colunas candidatas a histogramas, e depois voltando a consultar os dados, será possível ver que históricos foram gerados. Veja a seguir a sequência de execuções.

```
SQL> select count(*) from pesquisa where assistem like 'Programa%
';

COUNT(*)
--------
10

SQL> select count(*) from pesquisa where genero = 'F';

COUNT(*)
--------
499973

SQL> select count(*) from pesquisa where idade between 20 and 30;


COUNT(*)
--------
130702

SQL> select count(*) from pesquisa where codigo = 1000;

COUNT(*)
--------
4

SQL>
```

Na primeira instrução SQL, está sendo solicitada a quantidade de registros que começam com a string "Programa" na coluna `ASSISTEM`. Para isso, a função `LIKE` foi utilizada. Já na segunda instrução, assim como nas demais, foi solicitada a quantidade de registros, sobretudo de registros com a string F na coluna `GENERO` (F referente ao gênero Feminino). Na terceira instrução, na coluna `IDADE`, que traga registros que tenham dados de pessoas entre 20 e 30 anos, e, na quarta instrução, registros que tenham o valor

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

igual a 1000, na coluna `CODIGO` . Executados esses `SELECTs` , é possível verificar todas as informações, novamente.

Primeiramente, deve-se consultar a `sys.col_usage$` , mas não se pode esquecer de primeiro executar o comando `flush_database_monitoring_info` .

```
SQL> begin
2      dbms_stats.flush_database_monitoring_info;
3    end;
4    /

PL/SQL procedure successfully completed.

SQL> select c.name column1
2          ,equality_preds
3          ,equijoin_preds
4          ,nonequijoin_preds
5          ,range_preds
6          ,like_preds
7          ,null_preds
8  from     sys.col_usage$ u
9          ,sys.obj$       o
10         ,sys.col$       c
11         ,sys.user$      r
12  where   o.obj#   = u.obj#
13  and     c.obj#   = u.obj#
14  and     c.col#   = u.intcol#
15  and     o.owner# = r.user#
16  and     r.name   = 'TTUN'
17  and     o.name   = 'PESQUISA';

COLUMN1   EQUALITY_PREDS EQUIJOIN_PREDS NONEQUIJOIN_PREDS RANGE_P
REDS LIKE_PREDS NULL_PREDS
--------- -------------- -------------- ----------------- -------
---- ---------- ----------
CODIGO                 1              0              0
   0          0          0
ASSISTEM               0              0              0
   0          1          0
IDADE                  0              0              0
   1          0          0
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
GENERO                  1              0              0
   0          0          0

SQL>
```

Aqui já é possível ver que os dados já foram gerados para as colunas da tabela `PESQUISA`, com base na execução das instruções SQL executadas anteriormente. Através dos resultados desta instrução, em que são trazidas as informações sobre cada tipo de predicado monitorado, por exemplo, `EQUALITY_PREDS`, `EQUIJOIN_PREDS` etc., podemos ver o tipo de operação que foi realizada em cima de cada coluna. Na coluna `CODIGO`, por exemplo, há um registro de predicado de igualdade, `EQUALITY_PREDS (referente ao predicado usado, CODIGO = 1000)`. Já para a coluna `ASSISTEM`, vemos um registro de predicado do uso do `like`, `LIKE_PREDS (LIKE 'Programa%')`.

> **Nota:** As informações de DML são preenchidas apenas para tabelas com o atributo `monitoring`. Por motivos de desempenho, o banco de dados Oracle não preenche essas informações imediatamente quando ocorrem as modificações reais de DML. Para visualizá-las, execute o procedimento `flush_database_monitoring_info` no pacote `dbms_stats` para o preenchimento e exibição das informações mais recentes. O privilégio do sistema `analyze_any` é necessário para executar este procedimento.

Agora, com base nessas novas informações, é possível ver como o Oracle atualizou as estatísticas das colunas da tabela. Mas antes, é necessário atualizar as estatísticas, caso contrário, as

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

informações não serão atualizadas na base de dados.

```
SQL> begin
2    dbms_stats.gather_table_stats( 'TTUN'
3                                   ,'PESQUISA'
4                                   ,method_opt => 'for all colum
ns size auto'
5                                   ,cascade => true
6                                   ,degree => 4
7                                   ,estimate_percent => 100
8                                   );
9  end;
10  /

PL/SQL procedure successfully completed.

SQL>
```

Consultando as informações das colunas da tabela, temos:

```
SQL> select column_name
2          ,histogram
3          ,num_distinct
4          ,num_buckets
5          ,sample_size
6          ,last_analyzed
7    from   all_tab_col_statistics
8    where  table_name = 'PESQUISA'
9    order  by 1,2;

COLUMN_NAME         HISTOGRAM NUM_DISTINCT NUM_BUCKETS SAMPLE_SIZ
E LAST_ANAL
------------- --------------- ------------ ----------- ----------
- ---------
ASSISTEM            FREQUENCY            5           5     100001
0 18-JAN-19
CODIGO                   NONE       350000           1     100001
0 18-JAN-19
GENERO              FREQUENCY            2           2     100001
0 18-JAN-19
IDADE               FREQUENCY           84          84     100001
0 18-JAN-19

SQL>
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Veja que interessante, o Oracle atualizou os dados de histograma de três colunas das quatro existentes. Vamos analisar:

- Para a coluna `ASSISTEM`, foi gerado um histograma do tipo `FRENQUENCY` (Histograma de Frequência). Nele, para 5 tipos de informações distintas, o Oracle criou 5 `buckets`.
- Para a coluna `GENERO`, foi gerado também um histograma do tipo `FRENQUENCY`. Nele, para 2 tipos de informações distintas, o Oracle criou 2 `buckets`.
- Para coluna `IDADE`, foi gerado um histograma do tipo `FRENQUENCY`. Nele, para 84 tipos de informações distintas, o Oracle criou 84 `buckets`.
- A coluna `CODIGO`, embora tenha tido histórico de uso, não foi atualizada. Muito provavelmente pelo seu alto nível de desuniformidade.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Existem três tipos de histogramas que podem ser escolhidos pelo banco de dados. Os três tipos de histogramas são os seguintes: Histogramas do tipo `Frequency` (e `Top Frequency`), `Histogramas Height-Balanced` e `Histogramas Hybrid` . Com base no NDV e na distribuição dos dados, o banco de dados escolhe o tipo de histograma mais adequado a ser criado com base nas informações que ele colher sobre a coluna ou como ela é utilizada. Em alguns casos, ao criar um histograma, o banco de dados faz uma amostragem de um número de linhas predeterminado internamente. Para maiores detalhes de como cada tipo funciona internamente e dos critérios de escolha de cada tipo pelo banco de dados, consulte a documentação do banco de dados. Nas referências bibliográficas deste livro, há links para a documentação.

Consultando as informações de histogramas para tabela `PESQUISA` e verificando a coluna `ASSISTEM` , temos:

```
SQL> select column_name
2        ,endpoint_number
3        ,endpoint_value
4        ,endpoint_number - nvl(lag(endpoint_number) over (order
by endpoint_value),0) qdt_rows
5  from   all_histograms
6  where  table_name = 'PESQUISA'
7  and    column_name = 'ASSISTEM'
8  order  by 1;

COLUMN_NAME          ENDPOINT_NUMBER ENDPOINT_VALUE   QDT_ROWS
-------------------- --------------- -------------- ______
ASSISTEM                      300000     3.6560E+35     300000
ASSISTEM                      500000     4.0726E+35     200000
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
ASSISTEM                      500010     4.1770E+35         10
ASSISTEM                      850010     4.3302E+35     350000
ASSISTEM                     1000010     4.3821E+35     150000

SQL>
```

Ao contrário da visão anterior, agora o Oracle atualizou os histogramas com dados referentes aos 5 `buckets` criados por ele. Veja também a quantidade de linhas inseridas em cada `bucket`, sendo que no primeiro `bucket` há 300.000 registros, no segundo, 200.000 registros, no terceiro, 10 registros, no quarto, mais 350.000 e, no último, um bucket com 150.000 registros. Somando todos os registros, chegamos à soma de 1.000.010 registros, fechando com a quantidade de registros da tabela.

Dessa forma é possível ver como o método automático funciona e como, à medida que vamos executando instruções SQL em cima de tais tabelas, isso influencia na forma como esses histogramas vão sendo reavaliados e suas estatísticas atualizadas.

Nota: Uma observação importante é que as informações aqui mostradas, através das consultas às tabelas `all_histograms` e `all_tab_col_statistics`, também podem ser visualizadas mesmo não utilizando o método automático. No entanto, nesses casos, essas informações não estarão sofrendo influência da `sys.col_usage$`.

No caso da geração manual, atente-se para não realizar de forma inadequada o trabalho de geração, por exemplo, executando a geração para todas as colunas e ainda utilizando o tamanho máximo de buckets. Isso pode levar um tempo de processamento

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

elevado e exigir mais espaço de armazenamento e recursos do sistema caso o banco de dados seja de grande porte. Além disso, você provavelmente dificultará a tomada de decisão pelo otimizador ao fornecer a ele muitas informações desnecessárias, mas que, pelo fato de existirem, farão com que ele as considere, e também o confundirá na hora de decidir utilizá-las ou não.

De qualquer forma, se decidir utilizar a geração manual, via de regra, crie histogramas para os casos em que as colunas são referenciadas em instruções SQL, evitando colunas que nunca são utilizadas como filtros nas instruções. Lembre-se de que o que determina se uma coluna é candidata a ter um histograma é a carga de trabalho que existirá nela, ou seja, se ela é um fator importante na seleção dos dados, não para a lógica do negócio, mas sim, na forma como as informações são recuperadas.

Devemos ter cuidado na geração de histogramas, apenas gerando para colunas de índices (`method = 'for all indexed columnss'`). É pouco provável que, para todas as instruções SQL que são executadas no banco de dados, utilizemos apenas colunas indexadas como uso de predicados para realizar filtros de dados. Nesses casos, apenas favoreceríamos a instrução SQL onde tais colunas indexadas fossem utilizadas. Associado a este método, se utilizarmos o `size` igual a `skewonly` , isto poderá levar muito tempo, pois será analisada toda a distribuição dos valores de cada coluna de cada índice. Se na geração de histogramas forem encontradas colunas com dados desuniformes, isso ajudará o otimizador no caso de existir um histograma. Caso contrário, dados desnecessários serão gerados.

Quando criamos histogramas de forma manual, devemos

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

prestar atenção quanto a se as colunas escolhidas são muito utilizadas com valores passados através de variáveis `bind` . Nesses casos, a verificação do otimizador será mais custosa aos recursos do banco de dados, pois ele pode chegar a um nível de realizar comparações com valores literais (o valor contido na `bind` em si), verificando com outras execuções e histogramas, sem falar que a Library Cache poderá ficar abarrotada de informações, indicando a necessidade de mais armazenamento. Vale ressaltar que um histograma só é indicado para colunas que possuem uma distorção de dados considerável, conforme já foi comentado.

Como já vimos anteriormente, os histogramas são utilizados não só na questão da decisão por um índice ou não, pelo otimizador, mas também são mais comumente usados para prever o tamanho do conjunto de resultados (`Result Set`) intermediários de uma junção de tabelas. Isso nos leva para um ponto de atenção: o de como seria trabalhoso - e diria até inviável - realizar uma análise para determinar a escolha de colunas candidatas para histograma, dependendo da complexidade e tamanho de um sistema, tendo em vista a vasta quantidade de ligações que poderiam existir entre essas tabelas, dadas as mais variadas regras de negócio existentes. O que quero dizer com isso é que devemos levar em conta a complexidade de um esquema, onde podem existir dezenas de milhares de colunas e, com isso, pode ser impraticável avaliar cada coluna quanto a seu desvio de distorção, ou o seu nível de relacionamento com outras tabelas, o que justificaria o uso da opção `auto` , recomendada pela Oracle.

Vamos a um exemplo: suponhamos que temos uma instrução SQL na qual existem várias tabelas ligadas por suas junções. Nessas tabelas, há vários registros, centenas ou até milhares deles, mas no

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

final, apenas algumas linhas serão retornadas como resultado, como 30 linhas ou um pouco menos, por exemplo, na maioria das vezes. Via de regra, o otimizador vai unir essas tabelas de forma a gerar um conjunto de resultados menor já na primeira junção, para evitar ficar transportando grande quantidade de dados entre as junções intermediárias, tornando mais rápida a execução da instrução SQL. Para diminuir os resultados intermediários, ou seja, aqueles que antecedem ao resultado final e são frutos das junções que vão acontecendo, o otimizador tentará estimar a cardinalidade de cada conjunto de resultados retornados durante a fase de análise de toda a execução da instrução SQL. Nesse caso, ter histogramas nas colunas que possuem significativa distorção de valores poderá ajudar consideravelmente o otimizador a tomar decisões mais adequadas.

Em resumo, os histogramas deveriam ser criados:

- Quando uma coluna está sendo referenciada, de forma significativa, em instruções SQL;
- Quando uma coluna possui um alto nível de distorção de dados (dados não uniformes);
- Quando uma coluna, devido à sua distorção, está influenciando o otimizador de forma incorreta;
- Quando uma coluna está sendo utilizada em junções de tabelas que tenham um alto nível de relacionamento, significativo o bastante, para que o histograma venha a trazer ganhos ao otimizador.

Neste capítulo, vimos o que são e como trabalhar com histogramas como uma forma de ter mais um recurso que ajude o otimizador a conhecer melhor os dados das colunas das tabelas e,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

assim, melhor definir os passos para a geração dos planos de execução.

No próximo capítulo, abordaremos as ferramentas que podemos utilizar para visualizar os planos de execução de instruções SQL, que são baseados e gerados levando em consideração as estatísticas que geramos para os objetos do banco de dados e, consequentemente, os objetos utilizados nestas instruções. Veremos ferramentas que nos trazem o plano de execução real e estimado. Vamos ver também as formas de visualizar esses planos e quais informações temos à disposição para melhor interpretá-los e entender como o otimizador utiliza tais informações. Tudo isso utilizando os próprios recursos que já vêm com o banco de dados.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

CAPÍTULO 12

GERANDO E VISUALIZANDO PLANOS DE EXECUÇÃO

No capítulo 9 foi abordado o que é o plano de execução e como interpretá-lo. Neste capítulo, será mostrado como os planos de execução podem ser gerados e visualizados de diferentes formas e por mais de uma ferramenta.

A Oracle disponibiliza, dentro do próprio banco de dados, uma série de ferramentas, visões e outros recursos, como `explain plan`, `autotrace`, `trace`, `tkprof` e `dbms_xplan`, e os recursos `gather_plan_statistics`, `driving_site` e `v$sql_plan` proporcionam uma gama de informações úteis para a análise da performance de instruções SQL e de programas PL/SQL. As técnicas para *tuning* de aplicação podem ser aplicadas utilizando essas ferramentas a fim de tornar mais clara e objetiva a visão do problema em questão.

Existem ferramentas que já vêm desde versões mais antigas, que ainda são utilizadas, mas também novas ferramentas que foram sendo incluídas a partir de novas versões. A Oracle procura sempre manter todos os recursos mais antigos compatíveis com as

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

novas versões de banco e dados, evitando que programas escritos em versões anteriores venham a causar problemas de mau funcionamento.

Existem outras ferramentas de terceiros, algumas delas até melhores, com apelo visual, com as quais são mostrados gráficos e dados formatados no estilo relatório com vários indicadores e informações estatísticas. Sem contar que elas trazem uma série de funções automatizadas ao clique de um botão. Essas ferramentas são muito interessantes, principalmente pela produtividade que elas apresentam, pois não é necessário fazer todo o trabalho na mão. No entanto, a proposta é trabalharmos com os recursos que o próprio banco de dados Oracle disponibiliza em seu ambiente padrão, para executar os comandos e programas, e visualizar as informações dos planos de execução. Desta forma, utilizaremos o SQL*Plus, que está presente em toda instalação do banco de dados Oracle e disponível em praticamente 100% dos ambientes.

12.1 SQL*PLUS

A SQL*Plus acompanha o banco de dados Oracle quando ele é instalado. Ela é quem faz a interface entre o usuário e o banco de dados. Embora ela seja bem simples e sem muitos adornos visuais, será bem útil para mostrar os conceitos sem termos que nos preocupar com detalhes de instalação, parametrização, licenças e outros recursos.

Ao executar a ferramenta `SQL*Plus`, é apresentada uma tela de logon na qual devemos informar o nome do usuário, a senha e a string de conexão com o banco de dados. Em algumas situações, pode-se ter a versão gráfica do programa instalada no ambiente,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

mas na maioria das vezes, somente a versão via `prompt` de comando estará disponível, principalmente, se o acesso ao banco for apenas por algum servidor corporativo, ou pelo fato de a instalação do banco de dados não comportar a versão gráfica. Ambas possuem os mesmos recursos. A seguir, veja o programa SQL*Plus sendo executado pelo `prompt` de comando do Windows.

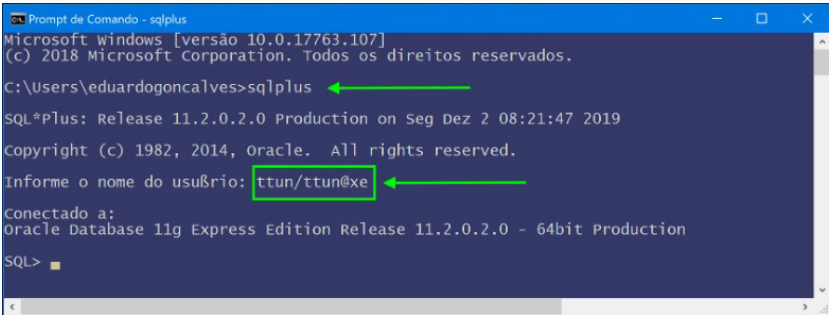

Figura 12.1: Conexão com SQL*Plus via Prompt de Comando.

Após a abertura do `prompt` de comando do Windows, foi digitado o comando `sqlplus` para a abertura do programa. Logo após serem apresentadas informações sobre o programa SQL*Plus, como versão, data e hora da abertura e outras versões, o programa pede para informar as credenciais de acesso. Depois de conectado, o cursor é direcionado para o `prompt` de comando do programa, onde ele estará aguardando por novas instruções.

A finalidade geral da ferramenta SQL*Plus é a execução de instrução SQL, código PL/SQL, procedimentos (`procedures`), gatilhos (`triggers`) e funções (`functions`), além da execução de comandos e ferramentas relacionadas à administração do banco de dados.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Como a maioria das ferramentas, o SQL*Plus possui diversas configurações de visualização de instruções e resultados, como formatação de linhas, colunas, páginas, comandos para mostrar informações sobre a execução do código, entre outras. Além disso, possui uma série de comandos que são úteis para a manipulação dessas instruções e para a formatação das informações apresentadas:

- Editar a instrução SQL armazenada no `buffer` ;
- Formatar os resultados retornados pelo banco de dados;
- Armazenar as instruções SQL em arquivos no disco e recuperá-los para execução;
- Modificar o modo de trabalhar do SQL*Plus;
- Enviar mensagens e receber respostas de outros usuários;
- Listar a definição de qualquer tabela;
- Fazer acesso e copiar dados entre bancos de dados;
- Realizar procedimentos de manutenção do banco de dados;
- Executar programas e ferramentas para o monitoramento e análise da performance.

Na tela a seguir, temos um exemplo de uma instrução digitada e executada no SQL*Plus:

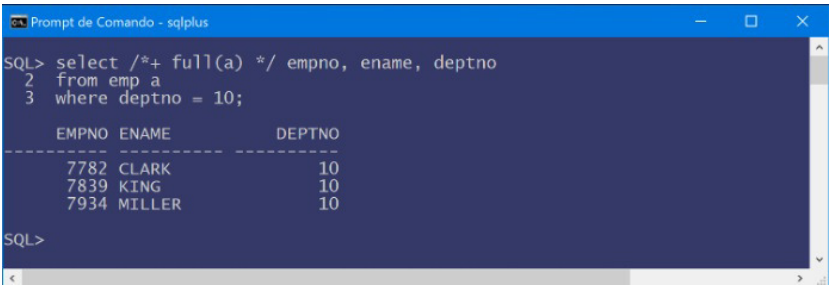

Figura 12.2: Exemplo de uma execução de instrução SQL no SQL*Plus

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

O objetivo não é ensinar o uso desta ferramenta, para isso, recomendo consultar outros materiais ou os próprios manuais da Oracle. Aqui vamos utilizar esta ferramenta não só com o intuito de executar as instruções SQL, mas sim, para auxiliar na execução das ferramentas que utilizaremos para analisar e monitorar a performance, obtendo informações referentes às aplicações do banco de dados, nesse sentido.

Agora vamos aprender um pouco sobre como extrair e visualizar as informações que precisamos para esse trabalho.

12.2 EXPLAIN PLAN

O `explain plan` da Oracle é um utilitário que mostra o plano de execução referente às instruções SQL executadas no banco de dados. Você vai ver mais adiante, quando falarmos sobre o `tkprof`, que podemos parametrizá-lo para que ele seja executado também quando extrairmos o plano de execução do arquivo de trace. Ele pode ser executado separadamente direto pelo SQL*PLUS, armazenando os dados estatísticos no banco de dados, através da tabela `plan_table`, ou qualquer outra tabela que comporte tal estrutura.

Quando uma instrução SQL é executada, o otimizador usa as estatísticas do banco de dados para gerar um plano de execução que ele usará para chegar até os dados. Quando você detectar que tal instrução está apresentando indícios de problema de performance, a primeira coisa a ser feita é verificar o plano de execução usado pelo otimizador. E, para isso, você pode utilizar o `explain plan`. Ele poderá revelar, por exemplo, que a instrução SQL não está usando índices relevantes ou até mesmo que podem

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

estar faltando índices para suportar de forma mais adequada a busca dos dados.

Quando executamos o `explain plan` para uma instrução SQL, todos os dados extraídos pelo motor do banco são salvos na Plan Table. Se este comando já foi utilizado na base de dados, provavelmente ela já deve existir. Caso contrário, tal tabela deve ser criada. O script (`utlxplan.sql`) para a criação desta tabela é fornecido pela Oracle, e no ambiente Windows pode ser encontrado, por exemplo, em `%ORACLE_HOME%\rdbms80\admin` e no UNIX, em `$ORACLE_HOME/rdbms/admin` . Se preferir, você também poderá criá-la manualmente utilizando a mesma estrutura contida no arquivo, trocando o nome. É possível definir qual tabela será utilizada para guardar os dados caso você precise, no entanto é comum mantermos o padrão.

Para criar a `plan table` através do SQL*Plus, execute o script `utlxplan.sql` , conforme pode ser visto a seguir:

```
SQL> @utlxplan.sql
```

> **Nota:** Vale lembrar que o arquivo precisa estar no diretório configurado como padrão para o SQL*Plus. Caso contrário, é necessário informar o caminho completo do arquivo. Exemplo: `@C:\utlxplan.sql` .

A sintaxe do comando `explain plan` é bem simples. Veja a sintaxe para o comando `explain plan` :

```
explain plan set statement_id = <text> [into <schema.table>] for
<statement>;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nela:

- `<text>` : será um descritivo identificador para a análise do explain. Este parâmetro pode ser qualquer texto válido com no máximo 30 caracteres. Inserir essa identificação é muito importante para a localização posterior do plano de execução, principalmente se a `plan_table` estiver sendo compartilhada com outros usuários.
- `<schema.table>` : indica a tabela onde serão armazenadas as informações geradas referentes aos planos de execução da instrução SQL (tabela `plan_table` ou sua tabela personalizada). Por padrão, o utilitário `explain plan` assume a `plan_table` .
- `<statement>` : instrução SQL que será analisada.

Para executar o `explain plan` , alguns pontos devem ser estabelecidos. Neste momento, a `plan_table` já deve estar criada, como já foi mencionado. O `explain plan` pode ser executado tanto no modo de otimização por Regra (RBO) como por Custo (CBO) do banco de dados. Para isso, pode ser usado o pacote `dbms_stats` (ou `analyze`), gerando ou excluindo as estatísticas referentes às tabelas utilizadas na instrução SQL que será analisada pelo `explain plan` .

O exemplo a seguir demonstra o `explain plan` de uma instrução SQL baseada nesses dois modos. Para executar utilizando o modo de otimização baseado em custo, vamos utilizar o `dbms_stats` (ou `analyze`) primeiro nas tabelas que fazem parte da cláusula `from` da instrução SQL, para atualizar as estatísticas. Para isso, vamos executar a chamada `gather_table_stats` do pacote `dbms_stats` :

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> exec dbms_stats.gather_table_stats('TTUN', 'EMPLOYEES');

Procedimento PL/SQL concluído com sucesso.

SQL> exec dbms_stats.gather_table_stats('TTUN', 'DEPARTMENTS');

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Feito isso, vamos executar o `explain plan` para uma instrução SQL qualquer, para que possamos visualizar seu plano de execução:

```
SQL> explain plan set statement_id = 'explain_exemplo_custo' into
 plan_table for
  2  select e.employee_id, e.first_name, d.department_name
  3  from employees e, departments d
  4  where e.department_id = d.department_id;

Explicado.
```

Agora recuperamos o resultado da table `plan_table`, referente ao `explain` do comando que acabamos de executar:

```
SQL> select lpad( ' ', 3 * level - 3 ) || operation || ' ' || opt
ions || ' ' || object_name
  2           || ' ' || decode ( id, 0, 'custo = ' || position )
"Plano de Acesso"
  3  from plan_table
  4  start with id = 0
  5  and   lower( statement_id ) = 'explain_exemplo_custo'
  6  connect by prior id = parent_id
  7  and     lower( statement_id ) = 'explain_exemplo_custo';
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plano de Acesso
--------------------------------------------------------------------------------
SELECT STATEMENT   custo = 5
    HASH JOIN
        TABLE ACCESS FULL DEPARTMENTS
        TABLE ACCESS FULL EMPLOYEES
```

Figura 12.3: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img01code.txt

Utilizando uma instrução SQL para acessar a `plain table` e informando como parâmetro o identificador no momento da execução do `explain plan`, conseguimos retornar o plano de execução para a instrução SQL cuja análise solicitamos. Estamos utilizando o `start with connect by` para nos ajudar a montar o sequenciamento dos passos do plano. Essa instrução SQL pode ser utilizada para recuperar o plano de execução de qualquer instrução. Você pode criar um arquivo de script passando como parâmetro o identificador, para sempre tê-lo em mãos quando necessitar.

Uma das características importantes do `explain plan`, e isto é muito importante saber, é que `explain plan` não executa a instrução SQL em si, ele apenas mostra qual seria um dos planos de execução que o otimizador escolheria se o comando fosse submetido a execução. Ele faz isso com base nos dados estatísticos que ele possui sobre os objetos utilizados e cursores já executados anteriormente.

Dessa forma, o método `explain plan` não requer que a consulta seja executada, o que é uma vantagem, pois acaba reduzindo muito o tempo necessário para obter um plano de execução de instruções SQL com tempo de execução longo em

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

comparação com `autotrace`, por exemplo. Através dele, você pode ter uma ideia de como está a escrita de sua instrução SQL, se da forma como você a escreveu estão sendo utilizados os melhores índices, se estão sendo feitas leituras completas (`full table scan`) em determinadas tabelas e demais informações.

Como sua execução é rápida, já que ela não executa a instrução em si, é uma ótima pedida para nós, pessoas desenvolvedoras, podermos rapidamente analisar uma instrução SQL antes de a liberarmos no programa. Na verdade, eu recomendo que, ao escrevermos uma instrução SQL, utilizemos o `explain plan` para vermos a "saúde" dela. Mesmo que estejamos trabalhando com banco de dados de desenvolvimento, é sempre importante avaliarmos se o que estamos escrevendo está adequado. Com base nisso, podemos até prever situações que podem ocorrer em ambiente de produção, dependendo dos casos.

Por outro lado, o uso do `explain plan` para a análise de algum problema de performance pode não ser uma saída tão boa assim. Isso porque pode haver distorção entre o plano de execução que ele mostra e o plano de execução que realmente o otimizador vai utilizar. Existe uma série de variáveis que o otimizador considera ao gerar vários planos para uma instrução SQL. Muitas delas serão vistas somente no momento da execução da instrução SQL propriamente dita, com base no ambiente que estiver executando e no momento em que estiver executando. O plano de execução é como se fossem etapas que o otimizador planeja realizar, mas quando ele for executar a instrução em si, ele pode perceber que algumas dessas etapas não são as melhores e, assim, mudar seu plano. Isso pode realmente acontecer, pois existem situações em que o `explain plan` é limitado. Veja a seguir

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

algumas destas limitações:

- O `explain plan` nos dá um plano de execução, mas pode não ser exatamente o plano da execução;
- Não utiliza `Bind Peeking / Adaptive Cursor Sharing (ACS)`. Recurso comentado no capítulo referente a histogramas;
- Todas as variáveis `bind` são consideradas do tipo `varchar2`. Isso pode gerar distorções quando estamos falando de índices, pois o tipo de dado usado como critério para uma coluna indexada pode influenciar o otimizador a utilizá-lo ou não;
- Depende do ambiente de execução (parâmetros, `trigger` de `logon` etc.), não leva em consideração alguns requisitos relacionados a questões de ambiente.

Eu diria que o `explain plan` seria uma ótima pedida para efeito de verificações simples, como ver se a instrução SQL está passando pelo menos por critérios básicos de performance, se está usando os índices corretos (ou não os usando), por exemplo, ou não fazendo leituras completas em tabelas, quando realmente não deveria, ou para apenas uma verificação rápida mesmo do plano, sem muito compromisso. Já para situações mais complexas ou de problemas, outros recursos seriam mais interessantes na análise, como o uso do `dbms_xplan` ou a geração de um `trace` de tal instrução ou programa.

Em minha experiência, já ocorreu de estar analisando problemas de performance onde o `explain plan` mostrava um plano de execução muito favorável à execução, mas a realidade não fechava com esse plano. Somente gerando um trace da execução

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

foi possível realmente ver as etapas reais que o otimizador estava fazendo, ou seja, o plano de execução que o otimizador estava utilizando era bem diferente do qual o `explain plan` havia gerado.

Pois bem, considerações feitas, vamos seguir conhecendo mais como o `explain plan` trabalha. Em resumo, o `explain plan` exibe os planos de execução para os seguintes tipos de instruções: `select`, `update`, `insert` e `delete`.

Com a visão do plano de execução gerada através do `explain plan`, é possível determinar as decisões do otimizador e compreender o desempenho de uma consulta.

Vimos que os dados listados referentes ao plano de execução foram extraídos da `plan table`, que o `explain plan` populou quando solicitamos que ele analisasse a instrução SQL. Vamos ver algumas características da tabela. As seguintes colunas da tabela `plan_table` foram utilizadas no `select`:

Coluna	Descrição
`statement_id`	Trata-se do valor do parâmetro `statement_id` (opcional) especificado no comando `explain plan`.
`operation`	É o nome da operação realizada na etapa corrente; contém um valor na primeira linha gerada para cada comando.
`options`	Mostra as opções para a operação (por exemplo, pode exibir a opção `remote` se uma tabela remota for referenciada na instrução SQL).
`object_name`	É o nome do objeto.
`id`	É o número deste passo no plano de execução.
`parent_id`	O ID do passo que opera sobre os resultados dessa etapa.
`position`	A ordem em que os passos com o mesmo pai são processados.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Dentre as colunas mais úteis da `plan_table`, estão:

Coluna	Descrição
`timestamp`	Contém a data e hora de quando o comando `explain plan` foi executado.
`cost`	O otimizador baseado em custo estima o custo para executar a operação, incluindo o custo de quaisquer operações que são necessárias para concluir esta operação (o valor desta coluna não tem uma determinada unidade de medida. É apenas um valor utilizado para comparar os custos dos planos de execução.).
`cardinality`	O otimizador baseado em custo estima o número de linhas.
`bytes`	O otimizador baseado em custo estima o número de `bytes`.
`other`	Texto SQL para cursores remotos e `parallel query slaves`, bem como informações de `parallel query dataflow-operation`.
`other_tag`	Descrições das funções do texto SQL da coluna `other`.
`optimizer`	Guarda o modo atual do otimizador.
`object_owner`	O nome do esquema que contém a tabela ou índice.
`object_type`	Um modificador que descreve o objeto, como `non-unique`, para um índice.

Nota: Um valor `null` para a coluna `cost` aparecerá quando a otimização for a baseada em regras (RBO).

Agora vamos ver o plano de execução para a mesma instrução analisada anteriormente, contudo, no modo baseado em regra (RBO).

Para executar o `explain plan` utilizando o modo de otimização baseado em regra, deve-se aplicar também o

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

dbms_stats (ou analyze), excluindo as estatísticas geradas para as tabelas que fazem parte da cláusula from da instrução SQL, alterando o modo de otimização da sessão ou utilizando o hint RULE. Nesse exemplo, vamos utilizar a exclusão das estatísticas, mas vamos também alterar o modo do otimizador na sessão.

Excluindo estatísticas para forçar o modo de otimização baseado em regra (RBO):

```
SQL> exec dbms_stats.delete_table_stats('TTUN', 'EMPLOYEES');

Procedimento PL/SQL concluído com sucesso.

SQL> exec dbms_stats.delete_table_stats('TTUN', 'DEPARTMENTS');

Procedimento PL/SQL concluído com sucesso.

SQL> alter session set optimizer_mode = 'RULE';

Sessão alterada.
```

Executando o explain plan para a seguinte instrução SQL (análise da consulta):

```
SQL> explain plan set statement_id = 'explain_exemplo_regra' into
 plan_table for
  2  select e.employee_id, e.first_name, d.department_name
  3  from   employees e, departments d
  4  where  e.department_id = d.department_id;

Explicado.

SQL>
```

Recuperamos então o resultado da table plan_table referente ao explain da instrução SQL que acabamos de executar:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> select lpad( ' ', 3 * level - 3 ) || operation || ' ' || opt
ions || ' ' || object_name
  2         || ' ' || decode ( id, 0, 'custo = ' || position ) "P
lano de Acesso"
  3  from plan_table
  4  start with id = 0
  5  and   lower( statement_id ) = 'explain_exemplo_regra'
  6  connect by prior id = parent_id
  7  and     lower( statement_id ) = 'explain_exemplo_regra'
  8  /

Plano de Acesso
--------------------------------------------------------------------------------
SELECT STATEMENT   custo =
    NESTED LOOPS
        NESTED LOOPS
            TABLE ACCESS FULL DEPARTMENTS
            INDEX RANGE SCAN EMP_DEPARTMENT_IX
        TABLE ACCESS BY INDEX ROWID EMPLOYEES
```

Figura 12.4: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img02code.txt

Como as estatísticas foram excluídas e estamos trabalhando em uma sessão configurada para trabalhar em modo baseado em regras, a informação referente ao custo para a execução veio em branco. Além disso, o plano de execução mudou, tendo o otimizador escolhido outras formas para buscar os dados.

Se você necessitar executar o `explain` mais de uma vez, lembre-se de renomear a informação referente ao identificador ou excluir as linhas do `explain` executado anteriormente.

12.3 V$SQL_PLAN

A visão `v$sql_plan` pode ser usada também para se obter dados referentes ao plano de execução, tornando-se uma

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

alternativa a mais, além do `explain plan` . Contudo, ela possui algumas características diferentes, por exemplo, das características do `explain plan` , sendo que uma delas é conter o plano de execução real que foi utilizado pelo otimizador, para determinada instrução SQL. Enquanto, no `explain plan` , temos um plano teórico de como a instrução SQL seria executada, na consulta à `v$sql_plan` , temos o plano real utilizado depois da execução. Contudo, vale uma ressalva, pois igualmente ao `autotrace` (que veremos mais adiante), a instrução SQL precisa ser ou ter sido executada pelo menos um vez.

Com isso, podemos dizer que a `v$sql_plan` mostra, na verdade, o plano de execução para um cursor, e não para uma instrução SQL em si. Sendo assim, podemos ter vários cursores para uma mesma instrução SQL, cada qual identificado por um identificador numérico (`child_number`), sendo que todos serão filhos da mesma instrução.

As informações obtidas através da `v$sql_plan` são muito semelhantes às obtidas através da `plan_table` (alimentada pelo `explain plan`). Portanto, tais informações podem ser usadas para o mesmo tipo de análise (com a ressalva de estarmos trabalhando com os dados reais da execução). Podemos através dessas informações:

- Determinar o plano de execução atual;
- Identificar o efeito da criação de um índice em uma determinada tabela;
- Localizar cursores que contêm um determinado caminho de acesso (por exemplo, verificação completa da tabela - `FULL TABLE SCAN` , ou verificação do intervalo de índice -

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

INDEX RANGE SCAN);

- Identificar índices que foram ou não utilizados pelo otimizador;
- Determinar se o otimizador selecionou um plano de execução específico (como junção de loops aninhados - NESTED LOOPS JOIN), conforme esperado;
- Realizar comparação entre planos de execução, do antes e depois de terem sofrido alterações ou modificações pelo desenvolvedor;
- Identificar como variáveis binds estão sendo passadas para a instrução SQL.

Além das colunas presentes na plan_table , na visão v$sql_plan , temos as colunas address e hash_value , que juntas podem ser utilizadas como link com a visão v$sqlarea , para buscar mais informações que podem ser úteis na análise do plano de execução do cursor. Essas colunas, mais a coluna child_number , são o link com outra view muito interessante, a v$sql , onde há outras informações adicionais referentes a estatísticas.

Nota: Você vai perceber que, pelo fato de a v$sql_plan conter informações atualizadas sobre os cursores referentes às instruções SQL executadas, vários outros utilitários e ferramentas podem fazer uso dela, como o pacote dbms_xplan , que veremos mais adiante.

Vamos ver agora alguns exemplos do uso da view

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`v$sql_plan` e como podemos fazer para recuperar o plano de execução do cursor ou cursores de uma determinada instrução SQL. Vemos o exemplo da instrução SQL a seguir:

Primeiramente, vamos declarar um parâmetro, via SQL*Plus, que usaremos para a passagem de valores em nossa instrução SQL:

```
SQL> var w_department_id number;
SQL> exec :w_department_id := 40;

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Definimos a variável `w_department_id` do tipo `NUMBER` e, logo após, atribuímos o valor 40 a ela. Este valor corresponde ao código do departamento. Agora vamos executar a instrução SQL propriamente dita.

```
SQL> select /* exemplo_vplan0005 */
  2         e.first_name
  3        ,d.department_name
  4        ,l.street_address
  5        ,c.country_name
  6        ,r.region_name
  7  from   employees e
  8        ,departments d
  9        ,locations   l
 10        ,countries   c
 11        ,regions     r
 12  where  e.department_id = d.department_id
 13  and    d.location_id   = l.location_id
 14  and    l.country_id    = c.country_id
 15  and    c.region_id     = r.region_id
 16  and    decode(:w_department_id,null,0,d.department_id) = nvl
(:w_department_id,0);

FIRST_NAME DEPARTMENT_NAME      STREET_ADDRESS        COUNTRY_NAM
E       REGION_NAME
---------- -------------------- -------------------- -----------
------- -------------
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Susan      Human Resources      8204 Arthur St        United King
dom     Europe

SQL>
```

Nessa instrução SQL, estamos selecionando o último nome do empregado, seu departamento e outras informações. Isso para todos os empregados cujo departamento seja igual ao valor contido no parâmetro `w_department_id`, que, no momento, seria o valor 40, definido através da variável criada. Note um detalhe importante logo no início do comando `select`, onde foi usada a seguinte string `"/* exemplo_vplan0005 */"`. Essa string vai nos ajudar a identificar a instrução SQL na `v$sql` e, através dela, vamos recuperar os valores referentes ao `child_number`, `address` e `hash_value`, para então usá-los na recuperação do plano de execução na `V$SQL_PLAN`. Veja outra instrução a seguir:

```
SQL> select /* no_v$sql */
  2         sql_id
  3        ,child_number
  4        ,hash_value
  5        ,address
  6  from   v$sql
  7  where  sql_text like '%exemplo_vplan0005%' and sql_text not
like '%no_v$sql%';

SQL_ID        CHILD_NUMBER HASH_VALUE ADDRESS
------------- ------------ ---------- ----------------
bacyyu7jts2c4            0 3818654084 00007FFE905521A8

SQL>
```

Note aqui que utilizamos a `view` `v$sql` para achar as informações referentes ao `child_number`, `address` e `hash_value` utilizando para isso a busca pela string de caracteres `"/* exemplo_vplan0005 */"`, na coluna `sql_text`. Nessa coluna, fica o texto da instrução SQL, então, o que fizemos foi

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

pesquisar nesta coluna a nossa string. Com isso, conseguimos recuperar as informações necessárias para acessar a `v$sql_plan`. Nesse exemplo, foi retornado apenas um cursor para instrução SQL. Com essas informações em mãos, vamos consultar a `v$sql_plan` para obter as informações referentes ao plano de execução do cursor referente a nossa instrução SQL executada, anteriormente.

```
SQL> select id
  2        ,lpad (' ', depth) || operation operation
  3        ,options
  4        ,object_name
  5        ,optimizer
  6        ,cost
  7        ,child_number
  8  from   v$sql_plan
  9  where  hash_value   = 3818654084
 10  and    address      = '00007FFE905521A8'
 11  and    child_number = 0
 12  start  with id = 0
 13  connect by (     prior id           = parent_id
 14               and prior hash_value   = hash_value
 15               and prior child_number = child_number
 16  )
 17  order siblings by id, position
 18  /
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Execution Plan
----------
| ID | OPERATION            | OPTIONS              | OBJECT_NAME            | OPTIMIZER     | COST | CHILD_NUMBER |
|----|----------------------|----------------------|------------------------|---------------|------|--------------|
| 0  | SELECT STATEMENT     |                      |                        | ALL_ROWS      |    5 |            0 |
| 1  |  NESTED LOOPS        |                      |                        |               |      |            0 |
| 2  |   NESTED LOOPS       |                      |                        |               |    5 |            0 |
| 3  |    NESTED LOOPS      |                      |                        |               |    4 |            0 |
| 4  |     NESTED LOOPS     |                      |                        |               |    3 |            0 |
| 5  |      NESTED LOOPS    |                      |                        |               |    3 |            0 |
| 6  |       TABLE ACCESS   | FULL                 | DEPARTMENTS            |               |    2 |            0 |
| 7  |       TABLE ACCESS   | BY INDEX ROWID       | LOCATIONS              |               |    1 |            0 |
| 8  |        INDEX         | UNIQUE SCAN          | LOC_ID_PK              |               |    0 |            0 |
| 9  |      INDEX           | UNIQUE SCAN          | COUNTRY_C_ID_PK        |               |    0 |            0 |
| 10 |     TABLE ACCESS     | BY INDEX ROWID       | REGIONS                |               |    1 |            0 |
| 11 |      INDEX           | UNIQUE SCAN          | REG_ID_PK              |               |    0 |            0 |
| 12 |    INDEX             | RANGE SCAN           | EMP_DEPARTMENT_IX      |               |    0 |            0 |
| 13 |   TABLE ACCESS       | BY INDEX ROWID       | EMPLOYEES              |               |    1 |            0 |
----------

14 linhas selecionadas.
```

Figura 12.5: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img03code.txt

Através dos parâmetros recuperados, conseguimos listar o plano de execução real que o otimizador utilizou para a execução da instrução SQL.

A `view` `v$sql` é semelhante ao `v$sqlarea`, exceto por ter mais estatísticas e sem agregação nas linhas. As colunas interessantes nesta `view`, além da `v$sqlarea`, são as seguintes: `optimizer_cost`, `cpu_time` e `elapsed_time`.

Já na `view` `v$sqlarea`, temos parte do texto de cada instrução SQL com até 1.000 caracteres, além de várias estatísticas. As colunas interessantes nesta `view` são: `sql_text`, `sorts`, `fetches`, `executions`, `loads`, `parse_calls`, `disk_reads`, `buffer_gets` e `rows_processed`. Mais adiante veremos como melhor extrair essas informações e visualizá-las de várias formas através do pacote `dbms_xplan`. De momento, vamos fazendo na mão para entendermos essas estruturas.

Será demonstrado agora um exemplo em que uma mesma

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

instrução - na verdade a mesma que executamos anteriormente - pode gerar mais de um plano de execução, sendo este referente a um novo cursor gerado, devido a uma pequena mudança que faremos no tipo da variável `bind` `w_department_id` , criada para receber o código do departamento a ser usado como parâmetro para a nossa instrução. Veja a seguir:

```
SQL> var w_department_id varchar2(10);
SQL> exec :w_department_id := '40';

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Nesse trecho de código, redeclaramos a variável `w_department_id` , que antes era um `NUMBER` , para um `varchar2` de dez posições. Logo depois, atribuímos o valor 40, no formato caractere, entre aspas simples. Agora vamos executar a mesma instrução SQL sem mudar uma única vírgula.

```
SQL> select /* exemplo_vplan0005 */
  2         e.first_name
  3        ,d.department_name
  4        ,l.street_address
  5        ,c.country_name
  6        ,r.region_name
  7  from   employees e
  8        ,departments d
  9        ,locations   l
 10        ,countries   c
 11        ,regions     r
 12  where  e.department_id = d.department_id
 13  and    d.location_id   = l.location_id
 14  and    l.country_id    = c.country_id
 15  and    c.region_id     = r.region_id
 16  and    decode(:w_department_id,null,0,d.department_id) = nvl
(:w_department_id,0);

FIRST_NAME DEPARTMENT_NAME      STREET_ADDRESS        COUNTRY_NAM
E   REGION_NAME
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
---------- -------------------- -------------------- -----------
--  ----------------
Susan    Human Resources        8204 Arthur St        United King
dom Europe

SQL>
```

O resultado foi o mesmo retornado anteriormente. Agora vamos recuperar as informações da `view v$sql` :

```
SQL> select /* no_v$sql */
  2         sql_id
  3        ,child_number
  4        ,hash_value
  5        ,address
  6  from   v$sql
  7  where  sql_text like '%exemplo_vplan0005%' and sql_text not
like '%no_v$sql%';

SQL_ID        CHILD_NUMBER HASH_VALUE ADDRESS
------------- ------------ ---------- ----------------
bacyyu7jts2c4            0 3818654084 00007FFE905521A8
bacyyu7jts2c4            1 3818654084 00007FFE905521A8

SQL>
```

O resultado nos mostra dois registros com os mesmos valores de `hash_value` e `address` , no entanto com `child_number` diferentes, ou seja, foi gerado um outro cursor devido a alteração do tipo de dado da variável `bind` , embora a instrução SQL seja a mesma. Se consultarmos a `v$sql_plan` , vamos ver que há dois planos de execução, um para cada cursor e ambos referentes à mesma instrução SQL.

```
SQL> select id
  2        ,lpad (' ', depth) || operation operation
  3        ,options
  4        ,object_name
  5        ,optimizer
  6        ,cost
  7        ,child_number
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  8  from   v$sql_plan
  9  where  hash_value   = 3818654084
 10  and    address      = '00007FFE905521A8'
 11  and    child_number = 0
 12  start  with id = 0
 13  connect by (     prior id           = parent_id
 14                and prior hash_value   = hash_value
 15                and prior child_number = child_number
 16  )
 17  order siblings by id, position
 18  /
```

```
Execution Plan

  ID OPERATION            OPTIONS              OBJECT_NAME            OPTIMIZER            COST CHILD_NUMBER
---- --------------------- -------------------- ---------------------- -------------- ---------- ------------
   0 SELECT STATEMENT                                                  ALL_ROWS                5            0
   1  NESTED LOOPS                                                                              0
   2   NESTED LOOPS                                                                             5            0
   3    NESTED LOOPS                                                                            4            0
   4     NESTED LOOPS                                                                           3            0
   5      NESTED LOOPS                                                                          3            0
   6       TABLE ACCESS   FULL                 DEPARTMENTS                                     2            0
   7       TABLE ACCESS   BY INDEX ROWID       LOCATIONS                                       1            0
   8        INDEX         UNIQUE SCAN          LOC_ID_PK                                       0            0
   9      INDEX           UNIQUE SCAN          COUNTRY_C_ID_PK                                 0            0
  10     TABLE ACCESS     BY INDEX ROWID       REGIONS                                         1            0
  11      INDEX           UNIQUE SCAN          REG_ID_PK                                       0            0
  12    INDEX             RANGE SCAN           EMP_DEPARTMENT_IX                               0            0
  13   TABLE ACCESS       BY INDEX ROWID       EMPLOYEES                                       1            0

14 linhas selecionadas.
```

Figura 12.6: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img04code.txt

Anteriormente, o plano de execução referente ao `child_number` foi igual a 0 e, na sequência, o plano de execução referente ao `child_number` é igual a 1.

```
SQL> select id
  2        ,lpad (' ', depth) || operation operation
  3        ,options
  4        ,object_name
  5        ,optimizer
  6        ,cost
  7    ,child_number
  8  from   v$sql_plan
  9  where  hash_value   = 3818654084
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
10  and    address      = '00007FFE905521A8'
11  and    child_number = 1
12  start  with id = 0
13  connect by (     prior id            = parent_id
14                 and prior hash_value    = hash_value
15                 and prior child_number = child_number
16  )
17  order siblings by id, position
18  /
```

```
Execution Plan

ID OPERATION           OPTIONS          OBJECT_NAME          OPTIMIZER              COST CHILD_NUMBER
---------------------- ---------------- -------------------- ---------------------- ---- ------------
0 SELECT STATEMENT                                           ALL_ROWS               5     1
1  NESTED LOOPS                                                                           1
2   NESTED LOOPS                                                                    5     1
3    NESTED LOOPS                                                                   4     1
4     NESTED LOOPS                                                                  3     1
5      NESTED LOOPS                                                                 3     1
6       TABLE ACCESS   FULL             DEPARTMENTS                                 2     1
7       TABLE ACCESS   BY INDEX ROWID   LOCATIONS                                   1     1
8        INDEX         UNIQUE SCAN      LOC_ID_PK                                  0     1
9      INDEX           UNIQUE SCAN      COUNTRY_C_ID_PK                            0     1
10     TABLE ACCESS     BY INDEX ROWID  REGIONS                                     1     1
11      INDEX           UNIQUE SCAN      REG_ID_PK                                  0     1
12    INDEX             RANGE SCAN      EMP_DEPARTMENT_IX                          0     1
13   TABLE ACCESS       BY INDEX ROWID  EMPLOYEES                                   1     1

14 linhas selecionadas.
```

Figura 12.7: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img05code.txt

Nesse caso, não houve diferenciação entre um plano de execução e outro, pois foi uma mudança simples que acabou não gerando uma interpretação diferente para o otimizador. Mas em alguns casos, dependendo da mudança, isso pode fazer com que o otimizador opte por outros planos de execução. O exemplo apresentado vale para demonstrar na prática como seria a geração de mais de um cursor para uma mesma instrução.

Existem outras `views` que podem ser analisadas e estudadas para um melhor aprofundamento do assunto. Nesse exemplo,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

utilizamos tais `views` para selecionar informações de uma única instrução SQL, contudo, você pode querer selecionar todos os cursores referentes a várias instruções SQL que foram executadas no sistema, ordenando por aqueles que estão gastando mais tempo de CPU ou fazendo mais leituras em disco, ou em memória, por exemplo. As aplicações do uso dessas `views` são vastas.

12.4 AUTOTRACE

Além do `explain plan`, a Oracle possui outro utilitário, chamado `autotrace`. Seu uso requer menos trabalho e é uma alternativa a mais. Seu papel é apresentar um relatório sobre a execução das instruções SQL logo depois que são executadas, de forma automática através do `SQL*Plus`. O relatório pode incluir a apresentação de estatísticas e o plano de execução. O `autotrace` está incluído desde a versão 3.3 do SQL*Plus e é um recurso disponível desde o Oracle Server versão 7.3. Trata-se de uma excelente ferramenta de diagnóstico para o monitoramento da performance. Vamos ver como este utilitário funciona.

Nota: Caso este recurso não esteja habilitado em seu ambiente de banco de dados, pode ser necessário executar o script referente à função `plustrace`. O script para criação (`plustrce.sql`) está disponível no diretório `ORACLE_HOME/sqlplus/admin`, por exemplo, em `C:\oraclexe\app\oracle\product\10.2.0\server\sqlplus\admin\plustrce.sql`, e pode ser executado através do próprio `SQL*PLUS`.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Habilitando o `autotrace` :

```
SQL> set autotrace on
```

Exemplo de instrução SQL sendo executada logo após habilitarmos o `autotrace` :

```
SQL> select e.employee_id, e.first_name, d.department_name, salar
y
  2  from   employees e, departments d
  3  where  e.department_id = d.department_id
  4  and    d.department_id = 60
  5  and    e.salary between 1000 and 4500
  6  /
```

A seguir o resultado obtido pelo `autotrace` e impresso automaticamente na tela. Aqui, para melhor visualização, mostrarei as informações em partes separadas:

No primeiro bloco, temos o resultado da instrução SQL:

```
EMPLOYEE_ID FIRST_NAME           DEPARTMENT_NAME
   SALARY
----------- -------------------- ------------------------------ -
---------
        103 Alexander            IT
     1800
        104 Bruce                IT
     1200
```

Logo após, o bloco referente ao plano de execução:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plano de Execução
----------------------------------------------------------
Plan hash value: 1323567469

-----------------------------------------------------------------------------------------------
| Id | Operation                    | Name              | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |                   |    2 |   162 |     2   (0)| 00:00:01 |
|  1 |  NESTED LOOPS                |                   |    2 |   162 |     2   (0)| 00:00:01 |
|  2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |    1 |    30 |     1   (0)| 00:00:01 |
|* 3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |    1 |       |     1   (0)| 00:00:01 |
|* 4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |    2 |   102 |     1   (0)| 00:00:01 |
|* 5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |    1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=60)
   4 - filter("E"."SALARY">=1000 AND "E"."SALARY"<=4500)
   5 - access("E"."DEPARTMENT_ID"=60)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Figura 12.8: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img06code.txt

E por último, os dados estatísticos:

```
Estatísticas
----------------------------------------------------------
          8  recursive calls
          0  db block gets
          9  consistent gets
          0  physical reads
          0  redo size
        839  bytes sent via SQL*Net to client
        523  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed

SQL>
```

O `autotrace` apresenta, além do plano de execução, as

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

estatísticas geradas para cada instrução SQL que for executada, tornando um pouco mais completo e mais apresentável que o `explain plan`. Ele mostra dados referentes a quantos `buffers` são usados, quanto tempo levou para executar, quantas leituras e gravações de disco foram realizadas, facilitando muito a visualização de quais pontos estão consumindo mais recursos.

Ele também pode mostrar essas estatísticas, pois, na verdade, ao contrário do `explain plan`, o `autotrace` executa a instrução SQL primeiramente, tornando suas estatísticas mais realistas.

Isso é importante, mas ao mesmo tempo pode ser um entrave, pois, para obter o plano de execução, a instrução SQL deve ser executada até sua conclusão. Se a instrução for particularmente ineficiente ou retornar muitas linhas, isso pode levar um tempo considerável. Em algumas situações, o plano de execução de uma instrução SQL está tão complexo que não é possível aguardar que a execução dela finalize. Nesses casos, o uso da opção `traceonly` do `autotrace` também não resolveria a situação (vamos ver as opções de uso do `autotrace` mais à frente), pois essa opção apenas suprimiria a saída dos dados da consulta, não impedindo que a instrução SQL fosse executada. Assim, as consultas com baixa performance ainda demorariam muito para serem concluídas e poderiam não ter seus dados apresentados, caso o problema de performance fosse mais complexo. Mas existe uma solução para esta questão: usar a opção `traceonly explain`, que executa somente o `explain plan`, em vez de executar a instrução. Nesses casos, teríamos praticamente a mesma funcionalidade do `explain plan`, mas, ainda assim, de forma automatizada. Veja um exemplo a seguir:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> set autot on exp
SQL> select e.employee_id, e.first_name, d.department_name, salar
y
  2  from   employees e, departments d
  3  where  e.department_id = d.department_id
  4  and    d.department_id = 60
  5  and    e.salary between 1000 and 4500
  6  /

EMPLOYEE_ID FIRST_NAME           DEPARTMENT_NAME
   SALARY
----------- -------------------- ------------------------------ -
---------
        103 Alexander            IT
     1800
        104 Bruce                IT
     1200

Plano de Execução
----------------------------------------------------------
Plan hash value: 1323567469

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     2 |   162 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                   |     2 |   162 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |     1 |    30 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |     1 |       |     1   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     2 |   102 |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=60)
   4 - filter("E"."SALARY">=1000 AND "E"."SALARY"<=4500)
   5 - access("E"."DEPARTMENT_ID"=60)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Figura 12.9: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img07code.txt

Antes de executar a instrução SQL, configuramos o `autotrace` com `set autot on exp` , forçando-o a trabalhar

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

apenas de forma a mostrar o plano de execução. É possível verificar também que, neste caso, os dados estatísticos de execução não foram gerados, pois a execução da instrução não ocorreu.

Com relação às estatísticas geradas pelo `autotrace`, vamos verificar cada elemento da estatística dela e seu respectivo significado.

Estatística	Descrição
`recursive calls`	Trata-se do número de comandos recursivos executados pelo Oracle até chegar ao resultado correspondente ao comando analisado. Essas instruções são referentes à verificação de sintaxe da instrução SQL, acessos de usuário, `extend` de tabelas entre outros. São como ações preparatórias e de verificação para que as ações de busca propriamente dita sejam executadas.
`db block gets`	Indica quantas vezes o otimizador precisou buscar informações referentes a dados do disco na memória. Quanto mais dados em memória, menos acessos ao disco e mais rápida pode ser a execução.
`consistent gets`	Indica quantas vezes o otimizador precisou buscar informações referentes a área de `rollback` na memória.
`physical reads`	É o número total de blocos lidos em disco (leitura física). Este total está relacionado à soma de `db block gets` e `consistent gets`. Este número pode ser alto, caso uma leitura `FULL TABLE` esteja sendo realizada.
`redo size`	Tamanho em `bytes`, usado pelo `buffer` de `redo`. Esta área está relacionada aos comandos DML: `insert`, `delete` e `update`.
`bytes sent via SQL*Net to client`	Número total de bytes enviados ao cliente referente aos processos executados em primeiro plano.
`bytes received via SQL*Net from client`	Número total de `bytes` recebidos do cliente via `SQL*Net`.
`SQL*Net roundtrips to/from client`	Número total de mensagens de ida e vinda através do `SQL*Net`. Esta contagem vai desde o momento do envio da instrução SQL até o recebimento das informações.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`sorts (memory)`	Quantidade de operações feitas em memória (leitura ou escrita).
`sorts (disk)`	Quantidade de operações realizadas em disco (leitura ou escrita).
`rows processed`	Número de linhas processadas, ou seja, número de linhas do resultado da instrução SQL.

Quando o `autotrace` é habilitado na sessão de SQL*Plus, ele é iniciado com algumas opções padrões de operação. Contudo, outras formas de funcionamento podem ser configuradas para que ele trabalhe de diferentes modos. A seguir, algumas destas opções que podem ser escolhidas para a execução do `autotrace`.

Opção	Função
`set autot on`	Modo abreviado de ativar `autotrace`
`set autot on exp`	Mostra apenas o `explain plan`
`set autotrace on stat`	Mostra apenas as estatísticas
`set autot trace`	Não mostra a saída da consulta

Diferenças entre `autotrace` e `explain plan`:

- O `autotrace` é mais fácil de utilizar pois é puramente declarativo.
- No `autotrace`, as estatísticas e o plano de execução são mais realistas, pois a instrução SQL é executada.
- O `autotrace` em sua forma padrão pode não ser uma boa pedida para consultas muito demoradas ou impossíveis de execução.
- Igualmente ao `explain plan`, o `autotrace` pode ser usado para avaliar qualquer instrução DML (`Data Manipulation Language`), como `select`, `delete`,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`update` ou `insert`.

- Os dois utilitários são muito úteis para monitorar e ajustar o desempenho de instruções DML.

Vimos que o `autotrace` pode ser bem produtivo no sentido de estarmos sempre atentos a como estamos escrevendo nossas instruções SQL e para analisar problemas de desempenho. Podemos deixá-lo ativo em nossa sessão do SQL*Plus para que, a cada instrução, ele possa nos passar informações sobre a saúde delas e nos alertar sobre algum possível problema.

Agora que vimos um pouco sobre os dois utilitários muito úteis no trabalho de extração e visualização de plano de execução, vamos falar de uma dupla, também muito utilizada: o `trace` e `tkprof`. Juntos, eles elevam mais o nível de informação e entendimento do que acontece quando uma instrução SQL é executada, mostrando mais dados e operações que podem ser analisadas para melhor entender todos os passos que o otimizador realiza para uma determinada execução. Seria como um raio-x bem mais completo da situação. Vamos lá.

12.5 DBMS_XPLAN

O pacote `dbms_xplan` possibilita a visualização de planos de execução em diferentes layouts, digamos assim, mostrando as informações de diferentes formas. Ele também pode ser utilizado para demonstrar as informações geradas pelo `explain plan`, através dos dados gerados na `plan_table`, bem como planos de execução reais que foram gerados através dos cursores executados para as instruções SQL. Além disso, ele pode mostrar os planos de execução com base em um histórico de execução, recuperando, por

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

exemplo, dados do AWR. Em tópicos anteriores, foi visto esse recurso em demonstrações de planos de execução e informações estatísticas.

> **NOTA:** O repositório do AWR (`Automatic Workload Repository`) é gerado automaticamente durante a criação do banco de dados. Nesse repositório, ficam todas as estatísticas recuperadas automaticamente pelo banco de dados. São informações históricas referentes a execuções de instruções SQL.

Esta ferramenta é muito produtiva para a análise das informações de performance das instruções SQL. Você poderá utilizá-la a partir do Oracle 9i, sendo que na versão 10G suas funções foram aprimoradas. Basicamente, através dela, você gera saídas mais básicas sobre as informações estatísticas e dos planos de execução, até visões mais complexas. Os níveis de saída são cinco, sendo eles:

- `BASIC` : exibe as informações mínimas no plano de execução, como o ID da operação, o nome da operação e sua opção.
- `TYPICAL` : este é o padrão. Quando não informada a opção de saída (passado `null` como parâmetro) é este o nível de saída que será apresentado. Exibe as informações mais relevantes no plano de execução, sendo elas o ID da operação, nome da operação, opção, `#linhas` , `#bytes` e `custo` do otimizador.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- `SERIAL` : idêntico ao `TYPICAL` , exceto que as informações paralelas não são exibidas, mesmo que o plano seja executado em paralelo.
- `ALL` : nível máximo de usuário. Inclui informações exibidas no nível `TYPICAL` , com informações adicionais (`PROJECTION` , `ALIAS` e informações sobre o `REMOTE SQL` se a operação for distribuída).
- `ADVANCED` : idêntico ao `TYPICAL` , mas também inclui as informações que ele chama de `Outline` (ou seja, o conjunto de dicas que reproduzirão o plano de execução), e as variáveis `bind` usadas para otimizar a instrução SQL.

Existem outros dois níveis de saída adicionais que também podem ser usados, o `RUNSTATS_LAST` e `RUNSTATS_TOT` . Contudo, pode haver restrições quanto a seus usos, podendo ser usados somente se os cursores em questão tiverem sido compilados e executados com o parâmetro de inicialização `STATISTICS_LEVEL` configurado como `ALL` . O nível `RUNSTATS_LAST` exibe as estatísticas de tempo de execução para a última execução de um determinado cursor. Já o `RUNSTATS_TOT` exibe o total agregado de estatísticas de tempo de execução para todas as execuções de uma instrução SQL específica, desde o momento em que a instrução foi analisada e executada pela primeira vez.

Devido à sua flexibilidade e contando com uma gama enorme de informações estatísticas disponíveis vindas do banco de dados, é possível gerar saídas com informações mais precisas, acrescentando palavras-chaves que podem ser adicionadas aos níveis de saída apresentados, personalizando seu comportamento padrão de saída. Cada palavra-chave representa um grupo lógico

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

de colunas da tabela referente ao plano de execução (como `PARTITION`) ou adições lógicas à saída da tabela do plano base (como `PREDICATE`). Mais de uma palavra-chave pode ser usada, sendo que elas devem ser separadas por vírgula ou espaço. Logo, para que o grupo de informações não apareça, devemos inserir seu nome precedido pelo sinal de subtração, como "-PREDICATE". Veja a seguir um quadro mostrando estas palavras-chaves e sua aplicação:

Palavra-chave	Descrição
`ROWS`	Mostra o número de linhas estimadas pelo otimizador.
`BYTES`	Mostra o número de `bytes` estimado pelo otimizador.
`COST`	Mostra informações de custo do otimizador.
`PARTITION`	Mostra informações de remoção de partição.
`PARALLEL`	Mostra informações de PX (método de distribuição e informações da fila de tabelas).
`PREDICATE`	Mostra a seção de predicado.
`PROJECTION`	Mostra a seção de projeção.
`ALIAS`	Mostra a seção `"Nome do bloco de consulta / alias de objeto"` .
`REMOTE`	Mostra as informações de consultas distribuídas.
`NOTE`	Mostra a seção de notas do plano de execução.
`IOSTATS`	Supondo que as estatísticas básicas do plano sejam coletadas quando as instruções SQL são executadas (usando a dica `GATHER_PLAN_STATISTICS` ou definindo o parâmetro `STATISTICS_LEVEL` como `ALL`), esse formato mostra estatísticas de E/S para o nível `ALL` (ou apenas para o nível `LAST`) referente aos planos de execução referente aos cursores.
	Supondo que o gerenciamento de memória `PGA` esteja ativado (ou seja, o parâmetro `PGA_AGGREGATE_TARGET` esteja definido como um valor diferente de 0), esse formato permite exibir estatísticas de gerenciamento de memória (por exemplo, modo de execução do

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

MEMSTATS	operador, quantidade de memória usada, número de bytes descarregados para o disco e assim por diante). Essas estatísticas se aplicam apenas a operações intensivas de memória, como junções de hash (HASH JOINS), classificação (SORT) ou alguns operadores de bitmap .
ALLSTATS	Atalho para IOSTATS MEMSTATS
LAST	Por padrão, as estatísticas do plano de execução são mostradas para todas as execuções do cursor. A palavra-chave LAST pode ser especificada para ver apenas as estatísticas da última execução.

Desta forma, podemos combinar diversos formatos para mostrar ou inibir as informações que desejamos ou não. Vale lembrar que nem todos os formatos estão disponíveis em todos os níveis de saída ou nas procedures (que veremos mais adiante). Veja alguns exemplos de combinação de formatos e seus resultados:

- Use ALL -PROJECTION -NOTE para exibir tudo, exceto as seções de projeção e nota.
- Use TYPICAL PROJECTION para exibir o formato típico com a seção de projeção adicional (que normalmente é excluída no formato TYPICAL). Como TYPICAL é o padrão, usar simplesmente PROJECTION é equivalente.
- Use -BYTES -COST -PREDICATE para exibir usando o formato TYPICAL , mas excluindo estimativas de custo e de byte do otimizador, bem como a seção de predicados.
- Use BASIC ROWS para exibir informações básicas, com o número adicional de linhas estimadas pelo otimizador.

Você verá também que algumas dessas palavras-chaves podem aparecer sozinhas, sem a menção de um dos cinco níveis de saída, como MEMSTATS , ALLSTATS e IOSTATS . Mais à frente veremos alguns exemplos na prática do uso dessas palavras-chaves.

Além dos níveis de saída (ou de apresentação) e de seus

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

formatos, temos as procedures propriamente ditas que fazem parte do pacote `dbms_xplan` e que vão fornecer os dados referentes aos planos de execução e dados estatísticos. Suas chamadas são bem simples e, na maioria das vezes, só será necessário informar os seus parâmetros, como qual o ID da instrução SQL e o formato de saída escolhido. A seguir as procedures que compõem o pacote:

- `display` : para formatar e exibir o conteúdo de uma `plan_table` , ou seja, de dados gerados através do `explain plan` (Plano Teórico).
- `display_cursor` : para formatar e exibir o conteúdo de um plano de execução de qualquer cursor gerado (Plano Real).
- `display_awr` : para formatar e exibir o conteúdo de um plano de execução de uma instrução SQL armazenada no `AWR` .
- `display_sql_plan_baseline` : para exibir um ou mais planos de execução para a instrução SQL identificada pelo identificador `"SQL handle"` .
- `display_sqlset` : para formatar e exibir o conteúdo de um plano de execução de instruções armazenadas em um `"SQL tuning set"` .

O foco aqui será na procedure utilizada na grande maioria das vezes, a `display_cursor` .

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Para poder trabalhar com a procedure `display_cursor`, você precisará ter privilégios para as seguintes views: `v$sql_plan`, `v$session` e `v$sql_plan_statistics_all`. Para cada procedure mencionada são necessários outros grants adicionais. Todos esses privilégios podem ser adquiridos através da role `select_catalog`.

A execução da procedure `display_cursor` sem informar parâmetro algum já mostra as informações do último cursor executado. Veja o exemplo a seguir:

```
SQL> select * from table(dbms_xplan.display_cursor);

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------
---------------------------
SQL_ID  97hdv5nu08dg2, child number 0
-------------------------------------
select e.employee_id       ,e.first_name   ,d.department_name
,d.department_id from   employees e       ,departments d where
e.department_id = d.department_id and    d.department_id = 80
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
SQL_ID  97hdv5nu08dg2, child number 0
-------------------------------------
select e.employee_id       ,e.first_name   ,d.department_name
,d.department_id from   employees e       ,departments d where
e.department_id = d.department_id and    d.department_id = 80

Plan hash value: 1492013603

--------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |       |       |     3 (100)|          |
|   1 |  NESTED LOOPS                |             |    34 |  1020 |     3   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    16 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK  |     1 |       |     0   (0)|          |
|*  4 |   TABLE ACCESS FULL          | EMPLOYEES   |    34 |   476 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=80)
   4 - filter("E"."DEPARTMENT_ID"=80)


24 linhas selecionadas.
```

Figura 12.10: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img08code.txt

Como não foi informado o nível de saída, por padrão é o `typical` .

Mas nem sempre vamos desejar buscar o último cursor. Podemos querer localizar uma determinada instrução SQL ou até mesmo uma lista de instruções SQL. Para isso, precisaremos informar algumas informações via parâmetro. Vale lembrar que, embora estejamos focando nesta procedure, o funcionamento da procedure `display` , que mostra informações da `plan_table` , é muito semelhante, conforme pode ser visto no exemplo a seguir com a mesma instrução SQL:

```
SQL> explain plan for
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
  2  select e.employee_id
  3        ,e.first_name
  4    ,d.department_name
  5    ,d.department_id
  6  from   employees e
  7        ,departments d
  8  where  e.department_id = d.department_id
  9  and    d.department_id = 80;

Explicado.

SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
Plan hash value: 1492013603

--------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    34 |  1020 |     3   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |             |    34 |  1020 |     3   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    16 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS FULL          | EMPLOYEES   |    34 |   476 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=80)
   4 - filter("E"."DEPARTMENT_ID"=80)

17 linhas selecionadas.
```

Figura 12.11: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img09code.txt

Agora vamos imaginar que gostaríamos de buscar as informações de uma instrução SQL específica e também utilizar o nível `all` para mostrar mais informações, além daquelas mostradas no nível `typical` .

Anteriormente, quando foi visto sobre a `view` `v$sql` , vimos como recuperar algumas informações de identificação de uma

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

instrução SQL, filtrando pelo campo `sql_text`. Na ocasião, estávamos recuperando o `hash_value`, `address` e `child_number`. Pois bem, vamos utilizar o mesmo método, mas agora para buscar o `sql_id`. Mas antes, vamos ver os possíveis parâmetros para a procedure `display_cursor`:

Sintaxe:

```
dbms_xplan.display_cursor(
   sql_id           in  varchar2  default  null,
   cursor_child_no  in  number    default  0,
   format           in  varchar2  default  'TYPICAL');
```

Nela:

- `sql_id`: especifica o `sql_id` da instrução SQL no cache do cursor. Você pode recuperar o valor apropriado consultando a coluna `sql_id` em `v$sql` ou `v$sqlarea`. Como alternativa, você pode escolher a coluna `prev_sql_id` para uma sessão específica na `v$session`. O parâmetro padrão é `null`; nesse caso, o plano do último cursor executado pela sessão é exibido.
- `cursor_child_no`: número filho do cursor a ser exibido. Se não for fornecido, o plano de execução de todos os cursores correspondentes ao parâmetro `sql_id` fornecido é exibido. O `child_number` pode ser especificado apenas se `sql_id` for especificado. O `child_number` também pode ser recuperado pela view `v$sql`.
- `format`: controla o nível de detalhes do plano de execução (apresentação das informações). Ver tabela de formatos (palavras-chaves).

Veja os exemplos:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Não informando parâmetros:

```
SELECT * FROM table (DBMS_XPLAN.DISPLAY_CURSOR);
```

Informando o SQL_ID:

```
SELECT * FROM table (DBMS_XPLAN.DISPLAY_CURSOR('atulksyanrpo'));
```

Informando todos os parâmetros: `SQL_ID`, `CURSOR_CHILD_NO` e `FORMAT`:

```
SELECT * FROM table (DBMS_XPLAN.DISPLAY_CURSOR('atfwcg8anrykp', N
ULL, 'ALLSTATS LAST'));
```

Agora vamos voltar ao cenário anterior, no qual desejávamos retornar as informações de uma instrução SQL específica e trazer a maior quantidade de informações:

```
SQL> select /* LOCATION_1400 */
  2         e.first_name
  3        ,d.department_name
  4  from   employees e
  5        ,departments d
  6  where  e.department_id = d.department_id
  7  and    d.location_id = 1400;

FIRST_NAME           DEPARTMENT_NAME
-------------------- ------------------------------
Alexander            IT
Bruce                IT
David                IT
Valli                IT
Diana                IT

SQL>
```

Primeiramente, foi executada uma instrução SQL para servir de base para a nossa busca. Foi colocado um comentário logo no início do `select`, servindo como identificador, da seguinte forma: `"LOCATION_1400"`. Agora, através das `view` `V$SQL`,

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

vamos localizar a instrução SQL executada, utilizando o identificador que criamos:

```
SQL> select /* no_v$sql */
  2         sql_id
  3        ,child_number
  4  from   v$sql
  5  where  sql_text like '%LOCATION_1400%'
  6  and    sql_text not like '%no_v$sql%';

SQL_ID        CHILD_NUMBER
------------- ------------
012317fyg0bch            0

SQL>
```

Com o `sql_id`, é feita a chamada à procedure `display_cursor` passando-o como parâmetro. Vamos aproveitar e passar como parâmetro o formato `all`.

```
SQL> select * from table (dbms_xplan.display_cursor('012317fyg0bc
h', NULL, 'ALL'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------
-----------------------------------------------------
SQL_ID  012317fyg0bch, child number 0
-------------------------------------
select /* LOCATION_1400 */        e.first_name   ,d.department_na
me
from   employees e       ,departments d where  e.department_id =
d.department_id and    d.location_id = 1400
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plan hash value: 1021246405

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |       |       |     4 (100)|          |
|   1 |  NESTED LOOPS                |                   |       |       |            |          |
|   2 |   NESTED LOOPS               |                   |    37 |  1073 |     4   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | DEPARTMENTS       |     4 |    76 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |    10 |       |     0   (0)|          |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |    10 |   100 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 12.12: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img10code.txt

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   3 - SEL$1 / D@SEL$1
   4 - SEL$1 / E@SEL$1
   5 - SEL$1 / E@SEL$1

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("D"."LOCATION_ID"=1400)
   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - "D"."DEPARTMENT_NAME"[VARCHAR2,30], "E"."FIRST_NAME"[VARCH
AR2,20]
   2 - "D"."DEPARTMENT_NAME"[VARCHAR2,30], "E".ROWID[ROWID,10]
   3 - "D"."DEPARTMENT_ID"[NUMBER,22], "D"."DEPARTMENT_NAME"[VARC
HAR2,30]
   4 - "E".ROWID[ROWID,10]
   5 - "E"."FIRST_NAME"[VARCHAR2,20]
```

Note que algumas informações são mostradas aqui com base no nível de detalhes que configuramos na chamada a procedure

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

`display_cursor`. Vamos dar uma olhada.

```
PLAN_TABLE_OUTPUT
---------------------------------------------------------------------
-------------------------------------------------------
SQL_ID  012317fyg0bch, child number 0
-------------------------------------
select /* LOCATION_1400 */        e.first_name   ,d.department_na
me
from   employees e       ,departments d where  e.department_id =
d.department_id and    d.location_id = 1400

Plan hash value: 1021246405

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |       |       |     4 (100)|          |
|   1 |  NESTED LOOPS                |                   |       |       |            |          |
|   2 |   NESTED LOOPS               |                   |    37 |  1073 |     4   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | DEPARTMENTS       |     4 |    76 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |    10 |       |     0   (0)|          |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |    10 |   100 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Figura 12.13: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img11code.txt

Nesse primeiro bloco é mostrada a instrução SQL, acompanhada do seu `Id` e `Child Number`. É mostrado também o valor `hash` do plano. Logo após essas informações, é mostrado o plano de execução. Referente ao plano, ele apresenta as informações referentes a cada operação (`Id / Operation`), com o número de linhas retornadas em cada operação executada pelo otimizador (`Rows`), bem como o número de `bytes` (`Bytes`), o custo (`Cost (%CPU)`) e o tempo de execução (`Time`). Através do plano, podemos ver que foi uma execução bem-sucedida, na qual o custo foi baixo (Custo = 4) e, se observarmos, os acessos às tabelas foram realizados utilizando índice e acesso `FULL`, que, para este cenário, foi bem plausível pelo número de registros em cada uma

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

das tabelas. Vamos para o próximo bloco.

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   3 - SEL$1 / D@SEL$1
   4 - SEL$1 / E@SEL$1
   5 - SEL$1 / E@SEL$1
```

Nessa seção, são mostrados os blocos de informações (conjunto de linhas: chamado de `Row Sources` ou `Result Set`) utilizados em cada operação do plano de execução. Como primeira informação, temos o `ID` da operação precedido da expressão `SEL$1` , que seria o bloco, com o resultado das linhas retornadas decorrente da execução de cada passo do plano, mais o bloco referente a cada tabela envolvida (por exemplo, na linha 3, temos o bloco `D@SEL$1`). O caractere `"D"` ou `"E"` , que aparecem na frente de cada bloco, referem-se os aliases de cada tabela utilizados na instrução SQL. Agora falaremos sobre a seção de predicados.

```
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("D"."LOCATION_ID"=1400)
   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Nessa seção, são mostrados os dois tipos de predicado, predicado de filtro e de acesso. O predicado de filtro (`filter`) mostra os predicados que ajudam a filtrar os dados de forma direta, através de alguma expressão, valor fixo ou de `bind` . Já os predicados de acesso são aqueles que podem limitar ou não o número de linhas, mas que são os usados para acesso a outras tabelas da instrução SQL. Temos como exemplo a igualdade da coluna `location_id` com o valor 1400, como sendo um predicado de filtro, e o `join` entre as colunas `department_id`

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

das tabelas `employees` e `departments`, representadas ali por `"E"` e `"D"`, respectivamente, como predicado de acesso. A próxima e última seção mostra as projeções:

```
Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - "D"."DEPARTMENT_NAME"[VARCHAR2,30], "E"."FIRST_NAME"[VARCH
AR2,20]
   2 - "D"."DEPARTMENT_NAME"[VARCHAR2,30], "E".ROWID[ROWID,10]
   3 - "D"."DEPARTMENT_ID"[NUMBER,22], "D"."DEPARTMENT_NAME"[VARC
HAR2,30]
   4 - "E".ROWID[ROWID,10]
   5 - "E"."FIRST_NAME"[VARCHAR2,20]
```

Nessa seção, são mostradas as informações de projeção de cada coluna, em cada passo do plano. O número que antecede cada projeção é o ID da operação no plano de execução. Logo após é mostrada cada coluna que é recuperada ou está envolvida na operação. A seguir, é mostrado o plano de execução novamente, para vermos com mais clareza o que acontece em cada passo.

```
Execution Plan
----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |       |       |     4 (100)|          |
|   1 |  NESTED LOOPS                |                   |       |       |            |          |
|   2 |   NESTED LOOPS               |                   |    37 |  1073 |     4   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | DEPARTMENTS       |     4 |    76 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |    10 |       |     0   (0)|          |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |    10 |   100 |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------
```

Figura 12.14: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img12code.txt

Vemos que nesse plano a primeira operação a ser executada é a 3. Vale ressaltar que a coluna ID não retorna a ordem de execução e sim o ID da operação. Note que é realizado um acesso completo

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

na tabela. Olhando a lista de predicados, podemos ver que, para a operação 3, temos o predicado de filtro, com o qual é feito o filtro selecionando apenas os departamentos onde a localização seja igual a 1400. Olhando para a informação de projeção referente à operação 3, vemos que as colunas recuperadas, ou seja, projetadas, nesta operação foram as `department_id` e `department_name`. A primeira será utilizada na execução da operação 4, onde acontecerá a junção com a tabela de empregados. Já a segunda coluna, `department_name`, será mostrada como resultado da instrução. Desta forma, vai se desenhando cada projeção ao longo da execução de cada operação do plano. Sabendo a que cada seção corresponde, é possível entender as saídas e os níveis de saída da procedure `display_cursor`. Vamos ver agora alguns exemplos envolvendo outras combinações de saída.

```
SQL> select * from table (dbms_xplan.display_cursor(null, null, '
BASIC'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------
---------------
EXPLAINED SQL STATEMENT:
------------------------
select e.employee_id       ,e.last_name   ,d.department_name from
employees e       ,departments d where  e.department_id =
d.department_id and    d.department_id = 30
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plan hash value: 1323567469

-----------------------------------------------------------
| Id  | Operation                    | Name              |
-----------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |
|   1 |  NESTED LOOPS                |                   |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |
|   3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |
|   5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |
-----------------------------------------------------------
```

Figura 12.15: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img13code.txt

Neste próximo exemplo, temos a saída do tipo `BASIC` apenas mostrando dados básicos do plano de execução.

```
SQL> select * from table (dbms_xplan.display_cursor(null, null, '
BASIC'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------
---------------
EXPLAINED SQL STATEMENT:
------------------------
select e.employee_id       ,e.last_name   ,d.department_name from
employees e       ,departments d where  e.department_id =
d.department_id and    d.department_id = 30
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plan hash value: 1323567469

-----------------------------------------------------------
| Id  | Operation                    | Name              |
-----------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |
|   1 |  NESTED LOOPS                |                   |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |
|   3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |
|   5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |
-----------------------------------------------------------
```

Figura 12.16: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img14code.txt

A seguir, uma junção do nível de detalhes básico mais a palavra-chave `COST`, ou seja, ele mostra os dados básicos, mais a informação referente ao custo gasto pelo otimizador.

```
SQL> select * from table (dbms_xplan.display_cursor(null, null, '
BASIC +COST'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------
---------------
EXPLAINED SQL STATEMENT:
------------------------
select e.employee_id       ,e.last_name   ,d.department_name from
employees e       ,departments d where  e.department_id =
d.department_id and    d.department_id = 30
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plan hash value: 1323567469

-----------------------------------------------------------------------
| Id  | Operation                    | Name              | Cost (%CPU)|
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     2 (100)|
|   1 |  NESTED LOOPS                |                   |     2   (0)|
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |     1   (0)|
|   3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |     0   (0)|
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     1   (0)|
|   5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     0   (0)|
-----------------------------------------------------------------------
```

Figura 12.17: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img15code.txt

Agora um exemplo da combinação do nível de apresentação `TYPICAL`, mas suprimindo as informações referentes à quantidade de `Bytes` e `Rows`. Neste exemplo, tais informações não aparecem na apresentação.

```
SQL> select * from table (dbms_xplan.display_cursor(null, null, '
TYPICAL -BYTES -ROWS'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------
SQL_ID  6mdzpx8mbpaqb, child number 0
-------------------------------------
select e.employee_id       ,e.last_name   ,d.department_name from
employees e       ,departments d where  e.department_id =
d.department_id and    d.department_id = 30
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Plan hash value: 1323567469

-----------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     2 (100)|          |
|   1 |  NESTED LOOPS                |                   |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |     0   (0)|          |
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     0   (0)|          |
-----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=30)
   5 - access("E"."DEPARTMENT_ID"=30)
```

Figura 12.18: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img16code.txt

No próximo exemplo, temos uma apresentação que mostra outras duas novas seções: `Peeked Binds` e `Note`. Cada uma delas tem um objetivo diferente. A seção `Peeked Binds`, indicada pela palavra-chave `+PEEKED_BINDS`, mostra informações sobre as variáveis `binds` contidas na instrução SQL. Já a seção `Note`, mostra informações sobre o plano de execução.

```
SQL> var w_department_id number;
SQL> var w_job_id varchar2(10);
SQL>
SQL> exec :w_department_id := 60;

Procedimento PL/SQL concluído com sucesso.

SQL> exec :w_job_id := 'IT_PROG';

Procedimento PL/SQL concluído com sucesso.

SQL>

SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALL
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
STATS LAST +PEEKED_BINDS -PROJECTION -ALIAS -PREDICATE +COST +BYT
ES'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------
SQL_ID  d6kvj71dxvg16, child number 0
-------------------------------------
select e.employee_id       ,e.last_name   ,d.department_name
,e.salary from   employees e       ,departments d where
e.department_id = d.department_id and    d.department_id =
:w_department_id and    e.job_id = :w_job_id

Plan hash value: 1323567469

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | E-Rows |E-Bytes| Cost (%CPU)|
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |        |       |     2 (100)|
|   1 |  NESTED LOOPS                |                   |      1 |    44 |     2   (0)|
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |      1 |    16 |     1   (0)|
|   3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |      1 |       |     0   (0)|
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |      1 |    28 |     1   (0)|
|   5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |      5 |       |     0   (0)|
-----------------------------------------------------------------------------------------

Peeked Binds (identified by position):
--------------------------------------

   1 - (NUMBER): 60
   2 - (VARCHAR2(30), CSID=873): 'IT_PROG'

Note
-----
   - Warning: basic plan statistics not available. These are only collected when:
       * hint 'gather_plan_statistics' is used for the statement or
       * parameter 'statistics_level' is set to 'ALL', at session or system level
```

Figura 12.19: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img17code.txt

Podemos verificar que na seção `Peeked Binds`, para as operações 1 e 2, há a existência do uso de uma variável `bind` do tipo numérico com valor igual a 60, referente ao ID de operação 1, que podemos observar ser o valor do parâmetro `w_department_id`, definido anteriormente um pouco antes da

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

execução da instrução SQL `SQL> var w_department_id number` . O mesmo vale para a informação da variável `bind` referente ao ID de operação 2, onde temos a indicação do uso de uma variável `varchar2` contendo como valor `IT_PROG` , referente ao cargo, também definida anteriormente à execução da instrução, conforme pode ser visto no exemplo.

Já na seção `Note` , vemos informações do plano de execução. Neste exemplo, há um alerta informando que as estatísticas básicas do plano de execução não estão disponíveis (`basic plan statistics not available`). Logo após, ele também informa em quais casos ela é coletada. Note também que aqui estamos utilizando a saída `ALLSTATS LAST` , na qual são mostradas informações de E/S e memória referentes à última execução do cursor.

Outra seção que podemos visualizar na saída de `display_cursor` é a `outline_data` . Ela é visualizada quando utilizamos o nível de saída `ADVANCED` . Conforme mencionado anteriormente, ela apresenta os mesmos dados da `TYPICAL` , mas com o acréscimo de dados referentes a como o otimizador estruturou as informações para o plano e também informações sobre variáveis `binds` criadas e utilizadas por ele para otimizar a instrução SQL. Veja um exemplo a seguir deste tipo de saída:

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ADV
ANCED'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------
SQL_ID  d6kvj71dxvg16, child number 0
-------------------------------------
select e.employee_id       ,e.last_name   ,d.department_name
,e.salary from   employees e       ,departments d where
e.department_id = d.department_id and    d.department_id =
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
:w_department_id and    e.job_id = :w_job_id

Plan hash value: 1323567469

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |       |       |     2 (100)|          |
|   1 |  NESTED LOOPS                |                   |     1 |    44 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |     1 |    16 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |     1 |       |     0   (0)|          |
|*  4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     1 |    28 |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     5 |       |     0   (0)|          |
-----------------------------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   2 - SEL$1 / D@SEL$1
   3 - SEL$1 / D@SEL$1
   4 - SEL$1 / E@SEL$1
   5 - SEL$1 / E@SEL$1
```

Figura 12.20: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img18code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.2')
      DB_VERSION('11.2.0.2')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      INDEX_RS_ASC(@"SEL$1" "D"@"SEL$1" ("DEPARTMENTS"."DEPARTMENT_ID"))
      INDEX_RS_ASC(@"SEL$1" "E"@"SEL$1" ("EMPLOYEES"."DEPARTMENT_ID"))
      LEADING(@"SEL$1" "D"@"SEL$1" "E"@"SEL$1")
      USE_NL(@"SEL$1" "E"@"SEL$1")
      END_OUTLINE_DATA
  */

Peeked Binds (identified by position):
--------------------------------------
...

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=:W_DEPARTMENT_ID)
   4 - filter("E"."JOB_ID"=:W_JOB_ID)
   5 - access("E"."DEPARTMENT_ID"=:W_DEPARTMENT_ID)

Column Projection Information (identified by operation id):
-----------------------------------------------------------
```

Figura 12.21: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img19code.txt

Para tornar a saída mais limpa, suprimi de parte do código algumas seções, mantendo apenas as seções `PLAN_TABLE_OUTPUT`, `Query Block Name`, `Predicate Information` e `Outline Data`, esta última se tratando do nosso exemplo. Veja que ela apresenta uma série de informações, desde os dados das versões do otimizador e banco de dados, o tipo de otimização, até os métodos de junção utilizados para os `joins` entre as tabelas e `RESULT SETs`, e `hints` para melhor otimizar o plano. Através dessas informações, você pode analisar com mais detalhes como o plano

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

foi gerado e entender melhor como sua execução ocorre.

Vimos que o uso da procedure `display_cursor` torna mais eficaz a análise da performance de instruções SQL, comparado com a procedure `display` do mesmo pacote, ou com o próprio uso do `explain plan` , pelo fato de trabalhar com dados reais de execução. No entanto, mesmo utilizando o `display_cursor` , embora o otimizador trabalhe com estatísticas e informações em cima de cursores executados anteriormente, para uma mesma instrução SQL, os dados do plano de execução e das estatísticas mostrados e utilizados pelo otimizador podem não refletir a atual situação em alguns casos.

Para melhorar ainda mais a precisão do plano de execução, é possível fazer com que outras estatísticas sejam colhidas em tempo de execução e mescladas às do plano de execução atual, fazendo com que o otimizador tenha dados mais precisos e tudo isso apenas para determinada instrução SQL, sem afetar toda a instância do banco de dados. Isso pode ser feito através do uso do `hint GATHER PLAN STATISTICS` . Veja a seguir mais informações sobre esta dica.

O hint GATHER PLAN STATISTICS

Se você deseja reunir estatísticas do plano de execução, estendendo sua visualização e trazendo informações reais de memória e cardinalidade, por exemplo, para uma instrução SQL, e não deseja comprometer o desempenho de uma instância inteira do banco de dados ao coletar essas informações, você pode utilizar o `hint GATHER PLAN STATISTICS` .

Vimos em capítulos anteriores como o uso de `hints` pode

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

influenciar o otimizador a tomar algumas decisões. O `hint GATHER PLAN STATISTICS` reúne estatísticas em tempo de execução, mas para isso a instrução SQL precisa ser executada. Se você possui algum cenário onde a performance esteja baixa e você acredita que as estatísticas talvez não estejam refletindo exatamente a realidade, pode ser interessante executar a instrução SQL utilizando o `hint GATHER PLAN STATISTICS`. O uso dele pode fornecer, de forma rápida, informações que você não possui em uma execução normal, pois, com o uso desse `hint`, informações reais e estimadas sobre as estatísticas da instrução SQL são mostradas. Assim, será possível analisar tais informações com mais precisão.

Aconselho pesquisar mais a fundo o funcionamento desse recurso na documentação da Oracle, e também consultar o DBA do banco de dados, ou responsável pela manutenção dele, apenas para deixá-lo alinhado sobre seu uso.

Vale ressaltar que, para estes casos, a instrução SQL precisa ser executada até o final para que o plano de execução possa ser mostrado com base nas estatísticas adicionais. Outra dica importante pode ser comparar as execuções, com e sem o uso do `hint`, para ver as diferenças, e talvez indicar a atualização das estatísticas do sistema como um todo, caso elas não estejam sendo realizadas periodicamente, por exemplo.

O uso deste `hint` só afetará a instrução SQL, reunindo estatísticas extras apenas para esta instrução em específico. Desta forma, seu escopo é limitado à instrução e não afetará outros processos no banco de dados ou mesmo a própria sessão onde a instrução será executada. Logo, para que o otimizador utilize estas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

estatísticas em qualquer outra situação, por outros programas, por exemplo, é recomendada a atualização das estatísticas para os objetos em questão. Em capítulos anteriores, já vimos como atualizar as estatísticas de tabelas específicas. Segue como ficaria uma saída, ao utilizarmos o nível `ALLSTATS LAST` como opção no `display_cursor`, e o `hint` `GATHER_PLAN_STATISTICS` na instrução SQL.

```
SQL> select /*+ GATHER_PLAN_STATISTICS */
  2     e.first_name
  3    ,e.last_name
  4    ,l.city
  5    ,sum(END_DATETIME - START_DATETIME) qtd_horas
  6  from   employees e
  7        ,timesheet t
  8    ,locations l
  9  where  1=1
 10  and    e.employee_id = t.employee_id
 11  and    t.location_id = l.location_id
 12  group  by e.first_name
 13       ,e.last_name
 14       ,l.city
 15  order  by e.first_name;
...
..
.
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALL
STATS LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------
-------------------------------------------------------------------
-------
SQL_ID  0jy2x6ud8y3jz, child number 0
-------------------------------------
select /*+ GATHER_PLAN_STATISTICS */    e.first_name   ,e.last_na
me
,l.city   ,sum(END_DATETIME - START_DATETIME) qtd_horas from
employees e       ,timesheet t   ,locations l where  1=1 and
e.employee_id = t.employee_id and    t.location_id = l.location_i
d
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
group  by e.first_name      ,e.last_name      ,l.city order  by
e.first_name
```

```
Plan hash value: 3456199970

-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation            | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |  OMem |  1Mem | Used-Mem |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |      1 |        |   5729 |00:00:01.02 |    4712 |   1865 |       |       |          |
|   1 |  SORT GROUP BY       |           |      1 |   309K|   5729 |00:00:01.02 |    4712 |   1865 |   690K|   690K|  613K (0)|
|*  2 |   HASH JOIN          |           |      1 |   309K|    288K|00:00:00.67 |    4712 |   1865 |  1000K|  1000K| 1238K (0)|
|   3 |    TABLE ACCESS FULL | EMPLOYEES |      1 |  1075 |   1075 |00:00:00.01 |      18 |      0 |       |       |          |
|*  4 |    HASH JOIN         |           |      1 |   309K|    288K|00:00:00.59 |    4694 |   1865 |  1156K|  1156K| 1106K (0)|
|   5 |     TABLE ACCESS FULL| LOCATIONS |      1 |    23 |     23 |00:00:00.01 |       3 |      0 |       |       |          |
|   6 |     TABLE ACCESS FULL| TIMESHEET |      1 |  1162K|   1000K|00:00:00.43 |    4691 |   1865 |       |       |          |
-----------------------------------------------------------------------------------------------------------------------
...
..
.
```

Figura 12.22: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img20code.txt

Nessa execução, foi utilizada a tabela `timesheet` como um auxílio para o exemplo. Foi selecionado o total de número de horas registradas pelo empregado, por localidade, em uma base de registro de horas de 1.000.000 de registros. Note que na instrução SQL declaramos o `hint` `GATHER_PLAN_STATISTICS` para que estatísticas adicionais fossem acrescentadas ao plano de execução. Para esse exemplo, foram suprimidas outras saídas e o próprio resultado da instrução SQL, apenas para não poluir a demonstração. Observe que, para esta saída, dados como `Starts` e `A-Rows` são mostrados. Vale ressaltar que, para que essas estatísticas adicionais possam ser mostradas, além da escolha do nível de saída no `display_cursor`, como a saída do tipo `'ALLSTATS LAST'`, é necessário ativar uma das três formas a seguir para a coleta delas:

- `/*+ GATHER_PLAN_STATISTICS */`
- `sql_trace = true;`
- `statistics_level = 'ALL'`

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nesse exemplo, foi utilizada a primeira forma, indicando o `hint` em nossa instrução SQL. A seguir, veja como cada uma destas formas podem ser configuradas:

```
-- Forma 1: Hint GATHER_PLAN_STATISTICS

SQL> select /*+ GATHER_PLAN_STATISTICS */
  2         e.employee_id
  3        ,e.last_name
  4        ,d.department_name
  5        ,e.salary
  6  from   employees e
  7        ,departments d
  8  where  e.department_id = d.department_id
  9  and    d.department_id = 60
 10  and    e.job_id = 'IT_PROG';

SQL>

-- Forma 2: sql_trace = true

SQL> alter session set sql_trace=true;

Sessão alterada.

SQL>

-- Forma 3: STATISTICS_LEVEL = ALL

SQL> alter session set statistics_level = 'ALL';

Sessão alterada.

SQL>
```

Também com relação às informações adicionais `Starts` e `A-Rows`, elas podem ainda ser avaliadas juntamente à coluna `E-Rows`. Comparando essas duas colunas, é possível constatar rapidamente se o otimizador está executando a consulta com base em estatísticas mais próximas da realidade. Se a discrepância entre essas colunas for considerável, isso pode ser um sinal de um plano

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

de execução não muito eficaz e não condizente com a realidade dos dados. Para se chegar ao resultado para a comparação, você precisa, primeiro, multiplicar o valor da coluna `Starts` pelo valor da coluna `E-Rows`, depois disso, comparar com o valor total da coluna `A-Rows`. Eles devem estar próximos. Vamos ver o significado de cada coluna adicional do plano mais adiante.

Outro detalhe importante, mas este com relação às estatísticas de memória estarem disponíveis, e assim poderem ser visualizadas no plano de execução, é o gerenciamento de memória PGA estar ativado. Não que esteja na alçada de um desenvolvedor configurá-lo ou alterá-lo, mas apenas para conhecimento. Aqui, mais uma vez, recomendo o suporte do DBA responsável pelo banco de dados.

Além das informações estendidas sobre `E-Rows`, `A-Rows` e `Starts`, existem outras que aparecem quando usamos a saída `ALLSTATS` e `GATHER_PLAN_STATISTICS`. Vamos ver a seguir o que são estas informações:

- `Starts`: mostra o número de vezes que uma etapa ou passo específico é executado. Na maioria das vezes, este número é 1, mas no caso de um loop aninhado, por exemplo, provavelmente será um número maior. Esse valor se trata do número real de vezes que a operação foi executada. Portanto, ele só aparece quando as estatísticas do plano de execução forem coletadas usando o `hint /*+ GATHER_PLAN_STATISTICS */` ou quando o parâmetro `STATISTICS_LEVEL` for definido como `ALL`.
- `E-Rows`: refere-se ao número estimado de linhas. Esse valor é estimado pelo otimizador durante a fase de análise.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- `A-Rows` : refere-se ao número real de linhas. Esse número é obtido da `PLAN_STATISTICS` . Portanto, é exibido apenas quando as estatísticas do plano de execução forem coletadas.
- `A-Time` : tempo real gasto para a execução da etapa.
- `Buffers` : número de leituras consistentes realizadas para a etapa.
- `Reads` : número de leituras físicas executadas para a etapa. `Buffers` e `Reads` são visíveis apenas quando `IOSTATS` ou `ALLSTATS` são usados, e as estatísticas das tabelas são coletadas, ou ainda, em tempo de execução, quando o `hint` `/*+ GATHER_PLAN_STATISTICS */` é usado.
- `OMem` : tamanho ideal estimado de `Mem (PGA)` para executar a operação na memória.
- `1Mem` : quantidade estimada de memória necessária para executar a operação em uma passagem (gravar e ler do disco (temp) apenas uma vez). Isso é chamado de "execução de uma passagem".
- `Used-Mem` : quantidade atual de memória usada e o número de passagens necessárias. `OMem` , `1Mem` , `Used-Mem` são visíveis apenas quando há uma operação real executando e usando memória para operações de `Sorting` , por exemplo. Em conjunto, pode ser usado apenas com `MEMSTATS` .

Nota: Algumas palavras-chaves, como `MEMSTATS` ou `IOSTATS` , podem ser utilizadas sozinhas como níveis de saída, na chamada `dbms_xplan.display_cursor` .

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Se você quiser ver mais sobre as informações adicionais referentes a estatísticas quando usado o `hint GATHER_PLAN_STATISTICS` ou qualquer um dos outros dois meios utilizados para a este fim, consulte manualmente as tabelas que guardam essas informações: `v$sql_plan_statistics` e `v$sql_plan_statistics_all`.

Para finalizarmos sobre `dbms_xplan`, a seguir algumas informações mostrando o que aparece ou não aparece utilizando ou não determinados níveis de saída e palavras-chaves.

Níveis de saída que fornecem apenas estimativas do otimizador:

	Rows	Bytes	TempSpc	Cost	Time
BASIC					
null	*	*		*	
TYPICAL	*	*	*	*	*
SERIAL	*	*	*	*	*
ALL	*	*		*	*
ADVANCED	*	*		*	*

Aqui, temos:

- `Rows`: número estimado de linhas retornadas pela operação.
- `Bytes`: quantidade estimada de dados retornados pela operação.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- `TempSpc` : quantidade estimada de espaço temporário usado pela operação.
- `Cost` : custo estimado da operação. A porcentagem do custo da CPU é dada entre parênteses. Esse valor vai sendo acumulado no decorrer do plano de execução. Em outras palavras, o custo das operações pai contém o custo das operações filho.
- `Time` : tempo estimado necessário para executar a operação (HH:MM:SS).

Níveis de saída que fornecem estatísticas reais de execuções, em conjunto com valores estimados (valores estimados representados por "o"). Aqui foram separados em quatro tabelas para melhor visualizar as informações estatísticas:

	E-Rows	Starts	A-Rows	Buffers
`MEMSTATS`	o	*	*	
`ALLSTATS`	o	*	*	*
`RUNSTATS_LAST`	o	LAST	LAST	LAST
`RUNSTATS_TOT`	o	TOT	TOT	TOT
`IOSTATS`	o	*	*	*

	Reads	Writes	A-Time
`MEMSTATS`			*
`ALLSTATS`	*	*	*
`RUNSTATS_LAST`	LAST	LAST	LAST
`RUNSTATS_TOT`	TOT	TOT	TOT
`IOSTATS`	*	*	*

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

	0Mem	1MEM	0/1/M
MEMSTATS	o	o	TOT
ALLSTATS	o	o	TOT
RUNSTATS_LAST			
RUNSTATS_TOT			
IOSTATS			

	Used-Mem	Used-Tmp	Max-Tmp
MEMSTATS	LAST	LAST	TOT
ALLSTATS	LAST	LAST	TOT
RUNSTATS_LAST			
RUNSTATS_TOT			
IOSTATS			

Nelas:

- `Writes` : número de gravações físicas executadas durante a execução.
- `A-Time` : quantidade real de tempo gasto na execução da operação (HH:MM:SS.FF).
- `0/1/M` : número de vezes que a execução foi executada no modo `optimal/one-pass/multipass` .
- `Used-Tmp` : quantidade de espaço temporário usado pela operação durante a última execução. Esse valor deve ser multiplicado por 1.024 para ser consistente com as outras colunas de utilização de memória (por exemplo, 32K significa 32MB).
- `Max-Tmp` : quantidade máxima de espaço temporário

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

usado pela operação. Esse valor deve ser multiplicado por 1.024 para ser consistente com as outras colunas de utilização de memória (por exemplo, 32K significa 32MB).

As demais colunas já foram explicadas anteriormente.

Em outros níveis, como em operações paralelas e distribuídas, podem aparecer as seguintes informações:

	TQ	IN-OUT	PQ Distrib	Pstart	Pstop	Instance
PARTITION	*	*	*			
PARALLEL				*	*	
REMOTE						*

Aqui:

- `TQ` : para processamento paralelo, representa a fila da tabela usada para a comunicação entre escravos.
- `IN-OUT` : relacionamento entre operações paralelas ou distribuídas.
- `PQ Distrib` : para processamento paralelo, representa a distribuição usada pelos `PRODUCERS` para enviar dados aos `CONSUMERS` .
- `Pstart` : representa a primeira partição usada na operação.
- `Pstop` : representa a última partição usada na operação.
- `Instance` : para processamento distribuído, representa o nome do link do banco de dados usado pela operação.

A seguir, veja a lista de níveis e as informações não estatísticas que aparecem em cada um deles:

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

	Predicate info	Note	Query Block Name
BASIC			
Null	*	*	
TYPICAL	*	*	
SERIAL	*	*	
ALL	*	*	*
ADVANCED	*	*	*
PREDICATES	*		
NOTE		*	
ALIAS			*
PROJECTION			
OUTLINES			
PEEK_BINDS			

	Column projection info	Outline	Bind vars
BASIC			
Null			
TYPICAL			
SERIAL			
ALL	*		
ADVANCED	*	*	*
PREDICATES			
NOTE			
ALIAS			
PROJECTION	*		
OUTLINES		*	

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

PEEK_BINDS			*

Para finalizar, uma amostra mais completa dos níveis e palavras-chaves que podemos utilizar para mostrar ou inibir certas informações estatísticas e não estatísticas de uma instrução SQL. Observe a linha de comando a seguir primeiramente:

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALL
STATS LAST +PEEKED_BINDS +PROJECTION +ALIAS +COST +BYTES +PARALLE
L +PARTITION +REMOTE'));
```

A saída gerada utilizando esse nível de detalhamento e com essas palavras-chaves, quando estivermos executando para uma instrução SQL cujas características condizem com todos estes atributos informados, será uma grade bem completa com diversas informações (inclusive, com informações estendidas, reais, caso os critérios para isso tenham sido obedecidos).

A seguir, veja um quadro que identifica quais são estas informações com relação a cada nível e palavra-chave, e como removê-las ou incluí-las explicitamente em outros tipos de saída, caso queira dar ênfase a apenas uma ou outra informação.

Nível ou palavra-chave	Ação	Informação	Localização
-ROWS	Remove	E-Rows	Na grade do plano de execução.
+PARTITION	Adiciona	Pstart e Pstop	Na grade do plano de execução.
-IOSTATS	Remove	Buffers e Reads	Na grade do plano de execução.
+BYTES	Adiciona	E-Bytes	Na grade do plano de

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

			execução.
`-MEMSTATS`	Remove	OMem, 1Mem e Used-Mem	Na grade do plano de execução.
`+ALIAS`	Adiciona	Query Block Name / Object Alias (identified by operation id)	Abaixo da grade do plano de execução.
`+COST`	Adiciona	Cost (%CPU)	Na grade do plano de execução.
`+PARALLEL`	Adiciona	TQ, IN-OUT e PQ Distrib	Na grade do plano de execução.
`-PREDICATE`	Remove	Predicate Information (identified by operation id)	Abaixo da grade do plano de execução.
`+PROJECTION`	Adiciona	Column Projection Information (identified by operation id)	Abaixo da grade do plano de execução.
`+REMOTE`	Adiciona	Remote SQL Information (identified by operation id)	Abaixo da grade do plano de execução.
`-NOTE`	Remove	Note	Abaixo da grade do plano de execução.

Na sequência, falaremos sobre o `trace` e `tkprof`, que são duas ferramentas muito interessantes e que são utilizadas para varrer sessões inteiras do banco de dados, coletando informações sobre as estatísticas reais das várias instruções SQL que possam ter sido executadas nelas, durante um período de tempo.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

12.6 RASTREIE E VISUALIZE COM TRACE E TKPROF

Resumidamente, `trace` e `tkprof` são dois utilitários que utilizamos em conjunto, cada qual com seu objetivo. Podemos utilizar o `trace` para rastrear desde a execução de uma instrução SQL, até tudo que esteja sendo executado em uma sessão inteira do banco de dados. Já o `tkprof` é utilizado para formatarmos o arquivo de trace gerado, organizando a imensa gama de informações que ele coleta.

Através dessas ferramentas, podemos visualizar tanto os dados de apenas uma instrução SQL, quanto de um programa inteiro que foi executado. Podemos ver isso de algumas formas, como por instruções ordenadas pelo maior tempo de execução ou por aquelas que mais acessaram memória ou disco. A seguir, vamos ver como funciona cada um desses recursos.

Rastreando com trace

Utilizamos o `trace` para localizar gargalos em potencial em um sistema através da análise dos relatórios gerados por este utilitário, que mostram dados estatísticos referentes às execuções realizadas no banco de dados. Abordaremos o uso do `trace`, no que diz respeito ao escopo de instruções SQL.

As informações geradas pelo `trace`, em geral, são guardadas em um arquivo que chamamos de *arquivo de trace*, que deve ser formatado pelo `tkprof`, pelo fato de sua leitura não ser amigável. Quando o `trace` é ativado, as estatísticas de desempenho de todas as instruções SQL executadas em uma sessão ou instância do

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

usuário são inseridas em arquivos de rastreamento. Com relação ao `trace`, podemos dizer que:

- Podemos ativá-lo para uma instância ou sessão;
- Podemos obter relatórios sobre estatísticas de volume e tempo para as fases: `Parse`, `Execute` e `Fetch`;
- Podemos produzir saídas que podem ser formatadas pelo `tkprof`.

Este utilitário é muito útil, pois podemos utilizá-lo para gerar, por exemplo, dois planos de execução referentes às mesmas instruções SQL (escritas de formas diferentes, mas que retornem o mesmo resultado), executando-as e comparando suas estatísticas para ver quais dos planos têm o melhor desempenho. Veja algumas das estatísticas que podem ser analisadas:

- Contagens de `Parse`, `Execute` e `Fetch`;
- CPU e tempos decorridos;
- Leituras físicas e leituras lógicas;
- Número de linhas processadas;
- Falta de instruções no cache da biblioteca;
- Nome de usuário sob o qual cada análise ocorreu e outros.

O `trace` pode ser executado de várias formas. Assim como outros recursos do banco Oracle, esse recurso foi sendo evoluído através dos tempos e com as novas versões do banco de dados foram acrescentadas novas features. A seguir, será mostrada a forma mais básica e simples para a geração de um arquivo Trace, a geração através do parâmetro `sql_trace`. Veremos outras formas de geração na sequência.

Para configurar e executar o utilitário `trace`, algumas etapas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

devem ser seguidas. Vamos falar destas etapas a partir de agora:

Primeira etapa: Antes de utilizarmos o Trace, precisamos verificar as definições dos parâmetros `timed_statistics`, `max_dump_file_size` e `user_dump_dest` no arquivo de inicialização do Oracle (`ini.ora`). Caso seja necessário definir ou atualizar estes parâmetros, isso pode ser realizado por meio do comando `alter system`. Os parâmetros `timed_statistics` e `max_dump_file_size` podem ser configurados em nível de sessão usando o comando `alter session`.

Contudo, muito cuidado. Recomendo aqui que este trabalho seja realizado pelo DBA responsável pelo banco de dados. Embora o `trace` possa ser executado por profissionais que não conheçam sobre administração de banco de dados, é expressamente recomendado que o trabalho de preparação para o uso desses utilitários, se não está definido ainda, seja realizado pela pessoa administradora do banco de dados. Só ela poderá ter todo o conhecimento suficiente do ambiente do banco de dados, inclusive dos sistemas operacionais para executar essas configurações da melhor forma possível. O que pretendo mostrar aqui é uma forma de como isso pode ser feito, com o intuito de entendermos o processo e poder explicar o assunto de forma mais didática.

Neste exemplo, será usada a configuração em nível de sessão. Somente o parâmetro `user_dump_dest` será configurado no arquivo `ini.ora`. Para realizar este trabalho, é necessário um usuário com permissões de administrador do banco (DBA) para executar algumas destas etapas.

Mas você deve estar pensando se não existem formas ou ferramentas que permitem fazer esse trabalho de forma mais

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

rápida. Sim, existem ferramentas que possibilitam, com apenas alguns cliques, a obtenção dessas informações. Contudo, nem sempre essas ferramentas vão estar disponíveis no ambiente em que você está operando, pois, geralmente, precisam ser licenciadas ou compradas. Outro ponto é se você como desenvolvedor terá acesso a elas. Geralmente, essas ferramentas têm acesso a mais funcionalidades e informações que competem mais ao uso do DBA e, como tal, são restritas para uso deste profissional. E como mencionei, executando cada passo, vamos conhecendo como funciona esse processo e entendendo como as coisas funcionam por trás das cortinas.

Veja a sintaxe para o preenchimento dos parâmetros citados anteriormente:

```
TIMED_STATISTICS = {FALSE | TRUE}
MAX_DUMP_FILE_SIZE = N
USER_DUMP_DEST = directory_path
```

Exemplo de configuração através da alteração do arquivo `ini.ora`:

```
TIMED_STATISTICS = TRUE
MAX_DUMP_FILE_SIZE = 2000
USER_DUMP_DEST = /oracle/admin/ora11g/udump
```

Nessa configuração, temos:

- `timed_statistics`: ao definir este parâmetro como `true`, o `trace` estará incluindo informações de tempo em seus rastreios. É necessário ter cautela ao definir este parâmetro no arquivo `init.ora`, pois pode diminuir o desempenho do sistema.
- `max_dump_file_size`: especifica o tamanho do arquivo de trace com relação aos blocos do sistema operacional. Se

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

a quantidade de informação que for sendo gerada, ultrapassar o tamanho deste arquivo, estas informações serão ignoradas e não serão registradas.

- `user_dump_dest` : através deste parâmetro é definido o local onde será gravado o arquivo de `trace` .

> **NOTA:** Geralmente o `trace` é ativado por sessão, pois existe uma sobrecarga de dados sendo gerada. Dessa forma, a ativação do `trace` por instância pode não ser uma boa ideia.

> **NOTA:** Embora algumas configurações só possam ser realizadas com perfil e conhecimentos de DBA, geralmente, em um banco de dados produtivo, que já esteja operacional, tais configurações podem já ter sido realizadas. Qualquer dúvida, consulte o DBA do banco de dados.

Segunda etapa: Nesta etapa, o `trace` é habilitado e a configuração baseada em sessão é realizada. O `trace` também pode ser ativado para todas as sessões, pela configuração do arquivo `init.ora` , inserindo a seguinte linha de comando no arquivo: `sql_trace = true` . Para habilitar apenas na sessão, devem ser executadas as seguintes linhas de comando através do `SQL*Plus` :

```
SQL> alter session set sql_trace = true;

Sessão alterada.
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
SQL> alter session set timed_statistics = true;

Sessão alterada.

SQL> alter session set max_dump_file_size = 2000;

Sessão alterada.

SQL>
```

Como alternativa aos comandos anteriores, você pode utilizar a chamada `set_sql_trace` do pacote `dbms_session` para ativar e desativar o `trace`. Este é um recurso que foi implementado em versões mais recentes do banco de dados (exemplo demonstrado à frente).

> **Nota:** Para executar esses passos serão necessárias permissões de `create session` e `alter session`. Os valores utilizados aqui foram meramente de cunho didático para demostrar exemplos de como a configuração pode ser realizada. Mais uma vez saliento a orientação de um profissional com os conhecimentos necessários, principalmente se estiver lidando com ambientes produtivos ou corporativos.

Terceira etapa: Esta etapa consiste em executar a instrução SQL para que o `trace` possa rastreá-la.

```
SQL> select e.employee_id, e.first_name, d.department_name
  2  from   employees e, departments d
  3  where  e.department_id = d.department_id;

EMPLOYEE_ID FIRST_NAME           DEPARTMENT_NAME
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
----------- -------------------- ------------------------------
        100 Steven               Executive
        101 Neena                Executive
        102 Lex                  Executive
...
...

SQL>
```

> **Nota:** Vale lembrar que, para conseguirmos verificar os dados estatísticos de forma confiável e completa, precisamos que a instrução SQL tenha sua execução finalizada entre o habilitar e o desabilitar do `trace`, ou seja, a instrução SQL precisa executar até o final.

Quarta etapa: Depois de rastreada a instrução SQL, o `trace` deve ser desativado na sessão. Esta ação não é necessariamente obrigatória, entretanto é recomendado que seja realizada para não comprometermos a performance do sistema e também para não termos um arquivo gigante sendo gerado no disco. Além disso, tudo o que executarmos depois da instrução será considerado no arquivo de trace até desabilitarmos, podendo gerar dados desnecessários ao foco do trabalho de análise.

```
SQL> alter session set sql_trace = false;

Sessão alterada.

SQL>
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Para descobrir traces habilitados no sistema, você poderá consultar a visão dba_enabled_traces .

Depois de executada a instrução SQL, o trace criará um arquivo com a extensão .trc , e o nome pode variar dependendo do sistema operacional onde o trace foi executado. Ele pode variar entre: "ora_%.trc" e "ora%.trc" , como "ora_19554.trc" . Quando o arquivo é gerado, ele leva no nome o process_id da sessão do trace . A localização deste arquivo pode acontecer de quatro formas:

1. Chegando até o caminho especificado no parâmetro user_dump_dest . Execute no SQL*PLUS : "show parameter user_dump_dest" ou "show parameter background_dump_dest" para listar os locais dos arquivos de trace referentes aos processos do usuário ou de background , respectivamente. Nesses locais, o arquivo de trace pode ser encontrado pela data e hora de criação ou atualização do arquivo. Os arquivos de trace de processos de usuário serão os arquivos que interessam para nós aqui. Já os arquivos de trace relacionados a background são os gerados pelo Oracle quando ocorre algum problema no banco de dados. Este tipo de trace é gerado automaticamente pelo banco.

2. Inserindo um marcador dentro do arquivo. Algo como "select '20191207-1' identificador_trace from dual" . Desta forma, a localização pode ser feita pelo search do sistema operacional;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

3. Também podemos configurar o parâmetro `tracefile_identifier` incluindo prefixos no nome do arquivo de `trace`. Exemplo: `“alter session set tracefile_identifier = "trace_032020";”` para configurar um identificador, e `“show parameter tracefile_identifier”` para visualizar o valor configurado;

4. Através da consulta das visões `V$` (`views` internas do Oracle), pesquisando na mesma sessão onde foi executado o `trace`.

Vamos pelo exemplo de localização através das visões `V$`. Utilizando a instrução SQL a seguir, obtemos o número (coluna `SPID`) utilizado pelo Oracle na geração do nome do arquivo de `trace`. Veja a instrução SQL a seguir sendo executada na mesma sessão onde o `trace` foi executado.

```
SQL> select spid, s.sid, s.serial#, p.username, p.program
  2  from v$process p, v$session s
  3  where p.addr = s.paddr
  4  and s.sid = (select sid
  5               from v$mystat
  6               where rownum = 1);

SPID                            SID    SERIAL# USERNAME        PR
OGRAM
------------------------ ---------- ---------- --------------- --
---------------------------
6396                             10        143 SISTEMA         OR
ACLE.EXE (SHAD)

SQL>
```

Com o número do `SPID`, fica mais fácil encontrar o arquivo no local definido no parâmetro `user_dump_dest`. Quanto ao nome exato do arquivo, ele pode ser parecido com o dos nossos exemplos. Tudo dependerá das configurações e do ambiente. Para

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

verificar onde o arquivo `trace` é gerado, podemos executar a instrução SQL a seguir:

```
SQL> select name, value from v$parameter where name like '%user_d
ump_dest%';

NAME                      VALUE
------------------------- ----------------------------------------
-----
user_dump_dest            C:\oraclexe\app\oracle\diag\rdbms\xe\xe\
trace


SQL>
```

Ou você pode verificar isso de maneira mais fácil através do comando `show parameter`, como já foi mencionado:

```
SQL> show parameter user_dump_dest

NAME                          TYPE        VALUE
----------------------------- ----------- --------------------------
----
user_dump_dest                string      C:\oraclexe\app\oracle\dia
g\rdbms\xe\xe\trace

SQL>
```

Veja como ficou o nome do nosso arquivo, utilizado para os testes deste livro.

```
C:\oraclexe\app\oracle\diag\rdbms\xe\xe\trace\xe_ora_6396.trc
```

> **Nota:** Uma das informações que compõem o nome do arquivo é o número da sessão (6396), identificado na coluna SPID, obtido através do resultado da instrução SQL executada, anteriormente.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nas versões mais novas do banco de dados, você poderá utilizar a `view v$diag_info` para localizar onde os arquivos de `trace` estão sendo gerados ou qual foi o arquivo gerado na sessão corrente, por exemplo. Veja exemplos a seguir:

```
SQL> select value
  2  from   v$diag_info
  3  where  name = 'Default Trace File';

VALUE
-----------------------------------------------------------------
--------------------------
C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_2920.trc
```

Essa instrução SQL retorna a localização do arquivo `trace` para a sessão corrente. Já a instrução a seguir dá o local de todos os arquivos de rastreio para a instância atual.

```
SQL> select value
  2  from   v$diag_info
  3  where  name = 'Diag Trace';

VALUE
-----------------------------------------------------------------
--------------------------
C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace
```

Já para localizar arquivos de `trace` referentes a todo o processo atual do banco de dados, você poderá consultar a `view v$process`:

```
SQL> select pid
  2        ,program
  3        ,tracefile
  4  from   v$process
  5  /

       PID PROGRAM            TRACEFILE
---------- ------------------ ----------------------------------
--------------------------------------------------
         1 PSEUDO             C:\ORACLEXE\APP\ORACLE\diag\rdbms\x
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
e\xe\trace\xe_ora_0.trc
         2 ORACLE.EXE (PMON)  C:\ORACLEXE\APP\ORACLE\diag\rdbms\x
e\xe\trace\xe_pmon_4068.trc
         3 ORACLE.EXE (PSP0)  C:\ORACLEXE\APP\ORACLE\diag\rdbms\x
e\xe\trace\xe_psp0_3196.trc
         4 ORACLE.EXE (VKTM)  C:\ORACLEXE\APP\ORACLE\diag\rdbms\x
e\xe\trace\xe_vktm_4016.trc
...
...
```

Através das informações `SERIAL` e `ID`, é possível localizar o arquivo de `trace` referente a uma sessão específica, através de uma instrução SQL. Essas informações são referentes à sessão do usuário. Veremos exemplos mais à frente.

Como mencionado anteriormente, existem outras formas de habilitar e gerar arquivos `trace`. No entanto, atente-se para o pacote `dbms_monitor` (veremos exemplos posteriormente), pois este pacote é o mais indicado pela Oracle para as versões mais novas do banco de dados. Veja a seguir formas de habilitar e gerar arquivos de `trace`:

- Ferramenta (Comando/Package): `oradebug`
  - A partir da versão: versão 8.1
  - Característica: pouco documentada e mais utilizada em situações específicas.

```
SQL> oradebug setospid 10121
Oracle pid: 91, Unix process pid: 10121, image: oracleorcl
SQL> oradebug EVENT 10046 trace name context forever, level 12
Statement processed.
SQL> ! vi /app/oracle/admin/orcl/bdump/ora_10121.trc
```

- Ferramenta (Comando/Package): `dbms_support`
  - A partir da versão: Versão 8.1.
  - Característica: não vem instalada por padrão e permite habilitar `sql trace` somente no nível da sessão.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

```
-- Habilitando o trace
SQL> begin dbms_support.start_trace(true, true); end;

-- Desabilitando o trace
SQL> begin dbms_support.stop_trace; end;
```

- Ferramenta (Comando/Package): `dbms_session`
  - A partir da versão: Versão 9i.
  - Característica: é mais robusta e possui mais recursos que as opções anteriores, como a geração de `trace` por grupos de sessão.

```
-- Habilitando o trace
SQL> begin dbms_session.set_sql_trace(true); end;

-- Configurando identificador
SQL> begin dbms_session.set_identifier('user0001'); end;

-- Desabilitando o trace
SQL> begin dbms_session.set_sql_trace(false); end;
```

- Ferramenta (Comando/Package): `dbms_system`
  - A partir da versão: Versão 8i.
  - Característica: funcionamento parecido com o do `dbms_session`, com a diferença de possibilitar estender seu escopo para qualquer sessão do banco de dados, não somente limitado à sessão atual.

```
-- Habilitando o trace em outro usuário:
SQL> begin dbms_system.set_sql_trace_in_session(sid => 15, serial
# => 2388, true); end;

-- Desabilitando o trace ref. a outro usuário:
SQL> begin dbms_system.set_sql_trace_in_session(sid => 15, serial
# => 2388, false); end;
```

Nesse caso, você deve passar as informações da sessão do usuário.

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Ferramenta (Comando/Package): `dbms_monitor`
  - A partir da versão: Versão 10G.
  - Característica: esta package é a recomendada pela Oracle, pois é melhor que sua anterior, `dbms_system`. Possui todas as funcionalidades da `dbms_system` e outras mais atuais relacionadas a operações mais complexas.

```
-- Habilitando o trace em outro usuário:
SQL> sql> begin dbms_monitor.session_trace_enable(session_id => 3
1, serial_num => 3598, waits => true, binds => true); end;

-- Desabilitando o trace ref. a outro usuário:
SQL> begin dbms_monitor.session_trace_disable(session_id => 31, s
erial_num => 3598); end;
```

Nesse caso, você deve passar as informações da sessão do usuário, que podem ser recuperadas através da `view` `v$session`.

Agora que vimos como gerar um arquivo de `trace`, vamos falar sobre o `tkprof` e como utilizá-lo para formatar o arquivo gerado, possibilitando analisar suas informações.

Formatando com tkprof

O arquivo gerado pelo `trace` precisa ser formatado através do utilitário `tkprof` para que possamos obter uma visualização inteligível dos dados rastreados. O `tkprof` também pode ser utilizado para a geração de planos de execução. O arquivo formatado pelo utilitário possui a extensão de arquivo `".prf"` e pode apresentar as seguintes informações:

- O texto da instrução SQL;

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

- Informações estatísticas em forma tabular;
- O número de vezes que faltou cache de dicionário para os passos referentes ao `Parse` e `Execute` para cada instrução SQL;
- O usuário que disparou a instrução SQL;
- O plano de execução.

Através dessas informações, podemos facilmente localizar as instruções que estão usando maior recurso do sistema, e tomar as ações necessárias para melhorar o cenário, se for o caso.

Exemplo de sintaxe para a execução do utilitário `tkprof` :

```
tkprof arq1 arq2 table = usuário.plan_table EXPLAIN=usuário/senha
```

Nela:

- `arq1` : é a indicação do nome do arquivo gerado pelo `trace` ;
- `arq2` : é a indicação do nome para o arquivo formatado do `tkprof` ;
- `table` : é o schema e a tabela onde são armazenadas as informações sobre os planos de execução.

Se você estiver utilizando o `SQL*Plus` através da aplicação gráfica, é possível chamar um `prompt` de comando do sistema operacional através do comando `host` de dentro da própria ferramenta. Esse comando fará com que seja aberta uma janela de `prompt` , o qual estará setado com o caminho onde se encontra alguns dos utilitários fornecidos pela Oracle, como o próprio SQL*Plus e o `tkprof` , por exemplo.

Caso não esteja utilizando o utilitário gráfico do SQL*Plus, mas

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

esteja executando direto o atalho em modo `prompt` (utilitário `"Executar Linha de Comandos SQL"`), como é mostrado a seguir, o processo será o mesmo, só não de forma gráfica.

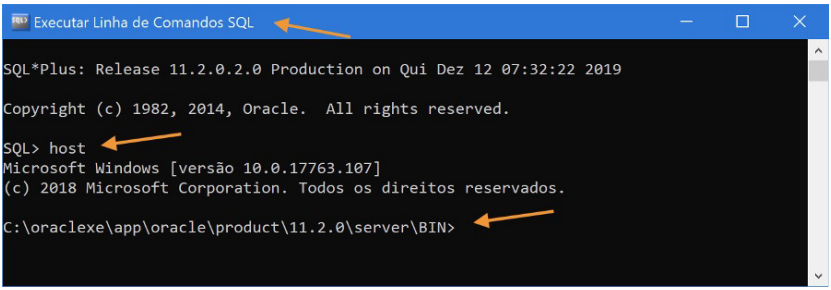

Figura 12.23: Comando host via SQL*Plus

Nesse exemplo, ao digitar o comando host, o `prompt` de comando do SQL já aparece setado no seguinte caminho: `C:\oraclexe\app\oracle\product\11.2.0\server\BIN`.

Estando no caminho onde podemos executar o utilitário `tkprof` e já tendo o caminho do arquivo de `trace`, basta executarmos o programa. Para isso, vamos utilizar aquele arquivo de trace gerado anteriormente, no tópico sobre o utilitário `trace`.

```
C:\oraclexe\app\oracle\diag\rdbms\xe\xe\trace\xe_ora_6396.trc
```

Para o exemplo, vamos digitar a seguinte linha de comando neste `prompt`:

```
tkprof C:\oraclexe\app\oracle\diag\rdbms\xe\xe\trace\xe_ora_6396.
trc C:\oraclexe\app\oracle\diag\rdbms\xe\xe\trace\xe_ora_6396.prf
 table=ttun.plan_table EXPLAIN=ttun/ttun;
```


E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Nota: Se você quiser indicar outra tabela `plan_table`, que não a utilizada pelo `tkprof`, você poderá criar uma sua (obedecendo os mesmos critérios referentes aos atributos) ou indicar um schema que a tenha. Nesse exemplo, foi indicada uma `plan_table` a que nosso usuário já possuía acesso, embora não sendo o dono dela.

Exemplo da execução de um `tkprof` no ambiente Windows:

```
Selecionar Executar Linha de Comandos SQL

SQL> host
Microsoft Windows [versão 10.0.17763.107]
(c) 2018 Microsoft Corporation. Todos os direitos reservados.

C:\oraclexe\app\oracle\product\11.2.0\server\BIN>TKPROF C:\oraclexe\app\oracle\diag\rdbms\xe\xe\trace\xe_ora_6396.trc C:
\oraclexe\app\oracle\diag\rdbms\xe\xe\trace\xe_ora_6396.prf table=ttun.plan_table EXPLAIN=ttun/ttun;

TKPROF: Release 11.2.0.2.0 - Development on Sex Dez 13 06:53:51 2019

Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All rights reserved.


C:\oraclexe\app\oracle\product\11.2.0\server\BIN>
```

Figura 12.24: Comando tkprof via SQL*Plus

Nota: Para sair do `prompt`, e sua sessão no SQL*Plus ser liberada, digite `exit`.

Veja na imagem o comando `tkprof` (linha em amarelo) sendo executado com os parâmetros fornecidos referentes ao nome e local do arquivo de `trace`, nome e local sugerido para o arquivo `tkprof`, que deve ser gerado, e demais informações

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

referentes à `plan table` que deve ser utilizada, e qual usuário e senha para a conexão com ela. Logo depois da execução da linha de comando do utilitário (linha verde) é possível ver informações referentes à execução do `tkprof`, como versão do utilitário e data da execução. Após o término, o programa é finalizado e o `prompt` de comando é mostrado novamente.

O arquivo formatado e criado pelo `tkprof` pode ser salvo em qualquer lugar que seja especificado. Para este exemplo, utilizamos o mesmo caminho de onde está o arquivo de `trace`. Caso não seja informado um caminho específico, o arquivo será gravado no mesmo diretório onde se encontra o utilitário `tkprof`. Depois de gerado, o arquivo formatado poderá ser aberto com o uso de um editor de texto comum. Parte do resultado da geração é mostrado a seguir:

```
...
...
...
********************************************************************************

select e.employee_id, e.first_name, d.department_name
from   employees e, departments d
where  e.department_id = d.department_id

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.07          0          9          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9      0.00       0.00          0         14          0         106
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      0.00       0.07          0         23          0         106

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 48
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
       106        106        106  HASH JOIN  (cr=14 pr=0 pw=0 time=167 us cost=5 size=7208 card=106)
        27         27         27   TABLE ACCESS FULL DEPARTMENTS (cr=3 pr=0 pw=0 time=16 us cost=2 size=810 card=27)
       107        107        107   TABLE ACCESS FULL EMPLOYEES (cr=11 pr=0 pw=0 time=10 us cost=2 size=4066 card=107)

********************************************************************************
...
...
...
```

Figura 12.25: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img21code.txt

E-book gerado especialmente para erlison souza - erlison.santos@guaraves.com.br

Os pontilhados antes e depois do resultado do `tkprof`, mostrados no exemplo aqui, indicam que dentro do arquivo há outras informações, além das informações referentes à instrução SQL propriamente dita. Essas informações são referentes a todos os outros passos (inclusive, instruções recursivas) que o otimizador executa até chegar ao resultado correspondente à instrução SQL analisada, estando tudo dentro do mesmo arquivo. Se você preferir, poderá escolher entre carregar ou não essas instruções dentro do arquivo formatado. Quanto aos dados do `tkprof` temos:

A instrução SQL que foi analisada pelo `trace`:

```
select e.employee_id, e.first_name, d.department_name
...
```

As informações estatísticas referentes à execução:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.07          0          9          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9      0.00       0.00          0         14          0         106
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      0.00       0.07          0         23          0         106
```

Figura 12.26: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img22code.txt

Aqui abriremos parênteses para falar um pouco sobre cada dado estatístico apresentado, para entender do que se tratam as informações apresentadas.

Coluna `CALL`:

- `Parse`: esta etapa consiste na tradução e verificação da

instrução SQL, no sentido de avaliar a sintaxe e a existência das tabelas e colunas, privilégios de usuário, entre outros. Já vimos em capítulos anteriores todos os detalhes desta fase quando falamos sobre os passos para a execução de uma instrução SQL.

- `Execute` : aqui acontece a execução propriamente dita da instrução SQL.
- `Fetch` : é nesta etapa que são recuperadas as linhas selecionadas pela instrução SQL, lembrando que o `Fetch` só ocorre quando se tratar de instruções `select` .

Coluna `COUNT` :

- Número de vezes que um comando foi analisado (`parse`), executado (`execute`) ou que retornou (`fetch`) linhas. No nosso exemplo, essas informações foram 1, 1 e 9, indicando que houve 1 operação de `parse` , 1 de execução, e 9 operações de `fetch` .

Coluna `CPU` :

- Nos mostra o total de tempo de CPU decorrido em segundos, para a execução de todos os `parses` , `executes` e `fetchs` . Se estivermos falando de instruções muito complexas, que exigem muito da CPU, por exemplo, para a execução de cálculos pesados ou qualquer outro processamento que não pode ser realizado em memória, o tempo para essas tarefas pode ser visto aqui.

Coluna `ELAPSED` :

- O total de tempo em segundos para a execução de todos os

`parses`, `executes` e `fetchs`. Ou seja, esses valores somados significam o tempo que a instrução SQL levou para ser executada. Esta informação é muito utilizada, pois nos dá a dimensão do tempo que a instrução está levando para executar e se é ela mesmo quem pode estar causando algum gargalo. Geralmente, quando mandamos gerar o arquivo pelo `tkprof`, solicitamos que ele faça isso ordenando por esta informação, para trazer, como foco, quais as instruções que estão levando mais tempo na execução.

Coluna `DISK`:

- Total de blocos de dados físicos lidos/escritos dos `datafiles` (arquivos de dados) em disco, para recuperar as informações necessárias ao processamento da instrução SQL. Quanto mais necessário acessar o disco físico para buscar estas informações, maior será esse tempo. Uma leitura `FULL TABLE SCAN` de uma tabela muito extensa que não esteja em memória pode resultar em mais leituras físicas.

Coluna `QUERY`:

- Total de `buffers` (área de memória) recuperados (leituras consistentes) para todos os `parses`, `executes` e `fetchs`. Podemos ver, nesse exemplo, um total de 23 `buffers` lidos para a execução da instrução SQL. Quanto mais os dados possam ser recuperados da memória, melhor, em relação a buscá-los do disco, por exemplo.

Coluna `CURRENT` :

- Total de blocos lidos em memória (`buffers`) referentes a dados atuais, lidos normalmente em `updates` , o que pode explicar, em nosso exemplo, o fato de esse número estar zerado.

Coluna `ROWS` :

- Número de linhas processadas/retornadas pela instrução SQL. Em nosso exemplo, a instrução SQL retorna 106 registros.

Logo após a exibição da grid de dados estatísticos da execução da instrução SQL, temos outras informações como as que estão a seguir:

```
Misses in library cache during parse: 1
```

Essa linha indica o número de falhas ao acessar o cache da biblioteca durante a análise da instrução SQL. Isso pode ocorrer por se tratar de uma instrução SQL nova ou de um cursor inválido que não pode ser lido naquele momento. Outra linha como esta, mas referente à execução, também pode aparecer: `"Misses in library cache during execute"` .

```
Optimizer mode: ALL_ROWS
```

Essa informação indica o modo de otimização utilizado pelo otimizador para a execução da instrução SQL.

```
Parsing user id: 48
```

Essa informação indica o id do usuário que executou a

instrução SQL.

```
Number of plan statistics captured: 1
```

Indica o número de estatísticas do plano que foram capturadas pelo otimizador, ou seja, diz de quantas execuções o otimizador registrou informações sobre o plano. É utilizado para preencher as colunas `Rows (1st)`, `Rows (avg)` e `Rows (max)`, em cada operação das linhas do plano (`row source operation`), no plano de execução.

Vamos nos adiantar um pouco e ver o plano de execução gerado pelo `trace`.

```
Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  -------------------------------------------------
       106        106        106  HASH JOIN  (cr=14 pr=0 pw=0 time=167 us cost=5 size=7208 card=106)
        27         27         27   TABLE ACCESS FULL DEPARTMENTS (cr=3 pr=0 pw=0 time=16 us cost=2 size=810 card=27)
       107        107        107   TABLE ACCESS FULL EMPLOYEES (cr=11 pr=0 pw=0 time=10 us cost=2 size=4066 card=107)
```

Figura 12.27: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img23code.txt

Para esse exemplo, de 1 estatística do plano que foi capturada, conforme o número indicado para este `trace` e apenas para a primeira operação do plano, tivemos:

- No primeiro processamento, 106 linhas.
- Uma média de processamento também de 106 linhas.
- O máximo de 106 linhas processadas.

Em resumo, para a primeira operação do plano de execução, sempre foram processadas 106 linhas.

```
Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
       106        106        106  HASH JOIN  (cr=14 pr=0 pw=0 time=167 us cost=5 size=7208 card=106)
        27         27         27   TABLE ACCESS FULL DEPARTMENTS (cr=3 pr=0 pw=0 time=16 us cost=2 size=810 card=27)
       107        107        107   TABLE ACCESS FULL EMPLOYEES (cr=11 pr=0 pw=0 time=10 us cost=2 size=4066 card=107)
```

Figura 12.28: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img24code.txt

Na próxima etapa, então, como já adiantamos, é mostrado o plano de execução referente à instrução SQL. Uma das funcionalidades do `tkprof` é gerar o plano de execução. Embora isso seja uma comodidade, deve-se ter atenção quanto a situações incomuns, como a possibilidade de ter sido excluído um índice depois de ter sido usado `trace` e antes de ter executado o `tkprof`. Afinal, o plano de execução será baseado no status do comando, depois da exclusão do índice e não de quando o `trace` foi realizado.

> **Nota:** Já vimos em outros capítulos, exemplos de plano de execução em diferentes formatos. Desta forma, dependendo do utilitário ou ferramenta utilizada para visualização, as mesmas informações podem ser apresentadas com formatações diferentes umas das outras, quanto à estética ou à disposição dos dados. Em tese, os dados deverão ser os mesmos, mas talvez com uma roupagem diferente.

Algumas considerações gerais sobre a geração de estatísticas mostradas no `tkprof`:

- A soma das colunas `query` e `current` é o total de `buffers` acessados;

- Para instruções `select` , o número de linhas retornadas aparece no passo `fetch` ;
- Para os comandos `update` , `delete` e `insert` , o número de linhas processadas aparece no passo `execute` ;
- Os números apresentados nos resultados das estatísticas são apresentados em centésimos de segundos. Entretanto, se alguma operação for executada em menos de um centésimo de segundo o valor mostrado será zero.
- Os arquivos de `trace` gerados imediatamente após a inicialização da instância contêm dados que refletem as atividades do processo de inicialização. Em particular, refletem uma quantidade maior de atividades de Entrada/Saída física, à medida que os caches no `SGA` são preenchidos. Para fins de monitoramento da performance, você deve ignorar essas informações.

O utilitário `tkprof` possui algumas opções que podem ser utilizadas no momento da sua execução e, dependendo dessas opções, podem influenciar na forma como o arquivo é formatado. Essas opções são como parâmetros, que podem ser ou não informados no momento que estivermos executando o `tkprof` . Para visualizarmos as opções disponíveis, você pode executar o utilitário `tkprof` na linha de comando, sem informar os parâmetros. Com isso, uma lista de todas as opções que podem ser utilizadas será mostrada. Veja um exemplo a seguir:

```
Executar Linha de Comandos SQL
Microsoft Windows [versão 10.0.17763.107]
(c) 2018 Microsoft Corporation. Todos os direitos reservados.

C:\oraclexe\app\oracle\product\11.2.0\server\BIN>tkprof
Usage: tkprof tracefile outputfile [explain= ] [table= ]
             [print= ] [insert= ] [sys= ] [sort= ]
  table=schema.tablename   Use 'schema.tablename' with 'explain=' option.
  explain=user/password    Connect to ORACLE and issue EXPLAIN PLAN.
  print=integer    List only the first 'integer' SQL statements.
  aggregate=yes|no
  insert=filename  List SQL statements and data inside INSERT statements.
  sys=no           TKPROF does not list SQL statements run as user SYS.
  record=filename  Record non-recursive statements found in the trace file.
  waits=yes|no     Record summary for any wait events found in the trace file.
  sort=option      Set of zero or more of the following sort options:
    prscnt  number of times parse was called
    prscpu  cpu time parsing
    prsela  elapsed time parsing
    prsdsk  number of disk reads during parse
    prsqry  number of buffers for consistent read during parse
    prscu   number of buffers for current read during parse
    prsmis  number of misses in library cache during parse
    execnt  number of execute was called
    execpu  cpu time spent executing
    exeela  elapsed time executing
    exedsk  number of disk reads during execute
    exeqry  number of buffers for consistent read during execute
    execu   number of buffers for current read during execute
    exerow  number of rows processed during execute
    exemis  number of library cache misses during execute
    fchcnt  number of times fetch was called
    fchcpu  cpu time spent fetching
    fchela  elapsed time fetching
    fchdsk  number of disk reads during fetch
    fchqry  number of buffers for consistent read during fetch
    fchcu   number of buffers for current read during fetch
    fchrow  number of rows fetched
    userid  userid of user that parsed the cursor


C:\oraclexe\app\oracle\product\11.2.0\server\BIN>
```

Figura 12.29: Opção de parâmetros para uso com o TKPROF

Agora vamos conhecer um pouco de cada opção e qual seu objetivo de uso:

Opções	Definição
`TRACEFILE`	Esse é o nome do arquivo `trace` contendo as estatísticas por `sql_trace` .
`OUTPUTFILE`	Esse é o nome do arquivo em que `tkprof` gravará sua saída.
`SORT = parâmetros`	Essa é a ordem em que as instruções SQL são exibidas na saída. Existem cerca de 20 opções diferentes para

	classificar a saída – você pode até mesmo combinar opções. Veja a lista de opções logo a seguir.
`PRINT = número`	Esse é o número de comandos para incluir na saída do arquivo formatado. Se essa declaração não for incluída, `tkprof` listará todos os comandos na saída.
`EXPLAIN = nome_de_usuário/senha`	Executa o `explain plan` referente às instruções SQL apresentadas no arquivo `trace`. Essa opção cria uma `plan_table` própria, de modo que o usuário precisará ter privilégios para acessá-la. Quando `tkprof` termina, essa tabela será removida.
`TABLE = nome_plan_table`	Indicação do nome da `plan_table`, caso você queria indicar uma tabela diferente da tabela criada pelo `tkprof`.
`INSERT = nome_de_arquivo`	Essa opção cria um `script` para criar uma tabela e armazenar as estatísticas do arquivo `trace` para cada instrução SQL rastreada.
`RECORD = nome_de_arquivo`	Essa opção produz um arquivo de todos as instruções SQL.
`SYS = YES/NO`	Essa opção permite solicitar que as instruções SQL recursivas (emitidas pelo usuário `SYS`) não apareçam na saída. O padrão é `YES`. A SQL recursiva normalmente inclui chamadas internas e qualquer manutenção de tabela, como a inclusão de uma `extend` em uma tabela durante uma inserção.

Algumas opção do parâmetro `SORT`, mencionado na tabela anterior:

- `FCHELA`: tempo decorrido para a execução do comando;
- `FCHCPU`: tempo de resposta de CPU da busca;
- `FCHDSK`: leituras de disco para busca;
- `FCHCU` e `FCHQRY`: leituras de memória por busca;
- `FCHROW`: número de linhas apanhadas;
- `EXEDSK`: leituras de disco durante a execução;
- `EXECU` e `EXEQRY`: leituras da memória durante a execução;
- `EXEROW`: leituras processadas durante a execução;

- `EXECPU` : tempo de CPU da execução;
- `PRSCPU` : CPU para análise;
- `PRSCNT` : número de vezes analisadas.

Agora que já vimos como gerar o `trace` e formatá-lo com o `tkprof`, vamos acompanhar exemplos dessas ferramentas em conjunto. Vamos gerar dois arquivos `tkprof`, um utilizando o pacote `dbms_system` e outro, o pacote `dbms_monitor`. Esses dois pacotes são os mais utilizados atualmente nas versões mais novas do banco de dados, sendo que este último é o mais recomendado pela Oracle, conforme já foi mencionado. Depois utilizaremos o `tkprof` para formatar esses dois arquivos.

1) Gerando o arquivo de trace utilizando o pacote `dbms_system`.

```
SQL> select b.sid, b.serial# from v$session b where b.audsid = us
erenv('sessionid');

       SID    SERIAL#
---------- ----------
        14         61

SQL> exec dbms_system.set_sql_trace_in_session(14,61,true);

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Logo no início, temos uma instrução SQL que recupera o `SID` e `SERIAL` referente à nossa sessão atual. Depois disso, é executado o procedimento `set_sql_trace_in_session` do pacote `dbms_system`, habilitando o `trace` com os seguintes parâmetros: `SID=14`, `SERIAL#=61` e `boolean TRUE` indicando que o `trace` deve ser habilitado. Logo depois disso, executamos, por exemplo, uma instrução SQL qualquer.

```
SQL> select /* PACOTE_DBMS_SYSTEM */
  2     e.first_name
  3    ,e.last_name
  4    ,l.city
  5    ,sum(end_datetime - start_datetime) qtd_horas
  6  from   employees e
  7        ,timesheet t
  8    ,locations l
  9  where  1=1
 10  and    e.employee_id = t.employee_id
 11  and    t.location_id = l.location_id
 12  group  by e.first_name
 13       ,e.last_name
 14       ,l.city
 15  order  by e.first_name
 16  /
```

Note que identificamos a instrução SQL com o comentário "PACOTE_DBMS_SYSTEM". Isso é apenas para, quando formatarmos o arquivo via tkprof, podermos detectar a instrução SQL dentro dele. Vale lembrar que aqui executamos apenas uma instrução SQL, no entanto poderíamos executar várias outras instruções e/ou programas PL/SQL, por exemplo.

Logo após a execução da instrução, vamos desabilitar o trace.

```
SQL> exec dbms_system.set_sql_trace_in_session(14,61,false);

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Depois de o trace ser desabilitado, podemos recuperar a localização do arquivo, através de duas formas.

```
-- 1) Através da `view` `v$diag_info`:

SQL> select value
  2  from   v$diag_info
  3  where  name = 'Default Trace File';
```

```
VALUE
-----------------------------------------------------------------
---------------------------
C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_2920.trc

SQL>


-- 2) Ou através da instrução SQL abaixo:

SQL> select c.value || `'\'` || d.instance || '_ora_' ||ltrim(to_
char(a.spid,'fm99999')) || '.trc' arquivo_trace
  2  from  v$process   a
  3        ,v$session   b
  4        ,v$diag_info c
  5        ,v$thread    d
  6  where  a.addr    = b.paddr
  7  and    c.name    = 'Diag Trace'
  8  --and   b.audsid = userenv('sessionid')
  9  and    b.SID     = 14
 10  and    b.SERIAL# = 61;

ARQUIVO_TRACE
-----------------------------------------------------------------
---------------------------
C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_2920.trc

SQL>
```

Você poderá utilizar a primeira opção caso esteja conectado à mesma sessão onde o trace foi gerado. Caso você esteja gerando um `trace` de outra sessão, utilize a opção 2, informando os dados `SID` e `SERIAL#`. Esses dados podem ser recuperados pela `view` `V$SESSION`, conforme já mostrado anteriormente. Agora, vamos gerar um novo arquivo `trace`, mas agora utilizando o pacote `dbms_monitor`.

2) Gerando o arquivo de trace utilizando o pacote `dbms_monitor`.

```
SQL> conn ttun/ttun@xe;
Conectado.

SQL> select b.sid, b.serial# from v$session b where b.audsid = us
erenv('sessionid');

       SID    SERIAL#
---------- ----------
        16        715

SQL> begin
  2    dbms_monitor.session_trace_enable(session_id => 16, serial
_num => 715, waits => true, binds => true);
  3  end;
  4  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Primeiramente, realizamos uma nova conexão para que uma nova sessão fosse criada. Depois disso, temos uma instrução SQL que recupera o `SID` e `SERIAL` referente à nossa sessão atual, como fizemos anteriormente no outro exemplo. Depois disso, é executado o procedimento `session_trace_enable` do pacote `dbms_monitor`, habilitando o `trace` com os seguintes parâmetros: `SESSION_ID=16`, `SERIAL_NUM=715`, `WAITS=True` e `BINDS=True`. Logo depois disso, executamos uma instrução SQL qualquer, como também fizemos anteriormente.

```
SQL> select /* PACOTE_DBMS_MONITOR */
  2         e.first_name
  3        ,e.last_name
  4        ,l.city
  5        ,sum(END_DATETIME - START_DATETIME) qtd_horas
  6  from   employees e
  7        ,timesheet t
  8        ,locations l
  9  where  1=1
 10  and    e.employee_id = t.employee_id
 11  and    t.location_id = l.location_id
```

```
 12  group  by e.first_name
 13        ,e.last_name
 14        ,l.city
 15  order  by e.first_name
 16  /
```

Nesse exemplo, identificamos a instrução SQL com o comentário `"pacote_dbms_monitor"`. Novamente, isso servirá apenas para, quando formatarmos o arquivo via `tkprof`, podermos detectar a instrução SQL dentro dele. Aqui também vale lembrar que executamos apenas uma instrução SQL, no entanto poderíamos executar várias outras instruções e/ou programas PL/SQL, por exemplo. Logo após a execução da instrução, vamos desabilitar o `trace`.

```
SQL> begin
  2    dbms_monitor.session_trace_disable(session_id => 16, seria
l_num => 715);
  3  end;
  4  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note que o pacote `dbms_monitor` tem um procedimento para cada ação: `session_trace_enable` habilita o `trace`, e o `session_trace_disable` desabilita o `trace`.

Depois de o `trace` ser desabilitado, podemos recuperar a localização do arquivo através de duas formas, da mesma forma como fizemos anteriormente.

```
-- 1) Através da `view v$diag_info`:

SQL> select value
  2  from   v$diag_info
  3  where  name = 'Default Trace File';
```

```
VALUE
-----------------------------------------------------------------
---------------
C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_5408.trc

SQL>

-- 2) Ou através da instrução SQL abaixo:

SQL> select c.value || `'\'` || d.instance || '_ora_' ||ltrim(to_
char(a.spid,'fm99999')) || '.trc' arquivo_trace
  2  from  v$process   a
  3        ,v$session   b
  4        ,v$diag_info c
  5        ,v$thread    d
  6  where  a.addr    = b.paddr
  7  and    c.name    = 'Diag Trace'
  8  --and   b.audsid = userenv('sessionid')
  9  and    b.SID     = 16
 10  and    b.SERIAL# = 715;

ARQUIVO_TRACE
-----------------------------------------------------------------
---------------
C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_5408.trc

SQL>
```

As características mencionadas anteriormente, para o primeiro exemplo, também são válidas aqui.

Agora partiremos para a formatação dos dois arquivos. Primeiro, vamos formatar o primeiro arquivo gerado através do pacote `dbms_system`.

3) Formatando o arquivo de `trace` gerado pelo pacote `dbms_system`.

```
-- Arquivo trace gerado pelo pacote `dbms_system`.

C:\>tkprof C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_2
920.trc C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_2920
```

```
.prf table=ttun.plan_table EXPLAIN=ttun/ttun;

tkprof: Release 11.2.0.2.0 - Development on Qui Abr 30 00:29:43 2
020

Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All righ
ts reserved.

C:\>
```

A seguir, parte do trecho referente à instrução SQL que executamos na sessão.

```
...
select /* PACOTE_DBMS_SYSTEM */
   e.first_name
  ,e.last_name
  ,l.city
  ,sum(END_DATETIME - START_DATETIME) qtd_horas
from   employees e
      ,timesheet t
  ,locations l
where  1=1
and    e.employee_id = t.employee_id
and    t.location_id = l.location_id
group  by e.first_name
     ,e.last_name
     ,l.city
order  by e.first_name
```

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.04          0         24          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      383      0.26       0.29       4672       4694          0        5729
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      385      0.28       0.33       4672       4718          0        5729

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 48
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
      5729       5729       5729  SORT ORDER BY (cr=4694 pr=4672 pw=0 time=283905 us cost=2654 size=696399 card=14817)
      5729       5729       5729   HASH GROUP BY (cr=4694 pr=4672 pw=0 time=267335 us cost=2654 size=696399 card=14817)
      5729       5729       5729    HASH JOIN  (cr=4694 pr=4672 pw=0 time=266120 us cost=2301 size=696399 card=14817)
      1075       1075       1075     TABLE ACCESS FULL EMPLOYEES (cr=17 pr=0 pw=0 time=70538 us cost=6 size=22575 card=1075)
      5729       5729       5729     VIEW  VW_GBC_9 (cr=4677 pr=4672 pw=0 time=255645 us cost=2294 size=385242 card=14817)
      5729       5729       5729      HASH GROUP BY (cr=4677 pr=4672 pw=0 time=255262 us cost=2294 size=503778 card=14817)
    288760     288760     288760       HASH JOIN  (cr=4677 pr=4672 pw=0 time=144252 us cost=1278 size=9817840 card=288760)
        23         23         23        TABLE ACCESS FULL LOCATIONS (cr=2 pr=0 pw=0 time=402 us cost=2 size=276 card=23)
    288760     288760     288760        TABLE ACCESS FULL TIMESHEET (cr=4675 pr=4672 pw=0 time=58345 us cost=1275 size=6352720 card=288760)

********************************************************************************

SQL ID: cdxcgw0zwfam3 Plan Hash: 0
...
```

Figura 12.30: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img25code.txt

Vamos agora partir para a formatação do segundo arquivo, gerado através do pacote `dbms_monitor`.

4) Formatando o arquivo de `trace` gerado pelo pacote `dbms_monitor`.

```
-- Arquivo trace gerado pelo pacote `dbms_monitor`.

C:\>tkprof C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_5
408.trc C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_5408
.prf table=ttun.plan_table EXPLAIN=ttun/ttun;

tkprof: Release 11.2.0.2.0 - Development on Qui Abr 30 00:42:02 2
020

Copyright (c) 1982, 2009, Oracle and/or its affiliates.  All righ
ts reserved.

C:\>
```

A seguir, parte do trecho referente à instrução SQL que executamos na sessão.

```
...
select /* PACOTE_DBMS_MONITOR */
   e.first_name
  ,e.last_name
  ,l.city
  ,sum(END_DATETIME - START_DATETIME) qtd_horas
from   employees e
      ,timesheet t
  ,locations l
where  1=1
and    e.employee_id = t.employee_id
and    t.location_id = l.location_id
group  by e.first_name
     ,e.last_name
     ,l.city
order  by e.first_name
```

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      383      0.06       0.29       4672       4694          0        5729
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      385      0.06       0.29       4672       4694          0        5729

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 48
Number of plan statistics captured: 1
```

Figura 12.31: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img26code.txt

```
Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------------
      5729       5729       5729  SORT ORDER BY (cr=4694 pr=4672 pw=0 time=277548 us cost=2654 size=696399 card=14817)
      5729       5729       5729   HASH GROUP BY (cr=4694 pr=4672 pw=0 time=269428 us cost=2654 size=696399 card=14817)
      5729       5729       5729    HASH JOIN  (cr=4694 pr=4672 pw=0 time=267984 us cost=2301 size=696399 card=14817)
      1075       1075       1075     TABLE ACCESS FULL EMPLOYEES (cr=17 pr=0 pw=0 time=786 us cost=6 size=22575 card=1075)
      5729       5729       5729     VIEW  VW_GBC_9 (cr=4677 pr=4672 pw=0 time=265985 us cost=2294 size=385242 card=14817)
      5729       5729       5729      HASH GROUP BY (cr=4677 pr=4672 pw=0 time=265347 us cost=2294 size=503778 card=14817)
    288760     288760     288760       HASH JOIN  (cr=4677 pr=4672 pw=0 time=144421 us cost=1278 size=9817840 card=288760)
        23         23         23        TABLE ACCESS FULL LOCATIONS (cr=2 pr=0 pw=0 time=13 us cost=2 size=276 card=23)
    288760     288760     288760        TABLE ACCESS FULL TIMESHEET (cr=4675 pr=4672 pw=0 time=65918 us cost=1275 size=6352720 card=288760)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                     383        0.00          0.00
  Disk file operations I/O                        1        0.01          0.01
  direct path read                               11        0.00          0.00
  asynch descriptor resize                        4        0.00          0.00
  SQL*Net message from client                   383       27.14         45.58
********************************************************************************
...
```

Figura 12.32: Acesso ao plano de execução no formato texto: https://github.com/eduardogoncalvesbr/livrotuning-casadocodigo/blob/master/cap12img27code.txt

E, com isso, obtivemos os arquivos de `trace` formatados com as informações referentes ao que foi executado nas sessões de exemplo.

Para finalizar, seguem dois `scripts` que auxiliam na montagem dos comandos, tanto para a geração do `trace`, como para a formatação dos arquivos, com base nas informações de uma determinada sessão. Você só precisa executar as instruções SQL a seguir e copiar os resultados, que nada mais são que os comandos que você pode utilizar para habilitar e desabilitar o `trace`, e também formatar os arquivos gerados.

Para o pacote `dbms_system`:

```
SQL> select sid
  2         ,serial#
  3         ,'exec dbms_system.set_sql_trace_in_session('||sid||',
'||serial#||',true);' as sql_start
  4         ,'exec dbms_system.set_sql_trace_in_session('||sid||',
'||serial#||',false);' as sql_end
  5  from   v$session
  6  where  sid = 14
```

```
  7  and    serial# = 133;

 SID   SERIAL# SQL_START
       SQL_END
---- --------- ---------------------------------------------------
------ ---------------------------------------------------------
  14       133 exec dbms_system.set_sql_trace_in_session(14,133,t
rue);  exec dbms_system.set_sql_trace_in_session(14,133,false);

SQL>
```

Para o pacote `dbms_monitor` :

```
SQL> select sid
  2        ,serial#
  3        ,'dbms_monitor.session_trace_enable(session_id => '||s
id||', serial_num => '||serial#||', waits => true, binds => true)
;' as sql_start
  4        ,'dbms_monitor.session_trace_disable(session_id => '||
sid||', serial_num => '||serial#||');' as sql_end
  5  from   v$session
  6  where  sid = 14
  7  and    serial# = 139;

       SID    SERIAL#
---------- ----------
SQL_START
-------------------------------------------------------------------
---------------
SQL_END
-------------------------------------------------------------------
---------------
        14        139
dbms_monitor.session_trace_enable(session_id => 14, serial_num =>
 139, waits => true, binds => true);
dbms_monitor.session_trace_disable(session_id => 14, serial_num =
> 139);

SQL>
```

Para o utilitário `tkprof` :

```
SQL> select 'tkprof '||c.value || `'\'` || d.instance || '_ora_'
||ltrim(to_char(a.spid,'fm99999')) || '.trc'||' '||
```

```
  2         replace(c.value || `'\'` || d.instance || '_ora_' ||l
trim(to_char(a.spid,'fm99999')) || '.trc','.trc','.prf')||' '||
  3         'table='||'corporativo.plan_table explain=ttun/ttun@x
e'||' '||
  4         'sys=no sort=exeela,prsela,fchela' tkprof
  5  from   v$process   a
  6        ,v$session   b
  7        ,v$diag_info c
  8        ,v$thread    d
  9  where  a.addr    = b.paddr
 10  and    c.name    = 'Diag Trace'
 11  --and   b.audsid = userenv('sessionid')
 12  and    b.sid     = 14
 13  and    b.serial# = 133;

tkprof
-----------------------------------------------------------------
---------------
tkprof C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_1740.
trc C:\ORACLEXE\APP\ORACLE\diag\rdbms\xe\xe\trace\xe_ora_1740.prf
 table=corporativo.plan_table explain=ttun/ttun@xe sys=no sort=ex
eela,prsela,fchela


SQL>
```

> **Nota:** O utilitário `tkprof` pode ser usado para a formatação dos arquivos `trace`, independente do pacote utilizado para a geração dos arquivos.

Através dos utilitários e comandos aqui mencionados, foi possível dar uma ideia de como obter informações sobre as instruções SQL executadas em um banco de dados e, com isso, ter elementos para gerar análises e montar soluções para resolver determinados problemas de performance.

Note que esta forma de análise é um pouco diferente das

demais apresentadas, pois se trata de um rastreio completo da sessão, levando em conta outros fatores, como concorrência, tráfego de rede e outras questões. Ele é bem interessante para quando precisamos analisar processos grandes nos quais há várias instruções SQL sendo executadas para uma única tarefa ou um conjunto de tarefas dependentes.

O `trace` de sessão coletará todos os dados destas instruções e os incluirá no arquivo de `trace`. Algumas vezes, pode ocorrer de você utilizar o `explain plan`, `autotrace` ou `dbms_xplan` para avaliar alguma instrução SQL, por exemplo, e não conseguir detectar problemas de performance em um primeiro momento, mas, ao gerar um `trace` na sessão do usuário, em determinadas condições de tráfego de informações, memória e outros fatores, você poderá ter panoramas diferentes. Portanto, esse recurso se torna fundamental no monitoramento e na análise da performance.

Capítulo 13

CONCLUSÃO

Atualmente, o fluxo de informação gerado entre os sistemas informatizados é muito maior em relação a épocas passadas. Portanto, a necessidade de buscar informações ou armazená-las com maior rapidez torna-se constante nesses sistemas, principalmente em médias e grandes corporações, nas quais o fluxo de informação é muito maior. Muitos desses sistemas precisam ter um tempo de resposta bastante alto.

É sabido que essas informações, a grande maioria, encontram-se armazenadas em bancos de dados e que a comunicação entre a interface do sistema e o banco de dados é realizada pela linguagem SQL. O otimizador do banco de dados, que é o responsável pela otimização das instruções SQL, deve executar essas instruções da forma mais adequada possível. Se isso não acontecer, poderá ter como consequência sistemas operando determinados processos de forma lenta, ou funcionar lentamente como um todo. Deve ficar claro que os sistemas dependem das informações para sobreviver; são criados para analisar, processar, emitir e armazenar essas informações e, para tanto, devem desempenhar um ótimo papel.

A busca pela performance adequada é o objetivo do Tuning de SQL. Por intermédio do uso de técnicas de Tuning, é possível fazer ajustes nas muitas instruções de manipulação de dados que são

inseridas em um banco de dados. Essas técnicas podem ser implementadas no sentido de melhorar todo um processo que possa estar prejudicando e inviabilizando o uso de alguma atividade do sistema. O Tuning de SQL deve ser pensado principalmente na fase de análise do projeto, evitando complicações futuras com o desenvolvimento e uso do sistema como foi visto. Com isso, é possível entender como o otimizador deste banco trata as instruções e assim auxiliá-lo para que ele escolha o melhor caminho para executá-las.

Através da abordagem das técnicas e ferramentas, demonstramos que diversos problemas relacionados à performance podem ser melhorados e que podemos aprender muito sobre o banco de dados que estamos utilizando se elas forem aplicadas no dia a dia. A criação de índices de maneira adequada, o uso desses índices pelo otimizador, a sintaxe das instruções, a forma propriamente dita da escrita de instruções SQL, o uso de sugestões (`hints`) para auxiliar o otimizador na construção do melhor caminho para execução das instruções, as ferramentas de análise e tratamento das instruções SQL, tudo isso é uma gama de conhecimento valiosa em tempos em que o uso da informação vem sendo tão difundido e sua alta disponibilidade, exigida cada vez mais.

Espero que tenha gostado do tema. Sugiro que não se apegue apenas aos conteúdos deste livro, pois aqui foram abordados dentro de um escopo mais específico e direcionado a quem desenvolve aplicações. Ainda há muito a se falar de Tuning em vários aspectos. Use este livro como um aprendizado inicial sobre o assunto, pois lhe dará uma boa base sobre as técnicas existentes e abrirá novas linhas para seguir em frente. Abraço!

Capítulo 14

REFERÊNCIAS BIBLIOGRÁFICAS

Adaptive Cursor Sharing and Adaptive Execution Plans. Disponível em: https://oracle.readthedocs.io/en/latest/plsql/bind/adaptive-cursor-sharing.html. Acesso em: 16 out. 2021.

Adaptive Cursor Sharing in Oracle Database 11g Release 1. Disponível em: https://oracle-base.com/articles/11g/adaptive-cursor-sharing-11gr1. Acesso em: 16 out. 2021.

Analisando o Plano de Execução para tunar instruções SQL. Disponível em: http://www.fabioprado.net/2011/03/analisando-o-plano-de-execucao-para.html. Acesso em: 16 out. 2021.

Analisando SQL traces em Bancos de Dados Oracle - Parte 1. Disponível em: https://www.fabioprado.net/2013/09/analisando-traces-em-bancos-de-dados.html. Acesso em: 16 out. 2021.

Analisando SQL traces em Bancos de Dados Oracle - Parte 2. Disponível em: https://www.fabioprado.net/2013/09/analisando-sql-traces-em-bancos-de.html. Acesso em: 16 out. 2021.

Analise do seu SGDB com o pacote DBMS_STATS. Disponível

em: http://oracle-startup.blogspot.com.br/2012/01/analise-do-seu-sgdb-com-o-pacote.html. Acesso em: 16 out. 2021.

Analyze and DBMS_STATS. Disponível em: https://asktom.oracle.com/pls/apex/f?p=100:11:::NO:RP:P11_QUESTION_ID:4347359891525. Acesso em: 16 out. 2021.

Clients - Parse, Execute, Fetch - Starttime. Disponível em: https://community.oracle.com/thread/555662?start=0&tstart=0. Acesso em: 16 out. 2021.

Coletando estatísticas para o otimizador de queries do Oracle. Disponível em: http://www.fabioprado.net/2012/04/coletando-estatisticas-para-o.html. Acesso em: 16 out. 2021.

Como identificar o SQL_ID de um processamento. Disponível em: http://oracle-sql-procedimentos.blogspot.com/2016/12/como-identificar-o-sqlid-de-um.html. Acesso em: 16 out. 2021.

Como o Oracle recupera os dados - Parte 1. Disponível em: http://www.linhadecodigo.com.br/artigo/768/como-o-oracle-recupera-os-dados-parte-1.aspx. Acesso em: 16 out. 2021.

Como o Oracle recupera os dados - Parte 2. Disponível em: http://www.linhadecodigo.com.br/artigo/770/como-o-oracle-recupera-os-dados-parte-2.aspx. Acesso em: 16 out. 2021.

Configurando SGA X PGA no Oracle Database. Disponível em: https://www.fabioprado.net/2016/06/configurando-sga-x-pga-no-oracle.html. Acesso em: 16 out. 2021.

Configurando SGA X PGA no Oracle Database. Disponível em: https://www.oracle.com/technetwork/pt/articles/database-

performance/configurando-sga-pga-database-3210653-ptb.html. Acesso em: 16 out. 2021.

Cost-Based Optimizer (CBO) And Database Statistics. Disponível em: https://oracle-base.com/articles/misc/cost-based-optimizer-and-database-statistics. Acesso em: 16 out. 2021.

Current Trace File Tips. Disponível em: http://www.dba-oracle.com/t_trace_file_current.htm. Acesso em: 16 out. 2021.

Data Warehousing Seminar Chapter 13 Indexing the Warehouse. Disponível em: http://slideplayer.com/slide/9325476/#. Acesso em: 16 out. 2021.

Database Performance Tuning Guide and Reference Release 2 (9.2). Disponível em: https://docs.oracle.com/cd/B10501_01/server.920/a96533.pdf. Acesso em: 16 out. 2021.

Database SQL Apress 2008 Troubleshooting Oracle Performance, Christian Antognini.

DBMS_XPLAN. Disponível em: https://oracle-base.com/articles/9i/dbms_xplan. Acesso em: 16 out. 2021.

DBMS_XPLAN and GATHER_PLAN_STATISTICS. Disponível em: http://ba6.us/?q=dbms_xplan-gather_plan_statistics. Acesso em: 16 out. 2021.

DBMS_XPLAN Format Parameters. Disponível em: https://hoopercharles.wordpress.com/2010/03/01/dbms_xplan-format-parameters/. Acesso em: 16 out. 2021.

DBMS_XPLAN versus Autotrace. Disponível em:

https://hourim.wordpress.com/2011/04/20/dbms_xplan-versus-autotrace/. Acesso em: 16 out. 2021.

DBMS_XPLAN: você sabe usar esse extraordinário pacote? Disponível em: http://dbtimewizard.com.br/blog/dbms_xplan-voce-sabe-usar-esse-extraordinario-pacote/. Acesso em: 16 out. 2021.

DBMS_XPLAN.Display_Cursor analysis [transfer]. Disponível em: http://www.programmersought.com/article/2343183261/. Acesso em: 16 out. 2021.

DBMS_XPLAN. Disponível em: https://jonathanlewis.wordpress.com/2008/03/06/dbms_xplan3/. Acesso em: 16 out. 2021.

Designing applications for performance and scalability. Disponível em: https://www.oracle.com/technetwork/database/performance/designing-applications-for-performa-131870.pdf. Acesso em: 16 out. 2021.

Dicas para otimização de consultas. Disponível em: http://www.ebah.com.br/content/ABAAAAYNsAC/dicas-sql#. Acesso em: 16 out. 2021.

E-Business Applications Performance Basic Skills Cookbook . Disponível em: http://ebstunning.blogspot.com/2016/02/e-business-applications-performance_99.html. Acesso em: 16 out. 2021.

Eficiência e segurança com SQL parametrizado. Disponível em: http://www.mhavila.com.br/topicos/bd/sqlbind.html. Acesso em:

16 out. 2021.

Enabling SQL Tracing and Using TKPROF. Disponível em: http://dbaahmed.blogspot.com/2016/03/enabling-sql-tracing-and-using-tkprof.html. Acesso em: 16 out. 2021.

Entendendo arquivo de trace. Disponível em: http://www.rauldba.com.br/entendendo-arquivo-de-tracefinal/. Acesso em: 16 out. 2021.

Entendendo o que se lê…. Disponível em: http://www.rauldba.com.br/entendendo-o-que-se-le/. Acesso em: 16 out. 2021.

Enterprise DBA Parte2: Workshop de Otimização e Performance.

Execution Plan: Como interpretar os valores estatísticos estimados. Disponível em: http://dbtimewizard.com.br/blog/execution-plan-como-interpretar-os-valores-estatisticos-estimados/. Acesso em: 16 out. 2021.

Execution Plans Part 11: Actuals. Disponível em: https://www.red-gate.com/simple-talk/sql/oracle/execution-plans-part-11-actuals/. Acesso em: 16 out. 2021.

Execution Plans Part 13: SQL Trace. Disponível em: https://www.red-gate.com/simple-talk/sql/oracle/execution-plans-part-13-sql-trace/. Acesso em: 16 out. 2021.

Execution Plans, Partitioning, and Parallelism in Oracle. Disponível em: https://www.centroid.com/execution-plans-partitioning-and-parallelism-oracle/. Acesso em: 16 out. 2021.

EXPLAIN PLAN. Disponível em: https://oraclepress.wordpress.com/2017/01/16/explain-plan/. Acesso em: 16 out. 2021.

Explain Plan Usage. Disponível em: https://oracle-base.com/articles/8i/explain-plan-usage. Acesso em: 16 out. 2021.

Explain Plan: Different formats/options with dbms_xplan. Disponível em: http://db.geeksinsight.com/2012/11/25/explain-plan-different-formatsoptions-with-dbms_xplan/. Acesso em: 16 out. 2021.

Explaining the Explain Plan Interpreting Execution Plans for SQL Statements . Disponível em: https://pt.slideshare.net/MariaColgan1/explain-the-explainplan. Acesso em: 16 out. 2021.

Formal SQL Tuning for Oracle Databases: Practical Efficiency - Efficient .... Disponível em: https://books.google.com.br/books?id=eoAXDQAAQBAJ&pg=PA48&lpg=PA48&dq=Cartesian+Join+Oracle+steps&source=bl&ots=k8VEYC57-E&sig=ACfU3U1ixJQqETCzc7ICTMoLtlyLUukcIw&hl=pt-BR&sa=X&ved=2ahUKEwjv-Yaf9OLpAhWtIbkGHZbIC1UQ6AEwE3oECA4QAQ#v=onepage&q=Cartesian%20Join%20Oracle%20steps&f=false. Acesso em: 16 out. 2021.

GATHER_PLAN_STATISTICS Hint Tips. Disponível em: http://www.dba-oracle.com/t_gather_plan_statistics.htm. Acesso em: 16 out. 2021.

Get the Execution Plan from/using V$SQL_PLAN. Disponível em: http://behindoracle.blogspot.com/2014/03/get-execution-

plan-fromusing-vsqlplan.html. Acesso em: 16 out. 2021.

Getting a Handle on Indexes. Disponível em: http://www.dba-oracle.com/t_callan_handle_on_indexes.htm. Acesso em: 16 out. 2021.

High number of misses in library cache during execute - where to dig for explanations? Disponível em: https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:9532001800346103738. Acesso em: 16 out. 2021.

High-Performance Oracle: Proven Methods for Achieving Optimum Performance .... Disponível em: https://books.google.com.br/books?id=JD4-pBV6Bf4C&pg=PA292&lpg=PA292&dq=HASH_VALUE+e+ADDRESS+plan_table&source=bl&ots=PLpwR4NOMZ&sig=ACfU3U3MANDHBNhwrQTwmeSqa83ofgxApg&hl=pt-BR&sa=X&ved=2ahUKEwigo4uSwqDoAhU0D7kGHfwbCjIQ6AEwAXoECAoQAQ#v=onepage&q=HASH_VALUE%20e%20ADDRESS%20plan_table&f=false. Acesso em: 16 out. 2021.

How do I capture a 10053 trace for a SQL statement called in a PL/SQL package? Disponível em: https://blogs.oracle.com/optimizer/how-do-i-capture-a-10053-trace-for-a-sql-statement-called-in-a-plsql-package. Acesso em: 16 out. 2021.

How do I display and read the execution plans for a SQL statement. Disponível em: https://blogs.oracle.com/optimizer/how-do-i-display-and-read-the-execution-plans-for-a-sql-statement. Acesso em: 16 out. 2021.

How to Create an Execution Plan. Disponível em: https://blogs.oracle.com/sql/how-to-create-an-execution-plan#tkprof. Acesso em: 16 out. 2021.

Interpreting Explain Plan. Disponível em: https://www.akadia.com/services/ora_interpreting_explain_plan.html. Acesso em: 16 out. 2021.

Library Cache Miss. Disponível em: https://jonathanlewis.wordpress.com/2007/07/19/library-cache-miss/. Acesso em: 16 out. 2021.

Mudando Parâmetros Ocultos do Oracle. Disponível em: http://www.oraclehome.com.br/2011/10/24/mudando-parametros-ocultos-do-oracle/. Acesso em: 16 out. 2021.

Nested Loops Join – the Smaller Table is the Driving Table, the Larger Table is the Driving Table. Disponível em: https://hoopercharles.wordpress.com/2011/03/21/nested-loops-join-the-smaller-table-is-the-driving-table-the-larger-table-is-the-driving-table/. Acesso em: 16 out. 2021.

Nested loops, Hash join and Sort Merge joins – difference? Disponível em: http://oracle-online-help.blogspot.com.br/2007/03/nested-loops-hash-join-and-sort-merge.html. Acesso em: 16 out. 2021.

NULL-Related Functions. Disponível em: https://oracle-base.com/articles/misc/null-related-functions#lnnvl. Acesso em: 16 out. 2021.

Oracle - Bitmap INDEX. Disponível em: http://kjk3071.tistory.com/entry/DB-Oracle-Bitmap-INDEX.

Acesso em: 16 out. 2021.

Oracle - Referência Para O Dba. Disponível em: https://books.google.com.br/books?id=oayqYCRhMQkC&pg=PA292&lpg=PA292&dq=sobre+pacote+oracle+DBMS_XPLAN&source=bl&ots=6OijNkowsC&sig=ss7m4eKCTylcMsdkj6bU1CtzWRY&hl=pt-BR&sa=X&ved=0ahUKEwj8mZ6Du7vaAhWLiZAKHVBjAJUQ6AEIVjAG#v=onepage&q=sobre%20pacote%20oracle%20DBMS_XPLAN&f=false. Acesso em: 16 out. 2021.

Oracle 10g de-supports rule-based SQL Optimizer. Disponível em: http://www.dba-oracle.com/oracle_tips_ault_rbo_10g.htm. Acesso em: 16 out. 2021.

Oracle 9i SQL Tuning Workshop v1. Disponível em: http://docslide.us/documents/oracle-9i-sql-tuning-workshop-v1.html. Acesso em: 16 out. 2021.

Oracle 9i SQL Tuning Workshop v2. Disponível em: http://docslide.us/documents/oracle-9i-sql-tuning-workshop-v2.html. Acesso em: 16 out. 2021.

Oracle ANALYZE INDEX. Disponível em: http://www.dba-oracle.com/t_oracle_analyze_index.htm. Acesso em: 16 out. 2021.

Oracle Bitmap Index Techniques. Disponível em: http://www.dba-oracle.com/oracle_tips_bitmapped_indexes.htm. Acesso em: 16 out. 2021.

Oracle Database - Trace File 10053 to understand the Optimizer (CBO). Disponível em: https://gerardnico.com/db/oracle/10053. Acesso em: 16 out. 2021.

Oracle Database 12c Performance Tuning Recipes: A Problem-Solution Approach. Disponível em: https://books.google.com.br/books?id=15oQAwAAQBAJ&pg=PA312&lpg=PA312&dq=Query+Block+Name+/+Object+Alias+oracle+dbms_xplan&source=bl&ots=4aIMw8mdv2&sig=ACfU3U2-K5J8ikGx0oj6L7khw_vF6hUL3A&hl=pt-BR&sa=X&ved=2ahUKEwjZ1P-kqbXoAhXkILkGHUrFDQw4ChDoATACegQIBxAB#v=onepage&q=Query%20Block%20Name%20%2F%20Object%20Alias%20oracle%20dbms_xplan&f=false. Acesso em: 16 out. 2021.

Oracle Database Online Documentation 11g Release 1 (11.1). Disponível em: https://docs.oracle.com/cd/B28359_01/appdev.111/b28419/toc.htm. Acesso em: 16 out. 2021.

Oracle Database Online Documentation 11g, Release 2 (11.2). Disponível em: https://docs.oracle.com/cd/E11882_01/appdev.112/e40758/toc.htm. Acesso em: 16 out. 2021.

Oracle Database Online Documentation 12c Release 1 (12.1). Disponível em: https://docs.oracle.com/database/121/TGSQL/toc.htm. Acesso em: 16 out. 2021.

Oracle Database Online Documentation, 10g Release 2 (10.2). Disponível em: https://docs.oracle.com/cd/B19306_01/server.102/b14231/general.htm#i1006229. Acesso em: 16 out. 2021.

Oracle dbms_support. Disponível em: http://www.dba-oracle.com/t_dbms_support.htm. Acesso em: 16 out. 2021.

Oracle Explain Plans: Driving Tables and Table Joins. Disponível em: http://logicalread.solarwinds.com/oracle-explain-plans-driving-tables-and-table-joins-h01/#.WEbSrHfOpn4. Acesso em: 16 out. 2021.

Oracle hash_area_size. Disponível em: http://www.dba-oracle.com/tips_oracle_hash_area_size.htm. Acesso em: 16 out. 2021.

Oracle Index skew. Disponível em: http://www.remote-dba.net/oracle_10g_tuning/t_oracle_index_skew.htm. Acesso em: 16 out. 2021.

Oracle Otimizando Comandos SQL. Disponível em: http://nervinformatica.com.br/Downloads/Materiais/WOTSQL.pd . Acesso em: 16 out. 2021.

Oracle Performance Tuning for 10gR2. Disponível em: https://books.google.com.br/books?id=wgBMWCvTEHQC&pg=PA397&lpg=PA397&dq=v$sql_plan+oracle+exemplos&source=bl&ots=pFyzPu2m-e&sig=ACfU3U3MKxi3uTjJB4ZNeYpmr7qXKFA-Aw&hl=pt-BR&sa=X&ved=2ahUKEwjJj_6d0qHoAhX2HbkGHSyRAOIQ6AEwBXoECAoQAQ#v=onepage&q=v%24sql_plan%20oracle%20exemplos&f=false. Acesso em: 16 out. 2021.

Oracle SQL execution steps. Disponível em: http://www.dba-oracle.com/t_sql_exeution_steps.htm. Acesso em: 16 out. 2021.

Oracle-btree and bitmap index. Disponível em:

https://www.programmersought.com/article/8668116234/. Acesso em: 16 out. 2021.

Oracle® Database Administrator's Guide 11g Release 2 (11.2). Disponível em: https://docs.oracle.com/cd/E11882_01/server.112/e25494/toc.htm. Acesso em: 16 out. 2021.

Oracle® Database Concepts 11g Release 2 (11.2). Disponível em: https://docs.oracle.com/cd/E11882_01/server.112/e40540/toc.htm. Acesso em: 16 out. 2021.

Oracle® Database Performance Tuning Guide 10g Release 1 (10.1). Disponível em: https://docs.oracle.com/cd/B13789_01/server.101/b10752/index.htı . Acesso em: 16 out. 2021.

Oracle® Database SQL Language Reference 11g Release 2 (11.2). Disponível em: https://docs.oracle.com/cd/E11882_01/server.112/e41084/toc.htm. Acesso em: 16 out. 2021.

Oracle7 Tuning, release 7.3.3. Disponível em: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/A48506/explan.htm. Acesso em: 16 out. 2021.

Oracle8 DBA Handbook, Kevin Loney, 1997.

Oracle9i Database Performance Tuning Guide and Reference Release 2 (9.2) October 2002. Disponível em: https://docs.oracle.com/cd/B10501_01/server.920/a96533/index.htı . Acesso em: 16 out. 2021.

Oracle9i Data Warehousing Guide Release 2 (9.2). Disponível em: https://docs.oracle.com/cd/A97630_01/server.920/a96520/indexes.htm. Acesso em: 16 out. 2021.

Oracle9i Database Performance Tuning Guide and Reference Release 2 (9.2). Disponível em: https://docs.oracle.com/cd/A97630_01/server.920/a96533/index.htı . Acesso em: 16 out. 2021.

Otimização de ambiente Oracle – SQL Performance and Tuning de SQL's. Disponível em: https://www.profissionaloracle.com.br/swillians/2013/09/08/otimizacao-de-ambiente-oracle-sql-performance-and-tuning-de-sql-s/. Acesso em: 16 out. 2021.

Otimizando a performance de bancos de dados relacionais. Roberto Carlos Mayer. 2001. Axcel Books do Brasil.

PL/SQL Packages and Types Reference 10g Release 1 (10.1). Disponível em: https://docs.oracle.com/cd/B14117_01/appdev.101/b10802/d_xplan.htm. Acesso em: 16 out. 2021.

Por que utilizar "Explain Plan" ou "AUTOTRACE" no processo de "Tuning" pode ser uma má idéia? Disponível em: http://dbtimewizard.com.br/blog/por-que-utilizar-explain-plan-ou-autotrace-no-processo-de-tuning-pode-ser-uma-ma-ideia-2/. Acesso em: 16 out. 2021.

Pro Oracle SQL. Disponível em: https://books.google.com.br/books?id=fRy6Q7DbdvAC&pg=PA166&lpg=PA166&dq=HASH_VALU

E+e+ADDRESS+plan_table&source=bl&ots=8aIZBO7_pW&sig=ACfU3U0aYgX3n7awSZy-bqoWEXNa5ViFiw&hl=pt-BR&sa=X&ved=2ahUKEwigo4uSwqDoAhU0D7kGHfwbCjIQ6AEwA3oECAgQAQ#v=onepage&q=HASH_VALUE%20e%20ADDRESS%20plan_table&f=false. Acesso em: 16 out. 2021.

QL trace, 10046, trcsess and tkprof in Oracle. Disponível em: https://oracle-base.com/articles/misc/sql-trace-10046-trcsess-and-tkprof. Acesso em: 16 out. 2021.

Quanto o EXPLAIN PLAN está errado – Bind Peeking. Disponível em: http://nervinformatica.com.br/blog/index.php/2015/07/01/quanto-o-explain-plan-esta-errado-bind-peeking/. Acesso em: 16 out. 2021.

Quer mais performance? Faça upgrade para o 19c. Disponível em: http://dbtimewizard.com.br/blog/quer-mais-performance-faca-upgrade-para-o-19c/. Acesso em: 16 out. 2021.

RECORD uma caracteristica do TKPROF. Disponível em: http://www.oraclehome.com.br/2013/01/11/record-uma-caracteristica-do-tkprof/. Acesso em: 16 out. 2021.

Recriando o repositório do AWR manualmente. Disponível em: http://olivertconsultoria.blogspot.com/2013/11/recriando-o-repositorio-do-awr.html. Acesso em: 16 out. 2021.

Running various oracle traces inclding 10053 and 10046 trace file. Disponível em: http://mrafeyblog.blogspot.com/2014/05/running-various-oracle-traces-inclding.html. Acesso em: 16 out. 2021.

Secrets of the Oracle Database. Disponível em: https://books.google.com.br/books?id=ZxEL-FO4u7wC&pg=PA279&lpg=PA279&dq=find+%22Plan+hash+value%22+tkprof+oracle&source=bl&ots=6fw3wcgB9b&sig=ACfU3U0M570LuuHSehrIRXQnc4eExvl7hw&hl=pt-BR&sa=X&ved=2ahUKEwiNpNTYu8HmAhW7H7kGHWUfC1k4ChDoATAGegQIChAB#v=onepage&q=find%20%22Plan%20hash%20value%22%20tkprof%20oracle&f=false. Acesso em: 16 out. 2021.

Sequence of steps in Oracle SQL explain plan. Disponível em: http://www.dba-oracle.com/t_order_sequence_sql_execution_explain_plans_steps.htm. Acesso em: 16 out. 2021.

SQL Tuning with Adaptive Cursor Sharing and Cardinality Feedback. Disponível em: https://skillbuilders.com/free-oracle-dba-tutorials/oracle-adaptive-cursor-sharing-and-cardinality-feedback/. Acesso em: 16 out. 2021.

SQL Tuning: Generating Optimal Execution Plans. Disponível em: https://books.google.com.br/books?id=dOYPtfVqS_MC&pg=PT44&lpg=PT44&dq=%22indexed+access+versus+a+full+table+scan%22&source=bl&ots=lXEiEu6OCp&sig=ACfU3U0Ismnq94Amn0ztWgMlleT-aoiENw&hl=pt-BR&sa=X&ved=2ahUKEwiguuCWu7HwAhVzrpUCHWWMA74Q6AEwAXoECAEQAw#v=onepage&q=%22indexed%20access%20versus%20a%20full%20table%20scan%22&f=false. Acesso em: 16 out. 2021.

The Anti-Join. Disponível em: http://www.remote-dba.net/t_op_sql_anti_join.htm. Acesso em: 16 out. 2021.

Three ways to get a plan with dbms_xplan. Disponível em: http://www.bobbydurrettdba.com/2012/07/11/three-ways-to-get-a-plan-with-dbms_xplan/. Acesso em: 16 out. 2021.

Trace Files -- 4 : Identifying a Trace File. Disponível em: https://hemantoracledba.blogspot.com/2015/10/trace-files-4-identifying-trace-file.html. Acesso em: 16 out. 2021.

Trace no Oracle 2. Disponível em: https://www.devmedia.com.br/trace-no-oracle-2/4315. Acesso em: 16 out. 2021.

Troubleshooting Oracle Performance. Disponível em: https://books.google.com.br/books?id=b3DIkYO2gBQC&pg=PA209&lpg=PA209&dq=Rows+Bytes+TempSpc+cost+time+oracle&source=bl&ots=yWXwFFdcuC&sig=ACfU3U1jfnAV4Wqk9tSy5eDOu-MWWNdqCw&hl=pt-BR&sa=X&ved=2ahUKEwjXvau53_voAhVRLLkGHQUSANwQ6AEwBnoECAoQAQ#v=onepage&q=Rows%20Bytes%20TempSpc%20cost%20time%20oracle&f=false. Acesso em: 16 out. 2021.

Troubleshooting Oracle Performance by Christian Antognini Publisher: Apress Publishing date: June 2008. Disponível em: https://www.antognini.ch/papers/InterpretingExecutionPlans_20091017.pdf. Acesso em: 16 out. 2021.

Troubleshooting Oracle Performance, 2nd Edition, Christian Antognini.

Tuning Básico de PGA. Disponível em: https://www.oraclehome.com.br/2012/02/16/tuning-basico-de-pga//. Acesso em: 16 out. 2021.

TUNING BY CARDINALITY FEEDBACK METHOD AND EXAMPLES. Disponível em: http://www.centrexcc.com/Tuning%20by%20Cardinality%20Feedback.pdf. Acesso em: 16 out. 2021.

Tuning, o que é isso? Disponível em: https://www.tiespecialistas.com.br/tuning-o-que-e-isso/. Acesso em: 16 out. 2021.

Uma introdução ao AWR. Disponível em: https://oraclepress.wordpress.com/2019/07/31/uma-introducao-ao-awr/. Acesso em: 16 out. 2021.

Using DBMS_XPLAN.DISPLAY_CURSOR to examine Execution Plans. Disponível em: https://sqlmaria.com/2017/08/08/using-dbms_xplan-display_cursor-to-examine-execution-plans/. Acesso em: 16 out. 2021.

Utilizando DBMS_XPLAN para obter o Plano de Execução de uma instrução SQL. Disponível em: https://fabricadedados.wordpress.com/2013/11/07/utilizando-dbms_xplan-para-obter-o-plano-de-execucao-de-uma-instrucao-sql/. Acesso em: 16 out. 2021.