Módulo 9

Banco de Dados

Lição 8

Processando Consultas

Versão 1.0 - Fev/2009

Autor
Ma. Rowena C. Solamo

Equipe
Rommel Feria
Rick Hillegas
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/60/relnotes.htm

Java™ DB System Requirements

Java™ DB is supported on the Solaris, Linux and Windows operating systems and Sun Java 1.4 or later.

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alberto Ivo da Costa Vieira
Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
Angelo de Oliveira
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Carlos Hilner Ferreira Costa
Daniel Noto Paiva
Daniel Wildt
Denis Mitsuo Nakasaki
Fábio Antonio Ferreira
Givailson de Souza Neves
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Vianney Barrozo Costa
José Francisco Baronio da Costa
Kleberth Bezerra Galvão dos Santos
Luiz Fernandes de Oliveira Junior
Maria Carolina Ferreira da Silva
Maricy Caregnato
Mauricio da Silva Marinho
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw
Seire Pareja
Sergio Terzella
Thiago Magela Rodrigues Dias

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Nesta lição, iremos concentrar no tópico de processamento de consulta. A primeira seção fornece uma visão geral de processamento de consulta e suas quatro fases centrais, são elas, decomposição da consulta, otimização da consulta, geração de código e tempo de execução. A segunda seção discute a decomposição da consulta que inclui checagem de sintaxe e semântica da consulta, e transforma linguagem de consulta de alto nível (SQL) para linguagem de baixo nível (implementando expressões algébricas de relacionamento). A Terceira seção discute otimização de consulta. Duas principais são abordadas, especificamente, abordagem heurística e abordagem de estimativa de custo, finalmente, será discutida a técnica *Pipelining* que pode ser usada para melhorar ainda mais o processamento de consultas.

Ao final desta lição, o estudante será capaz de:

- Como realizar consultas
- Decomposição e otimização de consultas
- Técnica *Pipelining*

2. Introdução

O processamento da consulta envolve as atividades que recuperam dados a partir de um banco de dados. Um dos seus objetivos é transformar uma consulta escrita em linguagem de alto-nível, tipicamente SQL, em uma estratégia correta e eficiente executada em linguagem de baixo-nível, e para executar a estratégia para recuperar os dados solicitados.

Como um exemplo, suponha que nos precisamos encontrar, "Todos os itens vendidos em Alabama". A consulta é mostrada no código abaixo:

```
SELECT * FROM inventory, store
WHERE inventory.store_no = store.number
AND (store.address LIKE "%Alabama%");
```

Duas tabelas são usadas nesta consulta, são elas, inventory e store, suponha que existam 50 registros encontrados na tabela store, cada loja vende em media 1000 itens. A tabela do inventário poderá ter mais de 50.000 registros.

Para fazer uma simples ilustração, suponha que não existam índices ou chaves de classificação, também suponha que algum resultado intermediário esta armazenado no disco. Todas as vezes que nos acessarmos o dado, nos encontraremos um registro a cada vez. (Embora na realidade, acesso a disco sejam baseados em blocos que normalmente contem mais que uma linha.)

A álgebra relacional equivalente de consultas para esta Sentença SQL esta como abaixo, e os acessos a disco são calculados da seguinte maneira:

1ª Forma

```
σ(store.address like "%Alabama%")∧(inventory.store_no=store.number)(Inventory X Store)
```

Esta consulta calcula o produto cartesiano de inventário e armazenar as tabelas. Isso exigiria (50,000+50) acesso ao disco, e isso cria um relacionamento com (50.000 * 50) linhas. Então essa relação para é lida novamente para testar o predicado da seleção. (50.000 * 50) acessos ao disco são exigidos. O total do custo é calculado do seguinte modo:

(50,000+50) + (50,000*50) = 2,550,050 acessos ao disco

2ª Forma

```
σ(store.address like "%Alabama%")(Inventory  Inventory.store_no=store.number Store)
```

Esta consulta calcula primeiro a junção das tabelas inventory e store. Isso pode exigir (50.000 + 50) acesso ao disco. Isso cria um relacionamento com (50* 1000) pois sabemos que a loja vende em media 1000 itens. Próximo, isso poderia exigir (50 * 1000) acesso ao disco para fazer a seleção predicado. O total do custo é calculado do seguinte modo:

(50.000 + 50) + (50 * 1000) = 100.050 acessos ao disco

3ª Forma

```
Inventory  inventory.store_no=store.number σstore.address like "%Alabama%"
```

Está consulta calcula primeiro a seleção predicado na tabela store recuperando os registros em Alabama. Isso pode exigir 50 acessos ao disco. Isso cria um relacionamento com 1 linha (Assumindo que somente um registro pode ser encontrado no estado). Então, o resultado ira se juntado com a tabela inventory, requerendo 1000+ 1 acesso ao disco. O total é calculado como mostrado abaixo:

50 + 1000 + 1 = 1.051 acesso ao disco

Obviamente, a terceira consulta fornece a melhor opção e exigiria menos acessos ao disco de forma a obter o registro solicitado.

Processamento de consulta pode ser dividido em quatro fases principais: decomposição da consulta, otimização da consulta, geração de código, execução. Eles são mostrados na figura 2.

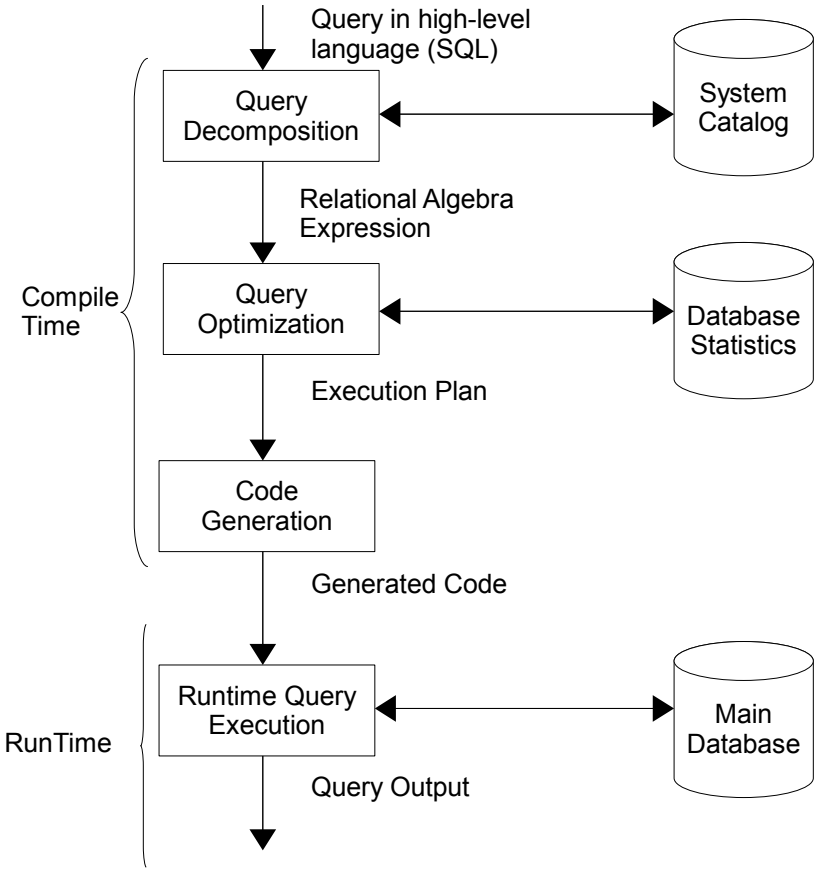

Figura 1: Fases da Consulta

Quatro fases centrais do processamento de consulta:

1. **Decomposição de Consulta**. Esta é a fase em que uma consulta de alto nível é convertido ou transformado em uma consulta de álgebra relacional . É também nesta fase em que é feita uma verificação para assegurar que a consulta é sinteticamente e semanticamente correta. Esta fase utiliza o sistema de catálogo. O sistema de catálogo contém informações sobre os metadados do banco de dados e objetos definidos no dicionário que o banco de dados, tais como tabelas, índices, views e etc. Informações Metadata sobre instâncias de dados são necessárias para o bom funcionamento de qualquer sistema de banco de dados. A sistema de banco de dados usa esta informação quando está atendendo uma solicitação do usuário, quer sob a forma de DML sentenças ou chamadas aos utilitários do banco de dados.
2. **Otimização de consulta**. Esta é a fase que envolve a escolha de uma execução eficiente estratégia de transformação da consulta. Ele utiliza banco de dados estatísticos para ajudar na avaliação sobre as diferentes opções disponíveis na execução de uma consulta. Abrange informações, tais como as relações, atributos, e índices. Também pode incluir cardinalidade das relações, do número de valores distintos para cada atributos, o número de níveis em uma indexação multi-nível etc Todas essas informações são armazenadas como parte do sistema de catálogo e atualizado periodicamente pelo DBMS.
3. **Geração de código**. Esta fase que transforma a estratégia de execução em operações de baixo-nível que podem ser executadas nesta fase.
4. **Consultas em tempo de execução**. Esta fase executa as operações de baixo-nível para recuperar dados do banco de dados. Este usa o banco central de dados onde os dados atuais estão armazenados e recuperados.

Iremos nos concentrar nas fases de decomposição de consulta e otimização de consulta.

3. Decomposição da Consulta

Esta é a primeira fase quando do processamento da consulta. Destina-se a transformar a linguagem de alto-nível, tais como SQL, em uma consulta de álgebra relacional, e também a verificação de erros de sintaxe e semântica. Os estágios normalmente utilizados para a decomposição de consulta são: analise normalização, analise de semântica, simplificação e reconstrução de consulta.

3.1. Análise

Nesta fase, a consulta é analisada por erros de sintaxe. Além disso, verifica que a relação e os atributos especificados na consulta são definidas no sistema de catálogo. Ele também verifica que qualquer operação aplicada ao dicionário objetos são o tipo de objeto adequado. Como um exemplo, considere a consulta mostrada no código a seguir:

```
SELECT itemCode FROM inventory WHERE store_no = '30';
```

Está consulta será rejeitada pelas seguintes razões:

- Na Cláusula SELECT, a coluna itemCode não é um objeto dicionário valido para a tabela inventory deste que esta não é uma coluna da tabela. Este deveria ser intem_code.
- Na cláusula WHERE, a comparação não é compatível desde que store_no é um dado do tipo numérico.

A saída deste estágio pode ser uma arvore de álgebra relacional. Isto é um arvore de consulta que representa o alto-nível consulta em uma representação interna que é mais adequado para processamento. Este é construído como as seguir.

- Um folha do *node* é criada para cada relacionamento de base na consulta.
- Nenhuma folha de *node* é criada para cada relacionamento intermediário produzido por uma operação de álgebra relacional.
- A raiz da arvore representa o resultado de uma consulta.
- A seqüencia de operações é direcionada para o deixar a raiz

Na Figura 2 vemos um exemplo de uma arvore de álgebra relacional para a seguinte consulta:

```
SELECT * FROM inventory, store
WHERE inventory.store_no = store.number
AND (store.address LIKE "%Alabama%");
```

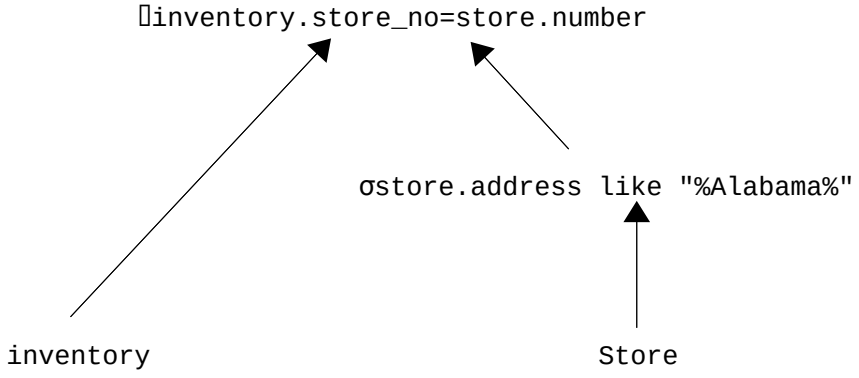

Figura 2: Árvore relacional Algébrica

3.2. Normalização

Nesta fase, a consulta é convertida em uma forma normalizada que pode ser facilmente

manipulada. O predicado (normalmente, uma cláusula WHERE e uma sentença SELECT) que pode ser complexa, pode ser convertida em uma ou duas formas pela aplicação das seguintes regras de transformação.

- **Forma Normal Conjuntiva**. É uma seqüência de conjuntos que esta conectada com o operador ^(AND). Cada conjunto contém um ou mais termos conectado pelo operador v(OR). Uma seleção conjuntiva contém somente aquelas linhas que satisfazem toda conjunção; i.e., Todas condições conectados pelo operador ^ é verdade.
- **Forma Normal Disjuntiva**. É uma seqüência disjuntiva que esta conectada pelo operador v(OR). Cada disjunção contem um ou mais termos conectados pelo operador ^(AND). A seleção disjuntiva contem aquelas linhas formadas pela união de todas as linhas que satisfaça a disjunção.

Considere a consulta mostrada a seguir:

```
SELECT employee.number, lastname, firstname
FROM employee, store
WHERE employee.store = store.number
AND store.number = 10
OR store.number = 30
```

Esta consulta retorna todos os trabalhadores empregados, seja na loja do Alabama (Loja Número 10) ou na loja de Dakota do Norte (Loja Número 30).

Possui a seguinte **Forma Normal Conjuntiva** da cláusula WHERE:

```
(employee.store = store.number) ∧ (store.number = 10 ∨ store.number = 30)
```

E a seguinte **Forma Normal Disjuntiva**:

```
((employee.store = store.number) ∧ (store.number = 10)) ∨
((employee.store = store.number) ∧ (store.number = 30))
```

A Forma Normal Disjuntiva normalmente deixa para replicação predicado. Observe no código que o predicado `employee.store = store.number` é repetido.

3.3. Análise Semântica

É necessária para rejeitar normalizada da consulta que estão incorreta mente formuladas ou são contraditórias. A consulta que não está propriamente formulada pode ter componentes que não contribuirão para a geração do resultado. Normalmente, isso acontece quando há componentes perdidos na consulta. Uma consulta que é contraditória pode ter predicados que não satisfaçam nenhuma linha na tabela.

Para um sub-ajuste das consultas que não têm nenhuma disjunção e negação, as seguintes checagem podem ser realizadas para determinar sua correção

Construindo gráfico de conexão relacional como foi formulado por Wong e Youssefi em 1976. Se o gráfico construído não esta conectado, nos dizemos que a consulta esta corretamente formulada. Para construir o gráfico, use os passos a seguir:

1. Criar um *node* para cada tabela participante da consulta.
2. Criar um *node* para representar o resultado da consulta.
3. Criar uma aresta para cada junção.
4. Criar uma aresta para cada código de operação de projeto.

Exemplo, considere a consulta mostrada no código a seguir:

```
SELECT item.description FROM store, inventory, item
WHERE store.number = inventory.store_no
AND (inventory.store_no = 10 or inventory.store_no = 30);
```

Esta consulta recupera os itens vendido para armazenar números 10 e 30. O Gráfico de conexão

relacional de consulta esta mostrada na figura a seguir. Note que o gráfico não está conectado. Neste caso, esquecemos de juntar *inventory* e *item*. portanto, a consulta está incorretamente formulada.

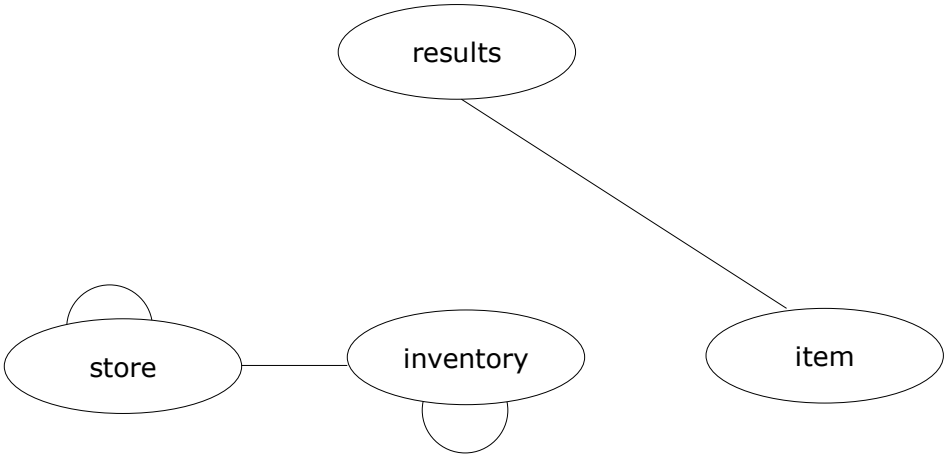

Figura 3: Gráfico de Conexão Relacional

Construindo atributo normalizado de conexão gráficas como foi proposto por Rosendranz e Hunt em 1980. Se o gráfico construído for um circulo para que o valor da soma seja negativo, A consulta é contraditória. Para construir o gráfico, use os seguintes passos:

1. Criar um *node* para cada referência para um atributo e para cada constante 0.
2. Criar uma aresta entre *nodes* que representam uma junção.
3. Criar uma aresta entre um *node* atributo e uma constante de 0 que representa a operação de seleção.
4. O peso das margens `a->b` terá o valor `c`, se isso representar a condição de desigualdade.
5. O peso das margens `0->a` terá o valor `-c`, se isso representar uma condição de desigualdade.

Como um exemplo, considere o código abaixo:

```
SELECT store.name, item.description FROM store, inventory, item
WHERE inventory.quantity > 500
AND store.number = inventory.store_no
AND inventory.quantity < 200
AND inventory.item_code = item.code
```

O Gráfico de Conexão do Atributo Normalizado da consulta é mostrado na figura a seguir:

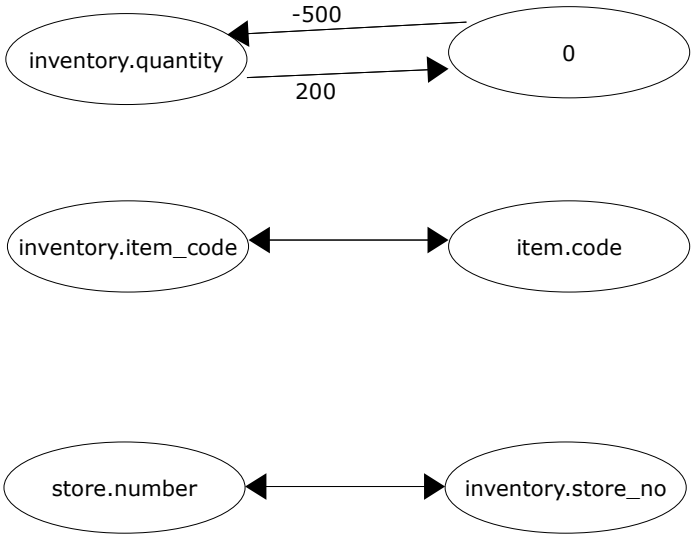

Figura 4: Gráfico de Conexão do Atributo normalizado

Possui um ciclo (inventory.quantity com constantes 0) e uma avaliação negativa tem uma soma (-500 +200 = -300). Isto indica que a consulta é contraditória. Claramente, um item não pode ter um valor para um (quantidade > 500) e (valor < 200), ao mesmo tempo.

3.4. Simplificação

O propósito da fase de simplificação é:

- Detectar qualificações de redundância
- Remover sub-expressões comuns.
- Transformar a consulta em uma semanticamente equivalente, mas de forma mais fácil e eficiente

Neste estagio, restrições de acesso, definições de visualização, e restrições de integridade são considerados. Se o usuário não tem o nível apropriado de privilegio para todos os componentes da consulta, a consulta é rejeitada. Uma inicial e simples otimização é aplicada usando o bom conhecimento das regras de álgebra booleana com estas:

```
p ∧ (p) ≡ p
p ∨ (p) ≡ p
p ∧ false = false
p ∨ false ≡ p
p ∧ true = p
p ∨ true ≡ true
p ∧ (~p) = false
p ∨ (~p) ≡ true
p ∧ (p ∨ q) = p
p ∨ (p ∧ q) ≡ p
```

3.5. Restruturação da Consulta

A consulta é reestruturada para fornecer de forma uma implementação mais eficiente. Reestruturação será discutida em detalhes na próximo seção.

4. Otimização da Consulta: Aproximação Heurística

A Aproximação Heurística otimiza as consultas usando a regra de transformação para converter uma expressão de algébrica relacional em uma expressão equivalente e conhecida por ser mais eficiente. Como mostrado para o código abaixo:

```
SELECT * FROM inventory, store
WHERE inventory.store_no = store.number
AND (store.address LIKE "%Alabama%");
```

É melhor executar a operação de seleção primeiro antes de uma operação JOIN. Nesta seção, veremos algumas regras de transformação que são válidas e apresentam um conjunto de heurística que é conhecido por produzir boas estratégias de execução; embora, não necessariamente podem ser ótimas.

4.1. Regras de Transformação para Operações Algébricas Relacionais

Otimizadores de consulta podem aplicar regras de transformação que permite converter uma expressão de álgebra de relacional em uma expressão de álgebra de relacional equivalente que é conhecida por ser mais eficiente. As regras serão aplicadas à árvore de álgebra relacional gerada durante a decomposição de consulta. As regras usam as seguintes estruturas:

1. Três relações descritas como R, S e T;
2. R possui atributos A = {A1, A2, ... , An};
3. S possui atributos B = {B1, B2, ... , Bn};
4. p, q e r são predicatos; e
5. L, L1, L2, M, M1, M2 e N representam um conjunto de atributos.

As regras de transformação são:

1.Cascata de Seleção

A operação conjuntiva de seleção pode ser feita em cascata com as operações de seleção individuais (e vice-versa).

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

Esta transformação é conhecida como Cascata de Seleção. Por exemplo:

$$\sigma_{\text{inventory.store\_no} = \text{store.number} \wedge \text{item\_code} = 1001}(\text{Inventory}) = \sigma_{\text{inventory.store\_no=store.number}}(\sigma_{\text{item\_code} = 1001}(\text{Inventory}))$$

2. Seleção Comutativa

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

Por exemplo:

$$\sigma_{\text{inventory.store\_no=store.number}}(\sigma_{\text{item\_code} = 1001}(\text{Inventory})) = \sigma_{\text{item\_code} = 1001}(\sigma_{\text{inventory.store\_no=store.number}}(\text{Inventory}))$$

3. Sequência de Operações de Projeção

Somente a última na sequência é requerida.

$$\Pi_L \Pi_M \ldots \Pi_N(R) = \Pi_L(R)$$

Por exemplo:

$$\Pi_{\text{number}} \Pi_{\text{name}}(\text{Store}) = \Pi_{\text{number}}(\text{Store})$$

4. Comutabilidade de seleção e projeção.

Se o predicado **p** envolve somente os atributos na lista de projeção, então a seleção são as

operações de projeção e comutação.

```
∏Ai, ..., Am (σ p (R) = σp( ∏Ai, ... Am(R)) where p ∈ {A1, A2, ... , Am}
```

Por exemplo:

```
∏number, name (σ number=30 (R) = σnumber=30( ∏number, name(R))
```

5. Comutabilidade de *theta-join* (e Produto Cartesiano). Como o *equi-join* e união natural é casos especiais do *theta-join*, esta regra se aplica a ambos.

```
R □ p S = S □ p R
R X S = S X R
```

Por exemplo:

```
Employee □ employee.store=store.number Store = Store □
employee.store=storem.number Employee
```

6. Comutatividade de seleção e *theta-join* (e Produto Cartesiano). Se o predicado da seleção, **p**, usa somente atributos de uma das relações que são unidas, então a seleção e a união (ou Produto Cartesiano) das operações de comutação.

```
σ p (R □ r S) = (σ p (R)) □ r S
σ p (R X S) = (σ p (R)) X S
quando p ∈ {A1, A2, ... , Am}
```

Se o predicado de seleção é um predicado conjuntivo que tem a forma (p q), onde **p** usa os atributos de **R**, e **q** usa os atributos de **S**, então a seleção é o *theta-join* das operações de comutação.

```
σ p∧q (R □ r S) = (σ p (R)) □ r (σq (S))
σ p∧q (R X S) = (σ p (R)) X (σq (S))
```

Por exemplo:

```
σ hiredate>'01-01-2000'∧store.number=30
  (Employee □ employee.store=store.number Store) =
  (σ hiredate>'01-01-2000' (Employee)) □
  employee.store=store.number σ store.number=30 (S)
```

7. Commutatividade de projeção e *theta-join* (ou Produto Cartesiano). Se a lista de projeção é `L = L1 ∪ L2` onde **L1** usa atributos de **R**, e **L2** usa atributos de **S**, e, a condição *join* contém atributo em **L**, a projeção e o *theta-join* das operações de comutação.

```
∏ L1 ∪ L2( R □ r S) =∏ L1(R) □ r ∏ L2(S)
```

Se a condição *join* contém atributos adicionais não achados em **L**, i.e., `M = M1 U M2` onde **M1** são atributos achou em **R** e **M2** é atributos achados em **S**, então a operação de projeto final é:

```
∏ L1 ∪ L2( R □ r S) =∏ L1 ∪ L2((∏ L1∪M1(R)) □ r (∏ L2□M2(S)))
```

Exemplo da primeira regra:

```
∏ employee.lastname, store.number(
  Employee □ employee.store = store.number Store) =
  ∏ employee.lastname(Employee) □ employee.store=store.number ∏
  store.number(Store)
```

Exemplo da segunda regra:

```
∏ employee.lastname, store.name( Employee □ employee.store =
  store.number Store) = ∏ employee.lastname, store.name(
  (∏ employee.lastname, employee.store(Employee)) □
  employee.store=store.number (∏ store.name, store.number(S)))
```

8. Comutatividade de união e interseção (sem diferença fixa).

```
R ∪ S = S ∪ R
R ∩ S = S ∩ R
```

9. Commutatividade de seleção e conjunto de operações (união, intersecção e diferença fixa).

```
σp(R ∪ S ))) = σp(R) ∪ σp(S)
σp(R ∩ S ))) = σp(R) ∩ σp(S)
σp(R - S ))) = σp(R) - σp(S)
```

10. Commutatividade de projeção e união.

```
∏L(R ∪ S ) = ∏L(R) ∪ ∏L(S)
```

11. Associatividade de *theta-join* (e Produto Cartesiano).

```
(R □ S) □ T = R □ (S □ T)
(R X S) X T = R X (S X T)
```

Quando a condição *join*, **q**, usa somente atributos da relação **S** e **T**, a união é dita **associativa** da seguinte maneira:

```
(R □p S) □q□r T = Rr∧p∧ □ (S □q T)
```

Por exemplo:

```
(Store □store.number = inventory.store_no Inventory) □
  inventory.item_code = item.code∧store.number = 10 Item =
  Storestore.number=inventory.store_no ∧ store.number=10 □
  (Inventory □ inventory.item_code=item.code Item)
```

12. Associatividade de união e intersecção (sem diferenças fixas).

```
(R ∪ S) ∪ T = R ∪ (S ∪ T)
(R ∩ S) ∩ T = R ∩ (S ∩ T)
```

4.2. Estrategias de processamento heuristico

Muitos sistemas de bancos de dados relacionais usam heuristica para determinar as estratégias que processamento de consulta. Esta seção fornece algumas boas heurísticas que possam ser aplicadas durante o processamento de consulta.

1. Realizar operações de seleção o mais cedo possível. A seleção reduz a cardinalidade da relação. Reduz o processamento subsequente das relações. Seria sábio usar a 1ª regra para cascatear o processamento das relações. Use as regras 2, 4, 6 e 9 para mover a operação de seleção o máximo possível para baixo da árvore. Mantenha os predicados da mesma relação juntos.
2. Combinar o Produto Cartesiano com a operação de seleção subsequente de onde o predicado representa uma condição de junção para uma operação de junção. Reescreva todos os Produtos Cartesianos com um predicado que defina a operação de junção.
3. Usar associação de operações binárias para rearranjar o *node* folha de modo que os *nodes* folhas com as operações de seleção mais restritivas sejam executadas primeiro. Realize a máxima redução (operação de seleção) quanto for possível antes de realizar uma operação binária (tipo uma junção). As regras 11 e 12 referem-se a associação de união e interseção para reordenar operações que resultem na menor junção sendo realizada primeiramente, o que significa que a segunda junção será baseada no primeiro menor operador.
4. Realizar projeção a mais cedo possível. Projeção reduz a cardinalidade da relação, e reduz o subsequente processamento da relação. Use a regra 3 para candidatar as operações de projeção. Use as regras 4, 7 e 10 para mover as operações de projeção o máximo possível para baixo da árvore. Mantenha atributos da mesma relação juntos.
5. 5.Compute expressões comuns uma vez. Se uma expressão comum aparecer mais que uma vez na árvore, e o resultado não for muito grnade, guarde o resultado depois de ter

sido computado, e depois reuse.

Um exemplo de como otimização de consulta usando heuristica reduz trabalho usando SELECT no código a seguir:

```
SELECT item.description, item.quantity
FROM store, inventory, item
WHERE store.number = inventory.store_no
AND inventory.item_code = item.code
AND store.number = 10;
```

A álgebra relacional para a consulta é,

```
∏item.description, item.quantity(σstore.number=inventory.store_no∧
  inventory.item_code=item.code∧store.number=10)(store X inventory) X item))
```

A árvore de álgebra relacional é mostrada na figura a seguir:

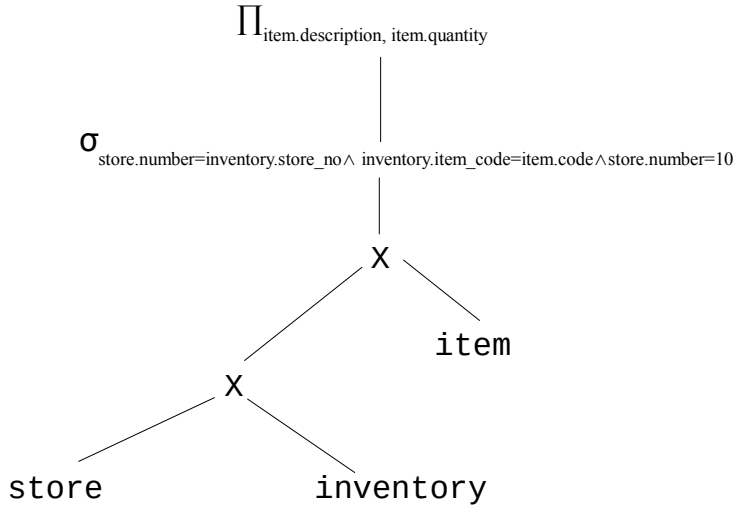

Figura 5: Grupo Simples da Álgebra Relacional

Aplicar a **regra 1** que separa a conjunção da operação de seleção em operações de seleção individuais. Depois, aplicar as **regras 2** e **6** para reordenar os operações de seleção e comutar as operações de seleção e Produto Cartesiano. A árvore resultante é mostrada a seguir:

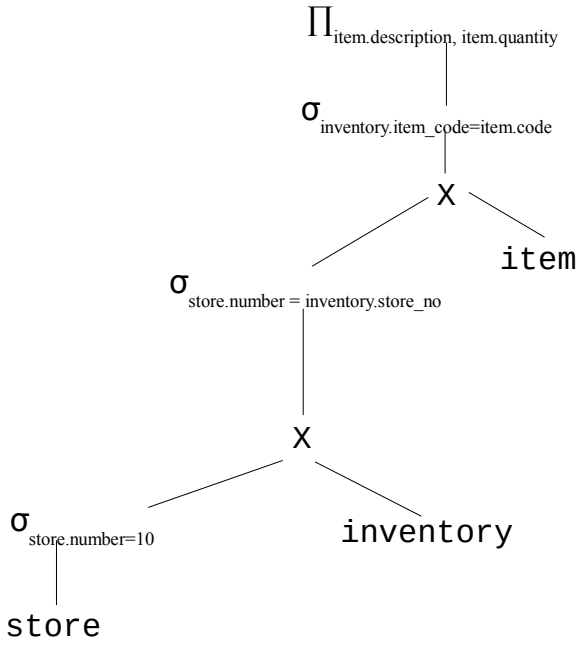

Figura 6: Aplicando as regras 1, 2 e 6

Mudando os Produtos Cartesianos como *equi-joins*, a árvore resultante é mostrada a seguir:

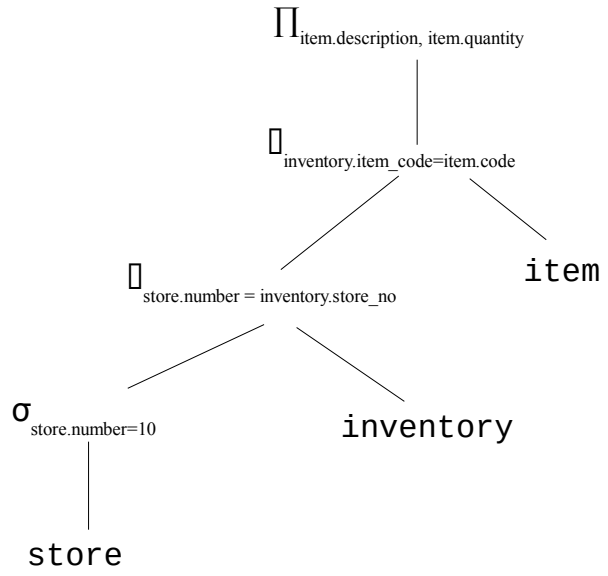

Figura 7: Mudando o produto cartesiano para equi-joins

Aplicando as **regras 4** e **7** para mover a projeção para baixo das *equi-joins*, e criando novas projeções conforme requerido. A árvore da algebra relacional resultante é mostrada a seguir:

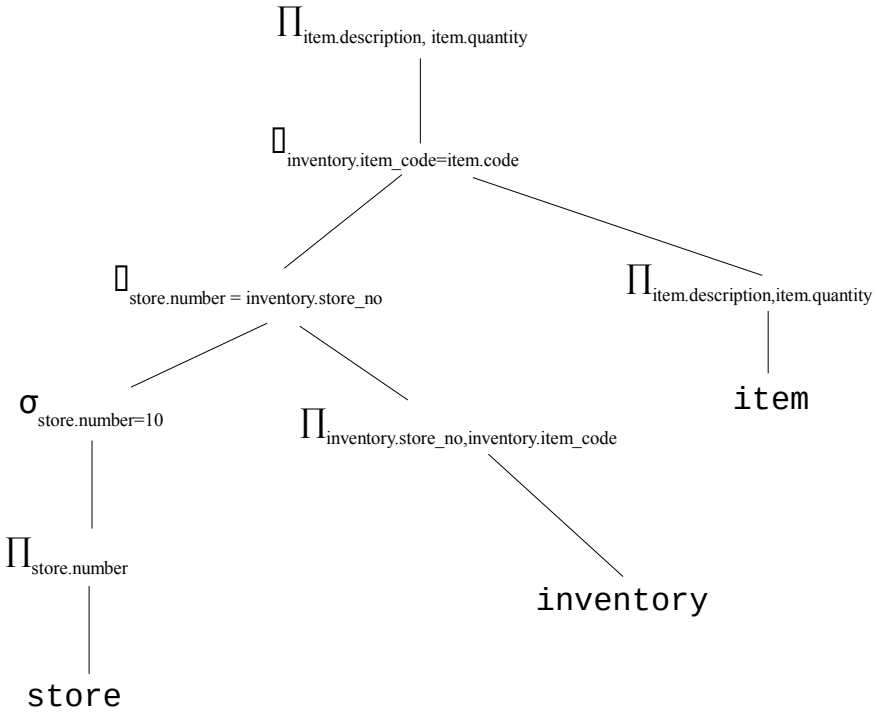

Figura 8: Modelo final da Árvore de Álgebra Relacional

5. Otimização de Consultas: Estimativa de custo para operações de álgebra relacional

O alvo da otimização de consulta é escolher o modelo mais eficiente. Para fazer isso, a implementação do sistema de banco de dados relacional usa fórmulas que estimam o custo para um número de opções, e seleciona a que possui o menor custo. Essa seção examina as diferentes opções disponíveis para implementação nas principais algebras relacionais. Consideraremos o seguinte:

- Concentrar o custo envolve acessos aos disco desde que eles sejam mais lentos comparados aos acessos à memória.
- Cada estimativa representa o número de acessos requeridos aos blocos de disco.
- Excluir o custo de escrever o resultado no disco.

Como será mostrado, a maioria das estimativas de custo são baseadas na cardinalidade da relação. Precisamos estar habilitadosa estimar o custo de relações intermediárias.

Estatísticas de banco de dados são requeridas para estimar o custo de acessos ao disco. Normalmente, um sistema de banco de dados teria estatisticas do banco de dados armazenado em seus catálogos de sistema. O sucesso de estimar o tamanho e custo da consulta vai depender o montante e frequencia da informação estatistica que os DBMs possuem. Gerlamente, DBMs armazenam o seguinte em seus catálogosde sistema:

Para cada relação R:

- nTuples(R). Esse é o número de linhas em uma relação, ou seja, sua cardinalidade.
- bFactor(R). Esse é o fator de bloco de R, ou seja, o número de linhas que cabe em um bloco.
- nBlocks(R). Esse é o número de bloco requeridos para armazenar R.

Se as linhas de R forem armazenadas juntas fisicamente, então:

```
nBlocks(R) = nTuples(R) / nFactor(R)
```

para cada atributo A:

- nDistinctA(R). Esse é o número de valores distintos que aparecem para o atributo A em relação a R.
- minA(R), maxA(R). Esse são os mínimo e máximo valores possíveis para o atributo A em R.
- SCA(R). Esse é a seleção cardinal do atributo A. Essa é a média de tuplas que satisfazem uma condição equivalente no atributo A.

Se os valores de A são uniformemente distribuidos em R, e existe pelo menos um valor que satisfaz a condição, então:

```
      1 se A é um atributo chave
SCA(R)=
      [nTuples(R) / nFactor(R)] de outra maneira

      [nTuple(R)*((maxA(R)-c)/maxA(R)-minA(R))] para desigualdade (A>c)
      [nTuple(R)*(c-(maxA(R))/maxA(R)-minA(R))] para desigualdade (A<c)
SCA(R)=   [nTuple(R)/nDistinctA(R)*n] para (A em c1,c2,...,cn)
      SCA(R)*SCB(R) para (A ^ B)
      SCA(R)+SCB(R)-SCA(R)*SCB(R) para (A v B)
```

Para cada indexI multi nível no atributo A:

- nLevelsA(I). Esse é o número de níveis no index I.
- nlfBlocksA(I). Esse é o número de blocos folha em I.

Em geral, as estatísticas são atualizadas periodicamente quando o sistema não está em seus períodos de pico ou quando o sistema está desocupado.

5.1. Operação de Seleção

Há uma série de maneiras de implementar a operação de seleção em função da estrutura do ar.quivo em que a relação é armazenada, e em qualquer atributo(s) envolvido no predicado que tiver sido indexado ou particionado. As principais estratégias e os seus correspondentes custos de computação é apresentado na Tabela 1.

Estratégias de Operação de Seleção	***Custo***
Busca Linear (Arquivo desordenado, sem índice)	Para igualdade de condição no atributo chave: `nBlocks(R)/2` Caso contrário: `nBlocks(R)`
Busca Binária (Arquivo ordenado, sem índice)	Para igualdade de condição no atributo ordenado: `log2(nBlocks(R))` Caso contrário: `log2(nBlocks(R)) +SCA(R)/bFactor(R) – 1`
Igualdade na chave *hash*	Assumindo sem *overflow*: 1
Igualdade de condição na chave primária	$nLevels_A(I) + 1$
Desigualdade de condição na chave primária	$nLevels_A(I) + [nBlocks(R)/2]$
Igualdade de condição ao gerar índice (secundário)	$nLevels_A(I) + [SC_A(R)/bFactor(R)]$
Igualdade de condição ao desmontar índice (secundário)	$nLevels_A(I) + SC_A(R)$
Desigualdade de contição no secundário B+-tree índice	$nLevels_A(I) + [nlfBlocks_A(I)/2 + nTuples(R)/2]$

Tabela 1: Resumo das Estratégias de Estimativa de Custo para as Operações de Seleção

5.2. Operação de Junção

A operação que deu maior trabalho foi a operação de junção, que além de ser um produto cartesiano, é a que mais consome tempo para processar, e temos que garantir que seja o mais eficiente possível. A Tabela 2 mostra as estratégias empregadas para a operação de junção, e o cálculo da estimativa dos custos de transformação.

Estratégias de Operação de Seleção	*Custo*
União de Laços Aninhados em Bloco	Para um buffer que tem apenas um bloco de R e S: `nBlocks(R) + (nBlocks(R) * nBlocks(S))` Para (nBuffer – 2) blocos para R: `nBlocks(R) + [nBlocks(S)*(nBlocks(R)/(nBuffer-2))]` Para todos os blocos de R que pode ser lido no buffer do banco de dados: `nBlocks(R) + nBlocks(S)`
União de Laço Aninhados Indexado	Depende do método de indexação: Para uma união do atributo A com uma chave primária: `nBlocks(R) + nTuples(R)*(nLevels`$_A$`(I) +1)` Para a agregação índice I a um atributo A: `nBlocks(R) + nTuples(R) * (nLevels`$_A$`(I) + SC`$_A$`(R)/bFactor(R)])`
União Sort-merge	Para classificar: `nBlocks(R) * [log`$_2$`(nBlocks(R)] + nBlocks(S) * [log`$_2$`(nBlocks(S)]` Para fundir: `nBlocks(R) + nBlocks(s)`
União de Partes	Para reter índice hash em memória: `3(nBlocks(R)+nBlocks(S)) + 2*max_partitions` Caso contrário: `2(nBlocks(R)+nBlocks(S)) * [log`$_{nBuffer-1}$`(nBlocks(S))-1] + nBlocks(R)+nBlocks(S)`

Tabela 2: Resumo das Estratégias de Estimativa de Custo para as Operações de União

5.3. Operação de Projeção

Por definição, operação de projeção é um operador unárioque define a relação, S, que contém um subconjunto de uma relação vertical, R, extraindo valores dos atributos especificados, e eliminando valores duplicados. Dois passos são necessários para implementar uma operação projeção.

1. Remover todos os atributos que não são requeridos; e
2. Eliminar todas as linhas duplicadas.

Para determinar o custo estimado da projeção, precisamos determinar o custo da operação para cada passo.

1. Determinando co custo estimado para o passo 1 (cardinalidade da operação de projeção). A tabela abaixo mostra o cálculo do custo estimado.

Passo 1 (Cardinalidade da Operação de Projeção)	*Custo*
Para uma projeção que contém o atributo chave, não é necessário, para a performance, eliminar as duplicidades desde que não exista duplicidade para o resuldade da relação.	nTuples(S) = nTuples(R)
Para uma projeção que contém um único atributo não-chave: S = ∏A(R)	nTuples(S) = SCA(R)
Para uma relação produzida por um projeto cartesiano (que é irrealista)	nTuples(S) ≤ min (nTuples(R) M ∏ (nDistinct ai(R)) i=1

Tabela 3: Resumo do custo estimado da Etapa 1 da Operação de Projeção

2. Determinar o custo da eliminação das linhas duplicadas. Existem basicamente duas estratégias para este passo que são mostradas na tabela abaixo.

Passo 2 Eliminando Registros Duplicados	*Custo*
Usando o método de classificação. O objetivo desta passagem é classificar os registros de relações reduzidas, com efeito de arranjar fileiras duplicadas adjacentes umas com as outras, o que torna mais fácil remoção.	Ler os registros de R e copiar para uma relação temporária: nBlocks(R) Classificando os registros: nBlocks(R)*[log2(nBlocks(R)] O Custo total é: nBlocks(R) + nBlocks(R)*[log2(nBlocks(R)]
Usando o método Hashing. Este método é muito utilizando quando um grande número de blocos de buffer estão disponíveis. Abaixo seguem os passoas envolvidos: 1. Um bloco é atribuída à leitura da relação R, e (nBuffer-1) buffers para a saída. 2. Para cada registro em R, remover atributos não desejados, e aplicar uma função hash, h, com a combinação dos restantes atributos. Escreva a redução na fila para o valor hash. 3. Leia cada um dos (nBuffer-1), na posição da partição. 4. Aplique um segundo valor hash, h2, e inserir o valor hash computado para um hash em memória da tabela hash. 5. Se a linha dividir-se no mesmo valor, verifique se os dois são os mesmos, e elimine o novo se for uma duplicação. 6. Uma vez que uma partição tenha sido processada, escreva os resultados em arquivo de retorno.	Se o numero de blocos requeridos pela tabela temporaria que resulta do projeto R antes a eliminação da duplicidade é nb, o custo estimado calculado é: nBlocks(R) + nb Isso assume que não existe sobrecarga de hasing.

Tabela 4: Resumo do Custo Estimado para a Eliminação de Duplicidade

5.4. Álgebra relacional conjunto operações

O conjunto de operações binárias, interseção e diferença se aplicam apenas para relações que são

união-compatível, ou seja, relações que tenham estruturas idênticas. Estas operações são normalmente implementadas como segue:

1. Classifique as relações ambas pelos mesmos atributos.
2. Examine através das relações ordenadas depois de obter os seguintes resultados:

Para uma União (R ∪ S), o resultado é colocado em fila se ele não aparece em nenhuma das relações originais. Eliminar as duplicidades, quando necessário.

Para uma Intersecção (R ∩ S), o resultado e colocado em fila se ele aparecer nas duas relações originais.

Para a Diferença (R - S), é colocada em uma fila o resultado se ele aparece em R mas não em S.

Uma algoritimo de junção classificar-fundir pode ser utilizados para todos os casos. O custo estimado para cálculo é:

```
nBlocks(R) + nBlocks(S) +
nBlocks(R)*[log2(nBlocks(R)] + nBlocks(S)*[log2(nBlocks(S)]
```

Porque duplicidades são eliminados quando realizar a união operação, é geralmente difícil de estimar a cardinalidade da operação, mas nós podemos computar para os limites superiores e inferiores, que é:

```
max(nTuples(R), nTuples(S)) ≤ nTuples(T) ≤ nTuples(R) + nTuples(S)
```

Para definir a operação de diferença, os limites superiores e inferiores são definidos como:

```
0 ≤ nTuples(T) ≤ nTuples(R)
```

6. Pipelining[1]

Um conceito que permite a melhoria no desempenho de consultas é *pipelining*. Também é conhecido como *on-the-fly processing*. Quando a consulta é processada, são assumidos os resultados de intermediárias operações de álgebra relational para escrever no disco, i.e., é armazenado em uma relação temporária para processar à próxima operação. Isto é chamado materialização. Uma aproximação alternativa é o *pipeline* dos resultados, i.e., passar os resultados de uma operação para outra operação sem criar uma relação temporária. Isto economiza o custo de criar relações temporárias e reler os resultados. Considere a operação de seleção com um predicado especificado no código abaixo:

```
σitem_code=1001σstore_no=10(inventory)
```

Que pode ser reescrito para:

```
σitem_code=1001σstore_no=10(inventory)
```

Usando uma técnica *non-pipeline*, podemos usar o índice para processar a primeira operação através de uma seleção eficaz do inventário, i.e., (store_no = 10). Armazenamos o resultado em uma relação temporária, e aplicamos este a segunda operação de seleção, i.e., item_code = 1001.

Usando a técnica *pipeline*, dispensamos o uso de uma relação temporária. Ao invés disso, aplicamos a segunda operação de seleção (item_code = 1001) para cada fila devolvida pela primeira operação de seleção (store_no = 10).

Geralmente, um *pipeline* é implementado como um processo separado ou um serviço dentro do sistema de administração de banco de dados. Cada *pipeline* leva um *stream* de linhas com sua contribuição e cria um *stream* de linhas como saída. Um *buffer* é usado para cada par de operações adjacentes segurar o início das linhas passadas da primeira para a segunda operação. Uma desvantagem em usar *pipelining* é que isso abre operações que não necessariamente estão disponíveis para serem processadas e pode restringir a escolha de algoritmos que serão utilizados.

1 Arquitetura do processador central que permite a execução de inúmeras atividades ao mesmo tempo

7. Consulta Otimizada do JavaDB

Nesta seção faremos uma avaliação do *optimizer* do **JavaDB** e os discutiremos os assuntos de desempenho na execução das declarações **DML**. JavaDB é usado freqüentemente como uma escolha para executar mais rapidamente a recuperação de registros no banco de dados. Por exemplo, usar um índice para um rápido *lookup* em entradas específicas, ou percorrer uma tabela inteira para obter as linhas necessárias. Para declarações que requereriam um *join*, é possível escolher qual tabela para examinar primeiro (ordem do *join*) e como unir as tabelas (estratégia do *join*). **Otimizar** significa que JavaDB faz a melhor escolha para recuperar as linhas, ordem do *join*, e uma melhor estratégia de *join*. A verdadeira otimização de consultas significa uma boa escolha embora como a consulta é escrita. O *optimizer* necessariamente não faz a melhor escolha.

7.1. Índices do Java DB e Caminhos de Acesso

Se um índice estiver definido para uma coluna ou colunas, o *optimizer* de consulta pode achar os dados nas colunas mais rapidamente. **JavaDB** automaticamente cria índices para as chaves primárias, chaves estrangeiras, e *constraints* únicas. Estes índices sempre estarão disponíveis ao *optimizer*, como também os que serão criados explicitamente com o comando CREATE INDEX. O modo como **JavaDB** obtém aos dados, por um índice ou diretamente pela tabela é chamado de **caminho de acesso**. Considere nosso banco de dados da Loja Orgânica. Para os exemplos nesta seção, consideraremos as seguintes tabelas:

```
Tabela store - Chave primária com o campo number.
Tabela item - Chave primária com o campo code.
Tabela inventory - Chave composta com os campos store_no e item_code
```

Um índice guarda todo valor da coluna, e dados para recuperar as linhas. Quando um índice inclui mais de uma coluna, como no caso da tabela *inventory*, a primeira coluna é a principal pelo qual as entradas são ordenadas. Se houver mais de um valor da primeira coluna, essas entradas são ordenadas para a segunda coluna. Por exemplo, quando são executadas as consultas mostradas abaixo, o índice definido para a chave primária da tabela *store* é usado.

```
SELECT * FROM store WHERE number = 20;
SELECT * FROM store WHERE number < 30;
SELECT * FROM store WHERE number >= 30;
```

Se **JavaDB** usa um índice para uma declaração, isto é dito que a declaração é otimizável. Declarações que possuem cláusulas WHERE que indicam começo e condições de parada permitem que **JavaDB** utilize o índice. Quer dizer, contam para o **JavaDB** o ponto ao qual iniciar e terminar sua procura do índice. Um índice percorre do começo ou são chamadas condições de parada de um índice indexando a procura.

Alguns predicados são diretamente otimizáveis se proverem um começo claro e pontos de término de parada e se:

- Usar uma referência de uma simples coluna para uma coluna (a coluna não deveria ser usada em uma expressão ou chamada de método).
- Isto refere-se a uma coluna que é a primeira ou só uma coluna no índice.
- A coluna é comparada a uma constante ou para uma expressão que não inclui colunas na mesma mesa. Os operadores de comparação que devem ser usados são:
  - =, <=, >=, <, >
  - IS NULL
- Se os predicados não forem diretamente otimizáveis, é dito que eles são indiretamente predicados de otimização, e são transformados interiormente em predicados de otimização. É dito que os seguintes operadores de comparação são indiretamente predicados de otimização:
  - BETWEEN

- LIKE (em certas situações)
- IN (em certas situações)

Alguns exemplo de consultas que não são otimizáveis são mostradas no código a seguir.

```
SELECT * FROM store WHERE number <> 30
-- por causa do operador não otimizável <>

SELECT * FROM INVENTORY WHERE item_code = 1004
-- até mesmo se item_code está definido no índice que não é o primeiro
```

Um índice cobre os meios da consulta de todas as colunas que especificados na consulta e fazem parte do índice. Estes são todos referenciados das colunas pela consulta; não só as colunas na cláusula WHERE. Um índice ao fazer isto acelera a execução da consulta mesmo se não há começo definido ou pontos de parada para um índice procurar.

Se uma cláusula WHERE contém pelo menos um predicado de otimização, então é otimizável. É procurado um índice para percorrer e poder usar qualificadores que mais adiante restringem o conjunto do resultado. Considere a consulta do código abaixo. O segundo predicado não é otimizável mas o primeiro é. O segundo predicado é usado por **JavaDB** para avaliar as entradas no índice e como percorrê-lo.

```
SELECT * FROM store WHERE number < 50 AND number <> 30
```

Os seguintes operadores de comparação são qualificadores válidos:

```
=, <, <=, >, >=, <>
IS NULL, IS NOT NULL
BETWEEN
LIKE
```

Para união é especificada com a palavra chave JOIN é otimizável o que significa que **JavaDB** pode usar um índice na tabela interna da união porque o começo e condições de parada estão sendo providas implicitamente pelas filas da tabela exterior. Também, a união construída usando os predicados tradicionais também são otimizáveis.

Se a declaração devolver virtualmente todos os dados da tabela, é melhor percorrer uma tabela em lugar de usar o índice. Neste caso, **JavaDB** pode evitar o passo intermediário de recuperar as linhas dos valores de *lookup* de índice. Considere a consulta mostrada abaixo:

```
SELECT * FROM item WHERE code < 2000
```

Para a tabela *item*, a maioria da coluna *item_code* é menos que 2000. Porém, para a questão mostrada no código abaixo, **JavaDB** usa um índice:

```
SELECT * FROM item WHERE code < 1010
```

JavaDB precisa manter os índices. Isto significa que quando são inseridas ou apagadas linhas, o sistema deve inserir ou apagar as linhas em todos os índices da tabela. Se uma atualização em uma coluna tem um índice para execução, então uma atualização nos índices também precisa ser feita. Isto significa que ao ter muitos índices pode acelerar as declarações SELECT mas em compensação piorar os tempos das declarações de inserção, atualização e exclusão.

7.2. Joins e Performance

Como uma revisão, um *join* recupera os dados de duas ou mais tabelas que usam uma ou mais colunas fundamentais para cada tabela. O desempenho de uma operação *join* varia amplamente. Fatores que afetam o desempenho uma operação *join* é a ordem, a estratégia e os índices.

Ordem do Join

JavaDB percorre as tabelas em uma ordem particular, i.e., acessa primeiro primeiro linhas em uma tabela, e esta é chamada de tabela exterior. Então, para cada linha qualificativa na tabela exterior, emparelha a coluna na segunda tabela que é conhecida como a tabela interna. **JavaDB**

só acessa uma vez a tabela exterior. Porém, a tabela interna pode ser acessada muitas vezes. As regras para uma ordem são as seguintes:

- Se uma operação *join* não tem nenhuma restrição na cláusula WHERE que limitaria o número de linhas devolvidas em um das tabelas, as seguintes regras se aplicam:
  - Se só uma tabela estiver com um índice unida na coluna ou colunas, é melhor fazer aquela tabela como uma tabela interna porque para cada um do muitos *lookups* de tabela interno, **JavaDB** poderá usar um índice em vez de percorrer a tabela inteira.
  - Desde que os índices em tabelas internas são acessados muitas vezes, se o índice em uma tabela for menor que o índice em outra, a tabela com o menor índice deveria ser a tabela interna. Porque índices menores (ou tabelas) podem ser colocados em caché (detendo a memória, enquanto permite evitar caros I/O para cada repetição).
- Por outro lado, se a questão tiver restrições na cláusula WHERE para uma tabela que devolveria algumas linhas desta, é melhor a tabela restringida ser a tabela exterior. Deste modo, o sistema de qualquer maneira terá que ir para a tabela interna somente algumas vezes.

Estratégias do Join

Existem duas estratégias de *Join* utilizadas no JavaDB:

1. ***Nested Loop***. Isto é o tipo mais comum de estratégia *join* em JavaDB. Para cada linha qualificativa na tabela exterior, usa um caminho de acesso apropriado para achar as linhas emparelhando na tabela interna.
2. ***Hash Join***. **JavaDB** constrói uma tabela de *hash* que representa todas as colunas selecionadas na tabela interna. Para cada linha qualificativa na tabela exterior, executa um *lookup* rápido nesta tabela de *hash* para obter os dados internos. Neste caso, percorrer uma única vez a tabela interna ou índice, e isso é quando a tabela de *hash* é construída. Este tipo de estratégia *join* é preferível em situações nas quais a tabela interna contém valores que não são iguais, e há muitas linhas qualificativas na tabela exterior. Requer que a cláusula WHERE é o *equijoin* de otimização, i.e.:
   - Deve usar o operador de igualdade (=) para comparar as colunas na tabela exterior para as colunas na tabela interna.
   - Deve ter a referência das colunas que são referências de coluna simples como previamente discutido em predicados diretos de otimização.

A tabela de *hash* para um *hash join* é preso em memória. Se ficar grande, usará o disco como um *overflow*. A otimização faz uma forte estimativa na memória requerida para a tabela de *hash*. Se a estimativa for maior que o limite de memória, optará por usar uma estratégia de *nested loop*.

Otimização Baseada em Custo

A otimização de consulta de **JavaDB** toma decisões baseadas em custo para determinar:

- **Qual índice (se qualquer) usar em cada tabela em uma consulta que define a escolha do *optimizer* de caminho de acesso**. Isto pode depender do número de linhas que serão lidas. Isto irá escolher um caminho que requer poucas colunas para ler. Se tem uma operação *join*, também dependerá de uma ordem. Às vezes, é possível conhecer exatamente quantas linhas serão lidas. Em outros tempos, faz uma suposição educada que faz uso de seletividade e estatísticas da cardinalidade que serão discutidas mais tarde.
- **Qual ordem usar**. A ordem pode afetar qual índice será usado. O *optimizer* pode escolher um índice como o caminho de acesso para uma tabela se for a tabela interna; não se é a tabela exterior, e que não há nenhuma qualificação adicional. Escolhe só uma ordem de tabelas dentro de uma simples cláusula FROM. A maioria usa a palavra chave JOIN é aplainado em uma simples união, de forma que escolhe a ordem. Não escolhe a ordem para a união exterior; usa uma ordem especificada na declaração. Ao selecionar uma ordem, leva em conta:
  - Tamanho de cada tabela

- Índices disponíveis em cada tabela
- Se um índice em uma tabela é útil em uma ordem particular
- Número de linhas e páginas procuradas para cada tabela em uma ordem

- **Qual estratégia usar**. O *optimizer* compara o custo de escolher um *hash* (se um *hash* for possível) para o custo de escolher um *nested loop* e escolhe a estratégia mais barata. A tabela de *hash* usará a memória e os meios que precisar para aproximar o tamanho das tabelas de *hash*. Possui um limite superior no tamanho de uma tabela na qual considerará um *hash*. Não considerará um *hash* para uma declaração se calcular que o tamanho da tabela de *hash* excederia o limite do uso de memória para uma tabela. Escolherá, então, uma estratégia *nested loop*.
- **Evitar a escolha adicional**. Algumas declarações de SQL exigiriam ordenar dados que são incluídos em ORDER BY, GROUP BY e DISTINCT. Funções de agregação como MIN() e MAX() também necessita ordenar os dados. **JavaDB** pode às vezes evitar a escolha usando os seguintes passos:
  - **Custo baseado em ORDER-BY**. Se uma consulta em uma única tabela tem uma cláusula de ORDER-BY em uma única coluna, e há um índice naquela coluna, então ordenações podem ser evitadas desde que **JavaDB** possa usar o índice como um caminho de acesso.
  - **Evitando o Não-custo Baseado em Ordenação ou Filtro de Tuplas**. Declarações que usam as palavras chaves DISTINCT ou GROUP-BY, precisa executar dois passos separados:
    - O primeiro passo envolve a escolha das colunas selecionadas. Então, pode descartar as filas duplicadas ou as filas agrupadas. JavaDB pode evitar a escolha com filtro, i.e., as filas são lidas em uma ordem útil.
    - Para a palavra chave DISTINCT, JavaDB pode filtrar simplesmente valores fora de duplicatas quando são achados os resultados e voltados cedo para o usuário. O filtro é aplicado quando as seguintes condições forem estabelecidas:
      - A cláusula SELECT é composta inteiramente de colunas com simples referências e constantes
      - Simples referências de coluna vêm da mesma tabela, e o *optimizer* pode escolher a tabela para ser a tabela externa no bloco da consulta.
      - Um índice é escolhido como o caminho de acesso para a tabela.
      - As referências de simples coluna na cláusula SELECT, e qualquer referência de coluna simples da tabela tem predicados de igualdade, é um prefixo do índice que o *optimizer* pode selecionar como o caminho de acesso para a tabela.

    Veja o código a seguir para o **JavaDB** versão 10.3:

```
CREATE TABLE t1(c1 INT, c2 INT, c3 INT, c4 INT)
CREATE INDEX i1 ON t1(c1)
CREATE INDEX i1_2_3_4 ON t1(c1, c2, c3, c4)

--Este é o caso mais comum no qual são aplicados filtros:
SELECT DISTINCT c1 FROM t1

--Predicados de igualdade permitem os seguintes filtros:
SELECT DISTINCT c2 FROM t1 WHERE c1 = 5
SELECT DISTINCT c2, c4 FROM t1 WHERE c1 = 5 and c3 = 7
--As colunas não precisam estar na mesma ordem como o índice
SELECT DISTINCT c2, c1 FROM t1
```

    Para a cláusula GROUP BY, JavaDB pode agregar um grupo de linhas até que um novo conjunto de linhas seja descoberto e retorna um resultado mais rápido para o usuário. Filtros são aplicados quando as seguintes condições foram conhecidas:

- Todas as colunas do agrupamento vêm da mesma tabela e o *optimizer* escolher a tabela da consulta ser a tabela externa no bloco desta.
- O optimizer escolheu um índice como o caminho de acesso para a mesa em questão.
- As colunas de agrupamento, mais que qualquer referência de coluna simples da tabela que tem predicados de igualdade neles, são um prefixo do índice que o *optimizer* seleciona como o caminho de acesso para a tabela.

Veja o código a seguir para o **JavaDB** versão 10.3:

- **Otimização com MIN() e MAX()**. O *optimizer* sabe que pode evitar interação por todas as linhas fontes em um resultado para computar um MIN() ou MAX() agregado quando dados já estão na ordem certa. Quando são garantidos dados estar na ordem certa, **JavaDB** pode imediatamente ir para a menor (mínima) ou maior (máxima) linha. As condições seguintes devem ser satisfeitas:
  - O MIN() ou MAX() é a única entrada na lista SELECT.
  - O MIN() ou MAX() está em uma referência de coluna simples e não em uma expressão.
  - Para o MAX(), não deve haver uma cláusula WHERE.
  - Para MIN():
    - A tabela de referência e a tabela externa no *optimizer* escolheu uma ordem para o bloco de consulta.
    - O *optimizer* escolheu um índice que contém a coluna de referência como o caminho de acesso.
    - A coluna de referência é a primeira coluna fundamental naquele índice OU a coluna de referência é uma coluna fundamental naquele índice e os predicados de igualdade existem em todas as colunas de chave antes da referência da coluna simples naquele índice.

  Exemplos serão mostrados abaixo.

```
-- Por exemplo, o optimizer pode usar esta otimização para as seguintes
-- questões (se o optimizer usa o apropriado índices como os caminhos de
-- acesso):

-- índice com o número
SELECT MIN(number) FROM store

-- índice com o número
SELECT MAX(number) FROM store

-- índice com o segment_number, flight_id
SELECT MIN(item_code) FROM inventory
WHERE store_no = 10
```

- Em uma seleção de *bulk fetch*. Um *bulk fetch* vai buscar mais que um coluna de cada vez. É mais rápido que recobrar uma linha de cada vez quando um número grande de linhas for qualificada para cada percorrer a tabela ou índice. Utiliza uma memória extra para segurar as linhas pre-buscar. É evitado em situações nos quais a memória é escassa. É automaticamente trocado os cursores de *updatable*, dos *hash join*, para as declarações que devolvem uma única linha, e para subqueries. **JavaDB** vai buscar 16 linhas de cada vez por padrão.

7.3. Seletividade e Estatísticas de Cardinalidade

Na ordem para o *optimizer* decidir em um caminho de acesso para uma tabela particular, i.e., se usar o índice ou percorrer a tabela, deve determinar o número de linhas que serão percorridas nos discos. Saber exatamente o número de linhas que serão percorridas, ou calcular este número.

Como o *optimizer* determina o número de linhas da tabela?

O catálogo do sistema do JavaDB conta as linhas de uma tabela. O sismtea mantém isto automaticamente. E é usado pelo *optimizer* determinar o número de filas que será percorrido nos discos. Quando um sistema não fizer paralisação de empresas corretamente, a conta das linhas armazenadas ficarão inexatas. Quando um sistema executar uma paralisação de empresas por ordem, irá atualizar a conta de linha armazenada. As contas das linhas são automaticamente atualizadas quando o **JavaDB** executar uma consulta que percorre toda a base de dados. Considere a declaração SELECT a seguir:

```
SELECT * FROM store
```

Executando os conjuntos de consulta seguintes a conta de tabela para guardar o valor correto. JavaDB também fixa a conta de linha armazenada em uma tabela para um valor correto sempre que um usuário criar um novo índice ou chave primária, únicas, ou chave estrangeira na tabela.

```
SELECT * FROM inventory
WHERE store_no = 10;
```

Como o *optimizer* extima o número de linhas da tabela?

Quando um índice estiver disponível, o *optimizer* calcula o número de linhas que serão percorridas pelo disco. A precisão da estimativa está baseado no tipo consulta que inicia a otimização, especificamente, examina a condição de consulta conhecida ou examina com condição de procura desconhecida.

1. **Consultas com condição de procura conhecida**. É dito que uma questão sabe a condição de procura quando o começo e condições de parada são conhecidas a compile tempo. Neste caso, o optimizer usa o próprio índice para calcular o número de filas que serão esquadrinhadas de disco. Considere a consulta abaixo:

   ```
   SELECT * FROM inventory
   WHERE store_no = 10;
   ```

   A condição de pesquisa tem um valor de procura que é conhecido (store_no = 10). O *optimizer* pode fazer uma estimativa precisa baseado no índice definido na chave primária (store_no, item_code) para a tabela *inventory*. O Campo *store_no* é a primeira coluna da chave composta. Também, se o índice é sem igual, e a cláusula WHERE envolve um operador de igualdade (=) ou comparação IS NULL, a optimização conhecida só uma única linha será percorrida no disco. Considere a consulta abaixo. Neste exemplo, está a chave priimaria da tabela *store*. Então, o sistema indexado na chave primária é considerado um índice sem igual. O *optimizer* sabe que esta questão devolverá uma única linha.

   ```
   SELECT * FROM store
   WHERE number = 10
   ```

2. **Examinar condição de consulta desconhecida**. Há questões que têm condições de procura desconhecidas como a cláusula WHERE que pode conter parâmetros dinâmicos que só são conhecido durante tempo de execução, ou quando a declaração envolve um *join*. O *optimizer* fará uma suposição no número de linhas devolvidas usando os dados de **seletividade** das tabelas para uma cláusula WHERE em particular. Seletividade recorre à fração de linhas que serão devolvidas da tabela para a cláusula WHERE em particular. Se a seletividade para uma operação de procura particular é 0.10, e a tabela contém 100 linhas, as estimativas de optimizer devolve 10 linhas. Note que esta apenas uma suposição.

   JavaDB determina a selectividade para a cláusula WHERE usando:

   - **Seletividade de Estatísticas de Cardinalidade**. Estatísticas de Cardinalidade são computadas pelo JavaDB, e armazenadas nas tabelas do sistema. JavaDB usa isto se:
     - Existem as estatísticas
     - As colunas pertinentes na cláusula WHERE são colunas principais em um índice

- As colunas que usam o operador de igualdade são comparadas (=)
- As Estatísticas de Cardinalidade serão discutidas na próxima seção.

- **Seletividade de Suposições Hard-Wired**. Estas são suposições hard-wired para a JavaDB. Usa um número fixo que tenta descrever a porcentagem de filas que provavelmente serão devolvidas. Necessariamente poderia não corresponder à seletividade atual da operação em todo caso. A tabela mostra 5 a seletividade de várias operações para índice de procura quando os valores forem desconhecidos com antecedência e estatísticas não são usadas.

Operator	*Selectivity*
operadores relacionais como =, >=, <=, <, <> quando o tipo de dado do parâmetro é um lógico	0.5 (50%)
Outros operadores (exceto para IS NULL e IS NOT NULL) quando o tipo de dado do parâmetro é um lógico	0.5 (50%)
IS NULL	0.1 (10%)
IS NOT NULL	0.9 (90%)
Operador =	0.1 (10%)
>, >=, <, <=	0.33 (3%)
<> comparado com um tipo não-lógico	0.90 (90%)
LIKE transformado para um predicato LIKE	1.0 (100%)
Operadores >= e < quando transformado internamente com um LIKE	0.25 (0.5 x 0.5)
Operadores >= e <= quando transformado internamente com um BETWEEN	0.25 (0.5 x0.5)

Tabela 5: Suposições de Selectividade Hard-wired

O que são estatísticas de cardinalidade?

JavaDB calcula e armazena nas tabelas de sistema o seguinte:

- número de linhas em cada tabela
- número de valores únicos para um conjunto de colunas para as principais em uma chave de índice que também é conhecido como o cardinality

Colunas principais recorrem à primeira coluna, ou a primeira e segunda coluna, ou a primeira, segunda, e terceira coluna de um índice (e assim por diante). JavaDB não pode computar o número de colunas para as quais uma combinação das colunas não-principais é única.

São unidas as Estatísticas de Cardinalidade nas chaves de um índice quando este for criado.

Java cria automaticamente estatísticas novas ou atualizações existentes para as seguintes operações:

- Quando um índice novo em uma tabela não-vazia existente é criado. São automaticamente criadas estatísticas para o índice novo.
- Quando uma chave primária, constrangimento fundamental sem igual, ou estrangeiro para uma mesa non-vazia existente é somado. Se houver nenhum índice existente que pode ser usado para a chave nova ou constrangimento, JavaDB cria estatísticas automaticamente para os novos índices.
- Quando executar o procedimento de sistema SYSCS_UTIL.SYSCS_COMPRESS_TABLE. São criadas estatísticas automaticamente para todos os índices se estas já não existirem.
- Quando uma coluna que faz parte do índice de uma tabela é encerrada, são encerradas as estatísticas para o índice afetado. Estatísticas são automaticamente atualizadas para os outros índices na tabela.

É possível que estatísticas possam ser passadas porque existem poucos casos limitados que são automaticamente atualizados. As declarações de inserção, atualização, e exclusão não causam atualização das estatísticas. Deve estar atento que as estatísticas passadas podem reduzir a velocidade seu sistema, porque pioram a precisão do *optimizer* quanto as estimativas de seletividade.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Instituto Gaudium
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.