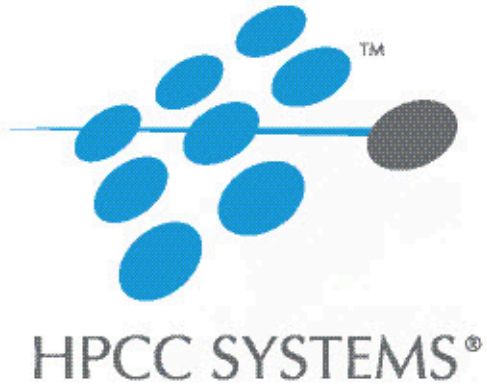
ECL Guia do Programador

Equipe de documentação de Boca Raton



ECL Guia do Programador

Equipe de documentação de Boca Raton

Sua opinião e comentários sobre este documento são muito bem-vindos e podem ser enviados por e-mail para <docfeedback@hpccsystems.com>

Inclua a frase **Feedback sobre documentação** na linha de assunto e indique o nome do documento, o número das páginas e número da versão atual no corpo da mensagem.

2022 Version 8.8.34-1

Conceitos de Programação ECL

Criação de Atributos

Similaridades e Diferenças

As semelhanças vêm da exigência fundamental de solucionar qualquer problema de processamento: Primeiro, entender o problema.

Depois disso, em diversos ambientes de programação, usa-se uma abordagem descendente para mapear a linha de lógica mais direta para que a entrada seja transformada no resultado desejado. É aí que o processo se diverge no ECL, uma vez que a própria ferramenta exige uma maneira diferente de se pensar sobre como formar a solução – e a rota direta nem sempre é a mais rápida. ECL exige uma abordagem ascendente para a solução de problemas.

Programação "Atômica"

No ECL, após entender qual resultado final deve ser obtido, a linha direta do início do problema até o resultado final é ignorada e, em vez disso, o problema começa a ser dividido no máximo número de partes possível – quanto menor ele for, melhor. Ao criar bits "atômicos" de código ECL, você cuidou de toda a parte conhecida e fácil do problema. Isso normalmente consiste em 80% do caminho até a solução sem precisar fazer nada terrivelmente difícil.

Depois de ter reduzido os bits ao máximo possível, é possível então começar a combiná-los para percorrer os outros 20% do caminho e chegar à solução final. Em outras palavras, você começa fazendo as partes pequenas mais fáceis e depois usa esses bits de forma combinada para gerar uma lógica cada vez mais complexa que desenvolve sua solução de modo orgânico.

Soluções crescentes

Os blocos de construção de atributos básicos no ECL são os tipos de atributos Set, Boolean, Recordset e Value. Cada um desses pode ser "atomizado" conforme necessário para que possam ser usados de forma combinada para produzir "moléculas" de lógica mais complexa que podem então ser novamente combinadas para produzir um "organismo" completo que gera o resultado final.

Por exemplo, suponha que você tem um problema que exige a produção de um grupo de registros onde um determinado campo em sua saída precisa conter um dos vários valores especificados (vamos imaginar 1, 3, 4 ou 7). Em muitas linguagens de programação, o pseudo-código para produzir esse resultado se pareceria o seguinte:

```
Start at top of MyFile
Loop through MyFile records
 If MyField = 1 or MyField = 3 or MyField = 4 or MyField = 7
  Include record in output set
 Else
  Throw out record and go back to top of loop
end if and loop
```

Enquanto no ECL, o código atual seria:

```
SetValidValues := [1,3,4,7];                    //Set Definition
IsValidRec := MyFile.MyField IN SetValidValues; //Boolean
ValRecsMyFile := MyFile(IsValidRec);            //filtered Recordset
OUTPUT(ValRecsMyFile);
```

O processo por trás da programação desse código é:

"Sei que tenho um conjunto de valores constantes na especificação, por isso posso começar criando um atributo de conjunto dos valores válidos...

"E agora que tenho um conjunto definido, posso usar esse conjunto para criar um atributo booleano para testar se o campo no qual estou interessado contém um dos valores válidos...

"E agora que tenho um booleano definido, posso usar esse booleano como a condição de filtro para produzir o recordset de que preciso para obter o resultado."

O processo começa com a criação de um "átomo" de atributo do conjunto, seguido de seu uso para criar a "molécula" booleana e depois usar essa "molécula" booleana para ampliar o "organismo" – a solução final.

ECL "feio" também é possível

É claro que o código ECL acima poderia ser programado da seguinte forma (seguindo um processo de raciocínio mais descendente):

```
OUTPUT(MyFile(MyField IN [1,3,4,7]));
```

O resultado final, neste caso, seria o mesmo.

No entanto, a utilidade geral desse código é reduzida significativamente porque, na primeira forma, todos os bits "atômicos" estão disponíveis para serem reutilizados em outro lugar quando problemas similares acontecem. Na segunda forma, eles não estão disponíveis. E em todos os estilos de programação, a reutilização de código é considerada uma boa prática.

Otimização Fácil

O mais importante: ao dividir seu código ECL nos menores componentes possíveis, permite-se que o compilador de otimização do ECL trabalhe da melhor forma possível para determinar como o resultado desejado pode ser alcançado. Isso leva a outra dicotomia entre ECL e outras linguagens de programação: normalmente, quanto menos códigos você escrever, mais "elegante" será a solução; porém, no ECL, quanto mais códigos, melhor e mais elegante será a solução gerada para você. Lembre-se de que atributos são apenas **definições** que informam o compilador o que fazer, não como fazer Quanto mais você "desintegrar" o problema em suas partes componentes, maior será a margem de trabalho oferecida ao otimizador para a produção do código executável mais eficiente e rápido possível.

Criando Dados de Exemplo

Obtendo Arquivos de Código

Todo o código de exemplo do *Guia do Programador* está disponível para download no site do HPCC Systems na mesma página do PDF (clique aqui) Para disponibilizar o código para uso no ECL IDE, basta:

1. Faça o download do arquivo ECL_Code_Files.ZIP.
2. No ECL IDE, escolha a pasta “My Files", clique com o botão direito e selecione “Insert Folder" no menu pop-up.
3. Nomeie a nova pasta "ProgrammersGuide" (observe que espaços NÃO SÃO permitidos em nomes de pasta de repositório ECL).
4. No ECL IDE, escolha a pasta "ProgrammersGuide", clique com o botão direito e selecione “Locate File in Explorer" no menu pop-up.
5. Extraia todos os arquivos do arquivo ECL_Code_Files.ZIP para sua nova pasta.

Gerando Arquivos

O código que gera os dados de exemplo usados por todos os artigos do *Guia do Programador* está contido em um arquivo denominado Gendata.ECL. Basta abrir esse arquivo no ECL IDE (selecione **File > Open** no menu, depois selecione o arquivo Gendata.ECL e ele será aberto em uma janela do compilador); em seguida, pressione o botão **Submit** para gerar os arquivos de dados. O processo leva alguns minutos para ser executado. Aqui está o código, totalmente explicado.

Algumas constantes

```
IMPORT std;

P_Mult1 := 1000;
P_Mult2 := 1000;
TotalParents := P_Mult1 * P_Mult2;
TotalChildren := 5000000;
```

Essas constantes definem os números usados para gerar 1.000.000 de registros principais e 5.000.000 de registros secundários. Ao defini-los como atributos, o código poderia facilmente gerar um número inferior de registros principais (como 10.000 ao alterar ambos os multiplicadores de 1.000 para 100). No entanto, o código, conforme escrito, foi projetado para no máximo 1.000.000 registros principais e precisaria ser alterado em vários locais para suportar novas gerações de registros. O número de registros secundários pode ser alterado em qualquer direção sem outras alterações ao código (embora, se elevado excessivamente, possa haver erros de tempo de execução quanto ao tamanho máximo do registro variável para o dataset secundário aninhado). Para fins de demonstrar as técnicas nestes artigos do *Guia do Programador* , 1.000.000 registros principais e 5.000.000 registros secundários são mais do que o suficiente.

Estruturas RECORD

```
Layout_Person := RECORD
  UNSIGNED3 PersonID;
  STRING15  FirstName;
  STRING25  LastName;
  STRING1   MiddleInitial;
  STRING1   Gender;
  STRING42  Street;
  STRING20  City;
  STRING2   State;
```

```
  STRING5   Zip;
END;

Layout_Accounts := RECORD
  STRING20  Account;
  STRING8   OpenDate;
  STRING2   IndustryCode;
  STRING1   AcctType;
  STRING1   AcctRate;
  UNSIGNED1 Code1;
  UNSIGNED1 Code2;
  UNSIGNED4 HighCredit;
  UNSIGNED4 Balance;
END;

Layout_Accounts_Link := RECORD
  UNSIGNED3 PersonID;
  Layout_Accounts;
END;

Layout_Combined := RECORD,MAXLENGTH(1000)
  Layout_Person;
  DATASET(Layout_Accounts) Accounts;
END;
```

Essas estruturas RECORD definem os layouts de campo para três datasets: um arquivo principal (Layout_Person), um arquivo secundário (Layout_Accounts_Link) e um arquivo principal com dataset secundário aninhado (Layout_Combined). Eles são usados para gerar três arquivos de dados separados. As estruturas Layout_Accounts_Link e Layout_Accounts são separadas porque os registros secundários na estrutura aninhada não irão conter o campo de vínculo para o principal, considerando que o arquivo secundário separado precisa conter o link.

Dados iniciais

```
//define data for record generation:
            //100 possible middle initials, 52 letters and 48 blanks
SetMiddleInitials := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ                  ' +
                     'ABCDEFGHIJKLMNOPQRSTUVWXYZ                  ';

            //1000 First names
SET OF STRING14     SetFnames := [
  'TIMTOHY       ','ALCIAN        ','CHAMENE       ',
  ... ];

           //1000 Last names
SET OF STRING16 SetLnames := [
  'BIALES          ','COOLING         ','CROTHALL        ',
  ... ];
```

Esses conjuntos definem os dados que serão usados para gerar os registros. Ao fornecer 1.000 nomes e sobrenomes, este código pode gerar 1.000.000 nomes únicos.

```
//2400 street addresses to choose from
SET OF STRING31    SetStreets := [
     '1 SANDHURST DR                ','1 SPENCER LN              ',
     ... ];

         //Matched sets of 9540 City,State, Zips
SET OF STRING15 SetCity := [
     'ABBEVILLE      ','ABBOTTSTOWN    ','ABELL          ',
     ... ];

SET OF STRING2 SetStates := [
     'LA','PA','MD','NC','MD','TX','TX','IL','MA','LA','WI','NJ',
```

```
    ... ];

SET OF STRING5 SetZips := [
    '70510','17301','20606','28315','21005','79311','79604',
    ... ];
```

Contar com 2.400 endereços e 9.540 combinações (válidas) de cidade, estado e CEP oferece diversas oportunidades para gerar um conjunto significativo de endereços.

Gerando Registros Pais

```
BlankSet := DATASET([{0,'','','','','','','','',[]}],
                    Layout_Combined);
CountCSZ := 9540;
```

Aqui está o início do código de geração de dados. O BlankSet é um registro "semente" vazio e único usado para iniciar o processo. O atributo CountCSZ define simplesmente o número máximo de combinações de cidade, estado e CEP disponível para uso em cálculos subsequentes que determinarão o que usar em um determinado registro.

```
Layout_Combined CreateRecs(Layout_Combined L,
                           INTEGER C,
                           INTEGER W) := TRANSFORM
  SELF.FirstName := IF(W=1,SetFnames[C],L.FirstName);
  SELF.LastName  := IF(W=2,SetLnames[C],L.LastName);
  SELF := L;
END;

base_fn  := NORMALIZE(BlankSet,P_Mult1,CreateRecs(LEFT,COUNTER,1));

base_fln := NORMALIZE(base_fn ,P_Mult2,CreateRecs(LEFT,COUNTER,2));
```

A finalidade desse código é gerar 1.000.000 registros de nome/sobrenome únicos como um ponto de partida. A operação NORMALIZE é única no sentido de que seu segundo parâmetro define o número de vezes para acionar a função TRANSFORM para cada registro de entrada. Isso o torna excepcionalmente adequado para gerar o tipo de dados de "teste" de que precisamos.

Estamos realizando duas operações NORMALIZE aqui. A primeira gera 1.000 registros com nomes únicos a partir de um único registro em branco no DATASET BlankSet inline. A segunda operação usa os 1.000 registros da primeira NORMALIZE e cria 1.000 novos registros com sobrenomes únicos para cada registro de entrada, resultando em 1.000.000 de registros únicos de nome/sobrenome.

Um truque interessante aqui é usar uma única função TRANSFORM para as duas operações NORMALIZE. Definir TRANSFORM para receber um parâmetro "extra" (terceiro) em relação ao normal é o que permite fazer isso. Esse parâmetro simplesmente sinaliza qual passagem NORMALIZE o TRANSFORM está realizando.

```
Layout_Combined PopulateRecs(Layout_Combined L,
          Layout_Combined R,
          INTEGER HashVal) := TRANSFORM
  CSZ_Rec            := (HashVal % CountCSZ) + 1;
  SELF.PersonID      := IF(L.PersonID = 0,
                           Thorlib.Node() + 1,
                           L.PersonID + CLUSTERSIZE);
  SELF.MiddleInitial := SetMiddleInitials[(HashVal % 100) + 1 ];
  SELF.Gender        := CHOOSE((HashVal % 2) + 1,'F','M');
  SELF.Street        := SetStreets[(HashVal % 2400) + 1 ];
  SELF.City          := SetCity[CSZ_Rec];
  SELF.State         := SetStates[CSZ_Rec];
  SELF.Zip           := SetZips[CSZ_Rec];
  SELF               := R;
END;

base_fln_dist := DISTRIBUTE(base_fln,HASH32(FirstName,LastName));
```

```
base_people   := ITERATE(base_fln_dist,
                         PopulateRecs(LEFT,
                         RIGHT,
                         HASHCRC(RIGHT.FirstName,RIGHT.LastName)),
                         LOCAL);

base_people_dist := DISTRIBUTE(base_people,HASH32(PersonID));
```

Depois que as duas operações NORMALIZE realizarem seu trabalho, a próxima tarefa é preencher o restante dos campos. Uma vez que um desses campos é o PersonID, que é o campo de identificador único para o registro, a maneira mais rápida de preenchê-lo é com ITERATE usando a opção LOCAL . Ao usar a função Thorlib.Node() e a diretiva de compilador CLUSTERSIZE, é possível numerar unicamente cada registro em paralelo a cada nó com ITERATE. Você pode acabar com algumas lacunas na numeração até o final, mas já que o requisito aqui é a exclusividade e não a contiguidade, essas lacunas são irrelevantes. Uma vez que as primeiras duas operações NORMALIZE ocorreram em um único nó (confira as distorções mostradas no gráfico do ECL Watch), a primeira coisa a se fazer é usar a função DISTRIBUTE para os registros afim de que cada nó possua um conjunto proporcional dos dados para trabalhar. Desta forma, ITERATE pode realizar seu trabalho em cada conjunto de registros de forma paralela.

Para introduzir um elemento de aleatoriedade às escolhas de dados, o ITERATE passa um valor de hash para a função TRANSFORM como um terceiro parâmetro "extra". Essa é a mesma técnica usada anteriormente, porém passando os valores calculados em vez de constantes.

A definição do atributo CSZ _Rec ilustra o uso de definições de atributo local dentro das funções TRANSFORM. Definir a expressão uma vez e depois usá-la múltiplas vezes conforme necessário para produzir uma combinação válida de cidade, estado e CEP. O restante dos campos é preenchido por dados selecionados usando o que foi passado no valor hash em suas expressões. O operador de divisão de módulo (%--produz o restante da divisão) é usado para garantir que um valor seja calculado e que esteja no intervalo válido do número de elementos para o determinado conjunto de dados a partir do qual o campo é preenchido.

Gerando Registros Filhos

```
BlankKids := DATASET([{0,'','','','','',0,0,0,0}],
          Layout_Accounts_Link);

SetLinks  := SET(base_people,PersonID);

SetIndustryCodes := ['BB','DC','ON','FM','FP','FF','FC','FA','FZ',
                     'CG','FS','OC','ZZ','HZ','UT','HF','CS','DM',
                     'JA','FY','HT','UE','DZ','AT'];

SetAcctRates := ['1','0','9','*','Z','5','B','2',
                 '3','4','A','7','8','E','C'];

SetDateYears := ['1987','1988','1989','1990','1991','1992','1993',
                 '1994','1995','1996','1997','1998','1999','2000',
                 '2001','2002','2003','2004','2005','2006'];

SetMonthDays := [31,28,31,30,31,30,31,31,30,31,30,31];

SetNarrs     := [229,158,2,0,66,233,123,214,169,248,67,127,168,
                 65,208,114,73,218,238,57,125,113,88,
                 247,244,121,54,220,98,97];
```

Novamente, começamos definindo um registro "semente" para o processo como um DATASET em linha e vários DATASET adequados para os campos específicos. A função SET compila um conjunto de valores válidos de PersonID a ser usado para criar os vínculos entre os registros principal e secundário.

```
Layout_Accounts_Link CreateKids(Layout_Accounts_Link L,
                                INTEGER C) := TRANSFORM
```

```
  CSZ_IDX       := C % CountCSZ + 1;
  HashVal       := HASH32(SetCity[CSZ_IDX],SetStates[CSZ_IDX],SetZips[CSZ_IDX]);
  DateMonth     := HashVal % 12 + 1;
  SELF.PersonID := CHOOSE(TRUNCATE(C / TotalParents ) + 1,
                          IF(C % 2 = 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents - (C % TotalParents )]),
                          IF(C % 3 <> 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents  - (C % TotalParents )]),
                          IF(C % 5 = 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents  - (C % TotalParents )]),
                          IF(C % 7 <> 0,
                             SetLinks[C % TotalParents  + 1],
                             SetLinks[TotalParents  - (C % TotalParents )]),
                          SetLinks[C % TotalParents  + 1]);
 SELF.Account      := (STRING)HashVal;
 SELF.OpenDate     := SetDateYears[DateMonth] + INTFORMAT(DateMonth,2,1) +
                          INTFORMAT(HashVal % SetMonthDays[DateMonth]+1,2,1);
 SELF.IndustryCode := SetIndustrycodes[HashVal % 24 + 1];
 SELF.AcctType     := CHOOSE(HashVal%5+1,'O','R','I','9',' ');
 SELF.AcctRate     := SetAcctRates[HashVal % 15 + 1];
 SELF.Code1        := SetNarrs[HashVal % 15 + 1];
 SELF.Code2        := SetNarrs[HashVal % 15 + 16];
 SELF.HighCredit   := HashVal % 50000;
 SELF.Balance      := TRUNCATE((HashVal % 50000) * ((HashVal % 100 + 1) / 100));
END;

base_kids := NORMALIZE( BlankKids,
                        TotalChildren,
                        CreateKids(LEFT,COUNTER));
base_kids_dist := DISTRIBUTE(base_kids,HASH32(PersonID));
```

Esse processo é semelhante àquele usado para os registros principais. Desta vez, em vez de passar em um valor de hash, o atributo local realiza esse trabalho dentro da função TRANSFORM. Assim como antes, o valor de hash é usado para selecionar os dados reais em cada campo do registro.

A parte interessante aqui é a expressão para determinar o valor de campo PersonID. Uma vez que estamos gerando 5.000.000 de registros secundários, seria bastante simples apenas dar a cada registro principal cinco registros secundários. No entanto, os dados reais raramente são dessa forma. Consequentemente, a função CHOOSE é usada para selecionar um método diferente para cada conjunto de um milhão de registros secundários. O primeiro milhão usa a primeira expressão IF, o segundo milhão usa a segunda e assim por diante. Isso cria um número variável de registros secundários para cada registro principal no intervalo de um até nove.

Criando Registros Aninhados de Dataset Filhos

```
Layout_Combined AddKids(Layout_Combined L, base_kids R) := TRANSFORM
  SELF.Accounts := L.Accounts +
     ROW({R.Account,R.OpenDate,R.IndustryCode,
     R.AcctType,R.AcctRate,R.Code1,
     R.Code2,R.HighCredit,R.Balance},
         Layout_Accounts);
  SELF := L;
END;
base_combined := DENORMALIZE( base_people_dist,
    base_kids_dist,
    LEFT.PersonID = RIGHT.PersonID,
    AddKids(LEFT, RIGHT));
```

Agora que temos conjuntos de registros separados entre registros principal e secundário, o próximo passo é combiná-los em um único dataset com os dados secundários de cada registro principal aninhados no mesmo registro físico que

o registro principal. A razão para aninhar os dados secundários dessa forma é para facilitar as consultas de registro principal-secundário na Refinaria de Dados e no motor de entrega rápida de dados sem exigir o uso de etapas JOIN separadas para estabelecer os vínculos entre os registros principal e secundário.

A composição do dataset secundário aninhado requer a operação DENORMALIZE. Essa operação localiza os vínculos entre os registros principais e seus secundários associados, acionando a função TRANSFORM o mesmo número de vezes que houver registros secundários para cada registro principal. A técnica interessante aqui é o uso da função ROW para compor cada registro secundário aninhado adicional. Isso é feito para eliminar o campo de vínculo (PersonID) de cada registro secundário armazenado no dataset combinado, uma vez que esse é o mesmo valor contido no campo PersonID do registro principal.

Gravar arquivos em disco

```
O1 := OUTPUT(PROJECT(base_people_dist,Layout_Person),,
             '~PROGGUIDE::EXAMPLEDATA::People',OVERWRITE);

O2 := OUTPUT(base_kids_dist,,'~PROGGUIDE::EXAMPLEDATA::Accounts',OVERWRITE);

O3 := OUTPUT(base_combined,,'~PROGGUIDE::EXAMPLEDATA::PeopleAccts',OVERWRITE);

P1 := PARALLEL(O1,O2,O3);
```

Essas definições do atributo OUTPUT irão gravar os datasets em disco. Eles são gravados como definições de atributos, uma vez que serão usados em uma ação SEQUENTIAL. O atributo da ação PARALLEL simplesmente indica que todas essas gravações de discos podem ocorrer "simultaneamente" se o otimizador decidir que pode fazer isso.

O primeiro OUTPUT usa um PROJECT para produzir registros principais na forma de um arquivo separado, já que os dados foram originalmente gerados em uma estrutura RECORD que contém o campo DATASET secundário aninhado (Contas) em preparação para a criação do terceiro arquivo. O PROJECT elimina esse campo Contas vazio do resultado desse dataset.

```
D1 := DATASET('~PROGGUIDE::EXAMPLEDATA::People',
              {Layout_Person,UNSIGNED8 RecPos{virtual(fileposition)}}, THOR);

D2 := DATASET('~PROGGUIDE::EXAMPLEDATA::Accounts',
              {Layout_Accounts_Link,UNSIGNED8 RecPos{virtual(fileposition)}},THOR);

D3 := DATASET('~PROGGUIDE::EXAMPLEDATA::PeopleAccts',
              {,MAXLENGTH(1000) Layout_Combined,UNSIGNED8 RecPos{virtual(fileposition)}},THOR);
```

Essas declarações DATASET são necessárias para compilar índices. Os campos UNSIGNED8 RecPos são os campos virtuais (eles só existem no tempo de execução e não em disco), sendo os ponteiros do registro interno. Eles são mencionados aqui para que possamos referenciá-los nas declarações INDEX posteriores.

```
I1 := INDEX(D1,{PersonID,RecPos},'~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID');

I2 := INDEX(D2,{PersonID,RecPos},'~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID');

I3 := INDEX(D3,{PersonID,RecPos},'~PROGGUIDE::EXAMPLEDATA::KEYS::PeopleAccts.PersonID');

B1 := BUILD(I1,OVERWRITE);
B2 := BUILD(I2,OVERWRITE);
B3 := BUILD(I3,OVERWRITE);

P2 := PARALLEL(B1,B2,B3);
```

Essas declarações INDEX permitem que as ações BUILD usem o formato de parâmetro único. Novamente, o atributo da ação PARALLEL indica que toda a compilação de índice pode ser feita simultaneamente.

```
SEQUENTIAL(P1,P2);
```

Essa ação SEQUENTIAL simplesmente diz "grave todos os arquivos de dados em disco e depois compile os índices".

Definindo Arquivos

Depois que datasets e índices tiverem sido gravados em disco, é preciso declarar os arquivos para usá-los no código ECL de exemplo nos demais artigos. Essas declarações estão contidas no arquivo DeclareData.ECL . Para disponibilizá-las para o restante do código de exemplo, basta usar a função IMPORT. Dessa forma, no início de cada exemplo, você encontrará esta linha de código:

```
IMPORT $;
```

Isso IMPORTa todos os arquivos na pasta ProgrammersGuide (incluindo a definição de estrutura MODULE DeclareData). A referência a qualquer dado do DeclareData é feita ao anexar $.DeclareData ao nome da definição EXPORT que precisa ser usada, como mostrado a seguir:

```
MyFile := $.DeclareData.Person.File;  //rename $DeclareData.Person.File to MyFile to make
                                      //subsequent code simpler
```

Aqui está parte do código contido no arquivo DeclareData.ECL :

```
EXPORT DeclareData := MODULE

  EXPORT Layout_Person := RECORD
    UNSIGNED3 PersonID;
    STRING15  FirstName;
    STRING25  LastName;
    STRING1   MiddleInitial;
    STRING1   Gender;
    STRING42  Street;
    STRING20  City;
    STRING2   State;
    STRING5   Zip;
  END;

  EXPORT Layout_Accounts := RECORD
    STRING20   Account;
    STRING8    OpenDate;
    STRING2    IndustryCode;
    STRING1    AcctType;
    STRING1    AcctRate;
    UNSIGNED1  Code1;
    UNSIGNED1  Code2;
    UNSIGNED4  HighCredit;
    UNSIGNED4  Balance;
  END;

  EXPORT Layout_Accounts_Link := RECORD
    UNSIGNED3 PersonID;
    Layout_Accounts;
  END;

  SHARED Layout_Combined := RECORD,MAXLENGTH(1000)
    Layout_Person;
    DATASET(Layout_Accounts) Accounts;
  END;

  EXPORT Person := MODULE
    EXPORT File     := DATASET('~PROGGUIDE::EXAMPLEDATA::People',Layout_Person, THOR);
    EXPORT FilePlus := DATASET('~PROGGUIDE::EXAMPLEDATA::People',
                               {Layout_Person,UNSIGNED8 RecPos{virtual(fileposition)}}, THOR);
  END;
  EXPORT Accounts := DATASET('~PROGGUIDE::EXAMPLEDATA::Accounts',
                             {Layout_Accounts_Link,
```

```
                              UNSIGNED8 RecPos{virtual(fileposition)}},
                             THOR);
  EXPORT PersonAccounts:=   DATASET('~PROGGUIDE::EXAMPLEDATA::PeopleAccts',
                                    {Layout_Combined,
                                     UNSIGNED8 RecPos{virtual(fileposition)}},
                                     THOR);

  EXPORT IDX_Person_PersonID :=
  INDEX(Person,
        {PersonID,RecPos},
        '~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID');

  EXPORT IDX_Accounts_PersonID :=
  INDEX(Accounts,
        {PersonID,RecPos},
        '~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID');

  EXPORT IDX_PersonAccounts_PersonID :=
  INDEX(PersonAccounts,
        {PersonID,RecPos},
        '~PROGGUIDE::EXAMPLEDATA::KEYS::PeopleAccts.PersonID');

END;
```

Ao usar a estrutura MODULE como contêiner, todas as declarações DATASET e INDEX ficam em uma única janela de editor de atributos. Isso simplifica a manutenção e a atualização, além de garantir o acesso completo a todas elas.

Relatórios Cross-Tab

Relatórios de cross-tab são uma maneira bastante útil de descobrir informações estatísticas sobre os dados com os quais você trabalha. Eles podem ser facilmente produzidos usando a função TABLE e as funções agregadas (COUNT, SUM, MIN, MAX, AVE, VARIANCE, COVARIANCE, CORRELATION). O recordset resultante contém um único registro para cada valor único dos campos "agrupar por" especificados na função TABLE, juntamente com as estatísticas geradas com as funções agregadas.

Os parâmetros “agrupado por” da função TABLE são usados e duplicados como o primeiro conjunto de campos na estrutura RECORD, seguidos por qualquer número de acionamentos de função agregados, tudo isso usando a palavra-chave GROUP como substituto para o recordset exigido pelo primeiro parâmetro de cada uma das funções agregadas. A palavra-chave GROUP especifica a realização de uma operação agregada no grupo e é o segredo para criar um relatório de cross-tab. Isso cria uma tabela de resultados contendo uma linha única para cada valor único dos parâmetros "agrupar por".

Uma CrossTab Simples

O código de exemplo abaixo (contido no arquivo CrossTab..ECL) produz um resultado de State/CountAccts com contas do dataset secundário aninhado criado pelo código GenData.ECL (consulte o artigo **Criando Dados de Exemplo**):

```
IMPORT $;
Person := $.DeclareData.PersonAccounts;

CountAccts := COUNT(Person.Accounts);

MyReportFormat1 := RECORD
  State     := Person.State;
  A1        := CountAccts;
 GroupCount := COUNT(GROUP);
END;

RepTable1 := TABLE(Person,MyReportFormat1,State,CountAccts );
OUTPUT(RepTable1);

/* The result set would look something like this:
  State    A1  GroupCount
   AK     1    7
   AK     2    3
   AL     1    42
   AL     2    54
   AR     1    103
   AR     2    89
   AR     3    2    */
```

Pequenas modificações permitirão a produção de algumas estatísticas mais sofisticadas, como:

```
MyReportFormat2 := RECORD
  State{cardinality(56)}  := Person.State;
  A1          := CountAccts;
  GroupCount  := COUNT(GROUP);
  MaleCount   := COUNT(GROUP,Person.Gender = 'M');
  FemaleCount := COUNT(GROUP,Person.Gender = 'F');
 END;

RepTable2 := TABLE(Person,MyReportFormat2,State,CountAccts );

OUTPUT(RepTable2);
```

Isso adiciona um detalhamento à contagem homens e mulheres em cada categoria, usando o segundo parâmetro opcional para COUNT (disponível para uso apenas em estruturas RECORD onde seu primeiro parâmetro é a palavra-chave GROUP).

A adição de {cardinality(56)} à definição State é uma dica para o otimizador de que há exatamente 56 valores possíveis nesse campo, permitindo que ele selecione o melhor algoritmo para produzir o resultado da maneira mais rápida possível.

As possibilidades são infinitas quanto ao tipo de estatísticas que podem ser geradas em relação a qualquer conjunto de dados.

Um exemplo mais complexo

Como um exemplo levemente mais complexo, o código a seguir produz uma tabela de resultado de uma cross-tab com o saldo médio de uma transação em um cartão bancário, média elevada do crédito em uma transação com cartão bancário e o saldo total médio em cartões bancários, tabulados por estado e sexo.

Esse código demonstra o uso de atributos agregados separados como os parâmetros de valor para a função agregada na cross-tab.

```
IsValidType(STRING1 PassedType) := PassedType IN ['O', 'R', 'I'];

IsRevolv := Person.Accounts.AcctType = 'R' OR
        (~IsValidType(Person.Accounts.AcctType) AND
         Person.Accounts.Account[1] IN ['4', '5', '6']);

SetBankIndCodes := ['BB', 'ON', 'FS', 'FC'];

IsBank := Person.Accounts.IndustryCode IN SetBankIndCodes;

IsBankCard := IsBank AND IsRevolv;

AvgBal := AVE(Person.Accounts(isBankCard),Balance);
TotBal := SUM(Person.Accounts(isBankCard),Balance);
AvgHC  := AVE(Person.Accounts(isBankCard),HighCredit);

R1 := RECORD
  person.state;
  person.gender;
  Number          := COUNT(GROUP);
  AverageBal      := AVE(GROUP,AvgBal);
  AverageTotalBal := AVE(GROUP,TotBal);
  AverageHC       := AVE(GROUP,AvgHC);
END;

T1 := TABLE(person, R1,  state, gender);

OUTPUT(T1);
```

Um Exemplo Estatístico

O exemplo a seguir demonstra as funções VARIANCE, COVARIANCE e CORRELATION para analisar pontos de grade. Ele também mostra a técnica de inserir a cross-tab em uma MACRO, acionando essa MACRO para gerar um resultado específico em um determinado dataset.

```
pointRec := { REAL x, REAL y };

analyze( ds ) := MACRO
  #uniquename(rec)
  %rec% := RECORD
```

```
    c     := COUNT(GROUP),
    sx    := SUM(GROUP, ds.x),
    sy    := SUM(GROUP, ds.y),
    sxx   := SUM(GROUP, ds.x * ds.x),
    sxy   := SUM(GROUP, ds.x * ds.y),
    syy   := SUM(GROUP, ds.y * ds.y),
    varx  := VARIANCE(GROUP, ds.x);
    vary  := VARIANCE(GROUP, ds.y);
    varxy := COVARIANCE(GROUP, ds.x, ds.y);
    rc    := CORRELATION(GROUP, ds.x, ds.y) ;
  END;
  #uniquename(stats)
  %stats% := TABLE(ds,%rec% );

  OUTPUT(%stats%);
  OUTPUT(%stats%, { varx - (sxx-sx*sx/c)/c,
                    vary - (syy-sy*sy/c)/c,
                    varxy - (sxy-sx*sy/c)/c,
                    rc - (varxy/SQRT(varx*vary)) });
  OUTPUT(%stats%, { 'bestFit: y='+(STRING)((sy-sx*varxy/varx)/c)+'
                  + '+(STRING)(varxy/varx)+'x' });
ENDMACRO;

ds1 := DATASET([{1,1},{2,2},{3,3},{4,4},{5,5},{6,6}], pointRec);
ds2 := DATASET([{1.93896e+009, 2.04482e+009},
                {1.77971e+009, 8.54858e+008},
                {2.96181e+009, 1.24848e+009},
                {2.7744e+009,  1.26357e+009},
                {1.14416e+009, 4.3429e+008},
                {3.38728e+009, 1.30238e+009},
                {3.19538e+009, 1.71177e+009} ], pointRec);
ds3 := DATASET([{1, 1.00039},
                {2, 2.07702},
                {3, 2.86158},
                {4, 3.87114},
                {5, 5.12417},
                {6, 6.20283} ], pointRec);

analyze(ds1);
analyze(ds2);
analyze(ds3);
```

Uso Eficiente do Tipo de Valor

A arquitetura das estruturas de dados é uma arte que pode fazer uma grande diferença nos requisitos finais de desempenho e de armazenamento de dados. Apesar dos amplos recursos disponíveis nos clusters, pode ser importante economizar um byte aqui e alguns ali – mesmo em um sistema de processamento paralelo massivo de big data, os recursos não são infinitos.

Seleção de tipo de dados numérico

A escolha do tipo correto a ser usado em dados numéricos depende se os valores são números inteiros ou se eles contêm partes fracionárias (dados de ponto flutuante).

Dados Integer

Ao trabalhar com dados inteiros, você sempre deve especificar o tamanho exato de INTEGERn ou UNSIGNEDn necessários para manter o maior número possível para esse determinado campo. Isso melhorará o desempenho de execução e a eficiência do compilador, uma vez que o tipo de dados inteiro padrão é INTEGER8 (que também é o padrão para atributos com expressões inteiras).

A tabela a seguir define os valores maiores para cada tipo determinado:

```
Type       Signed                      Unsigned
INTEGER1   -128 to 127                 0 to 255
INTEGER2   -32,768 to 32,767           0 to 65,535
INTEGER3   -8,388,608 to 8,388,607     0 to 16,777,215
INTEGER4   -2,147,483,648              0 to 4,294,967,295
             to 2,147,483,647
INTEGER5   -549,755,813,888            0 to 1,099,511,627,775
             to 549,755,813,887
INTEGER6   -140,737,488,355,328        0 to 281,474,976,710,655
             to 140,737,488,355,327
INTEGER7   36,028,797,018,963,968      0 to 72,057,594,037,927,935
             to 36,028,797,018,963,967
INTEGER8   -9,223,372,036,854,775,808  0 to 18,446,744,073,709,551,615
             to 9,223,372,036,854,775,807
```

Por exemplo, se você possui dados "externos" onde um campo inteiro de 4 bytes contém valores que variam de zero (0) até noventa e nove (99), então faz sentido transferir esses dados para um campo UNSIGNED1. Isso economiza três bytes por segundo, algo que, se o dataset for muito grande (por exemplo, com 10 bilhões de registros), é possível gerar uma economia considerável nos requisitos de armazenamento de disco.

Uma vantagem que o ECL tem em relação à outras linguagens é a riqueza de seus tipos de números inteiros Ao permitir que você selecione o número exato de bytes (na faixa de um a oito), é possível adaptar seus requisitos de armazenamento para a faixa exata dos valores que precisam ser armazenados sem desperdiçar mais bytes.

Note que o uso dos formatos BIG_ENDIAN de todos os tipos inteiros é limitado para definir dados conforme são recebidos e retornam para a "fonte externa" de onde vieram – todos os dados inteiros usados internamente precisam estar no formato LITTLE_ENDIAN. O formato BIG_ENDIAN foi projetado especificamente para estabelecer a interface apenas com fontes de dados externas.

Dados de Pontos Flutuante

Ao usar os tipos de pontos flutuante, é necessário sempre especificar o tamanho exato do REALn necessário para manter o maior (e/ou menor) número possível para esse determinado campo. Isso melhorará o desempenho de execução e a eficiência do compilador, uma vez que o REAL retorna para o padrão REAL8 (oito bytes), exceto se for de outra forma especificado. Os valores REAL são armazenados internamente no formato de ponto flutuante IEEE assinado; REAL4 é o formato de 32 bits e REAL8 é o formato de 64 bits.

A tabela a seguir define o número de dígitos significativos de precisão e os maiores e menores valores que podem ser representados como valores REAL (ponto flutuante):

```
Type    Significant Digits     Largest Value     Smallest Value
REAL4      7 (9999999)         3.402823e+038      1.175494e-038
REAL8   15 (999999999999999)   1.797693e+308      2.225074e-308
```

Se precisar de mais de quinze dígitos significativos em seus cálculos, então é necessário considerar o tipo DECIMAL. Se todos os componentes de uma expressão forem do tipo DECIMAL, o resultado será calculado usando bibliotecas de matemática BCD (realizando a matemática de pontos flutuantes em base 10 em vez de base 2). Isso possibilita alcançar uma precisão de até trinta e dois dígitos, caso seja necessário. Ao usar a matemática em base 10, também é possível eliminar problemas de arredondamento que são comuns à matemática de pontos flutuantes.

Seleção do Tipo Dado String

Decidir qual dos vários tipos de dados de string será usado pode ser um processo complexo, já que há diversas escolhas: STRING, QSTRING, VARSTRING, UNICODE, e VARUNICODE. As escolhas óbvias ficam entre os vários tipos de STRING e UNICODE. Você precisa usar UNICODE e/ou VARUNICODE apenas se estiver lidando de fato com dados Unicode. Se for este o caso, a seleção é simples. No entanto, decidir exatamente que tipo de string usar pode ser mais desafiador.

STRING vs. VARSTRING

Dados que entram e saem das "fontes externas" podem conter strings terminadas em nulos. Se for este o caso, é necessário usar VARSTRING para definir esses campos no arquivo de dados de ingestão/saída. No entanto, programadores com muita experiência em C/C++ são propensos a usar VARSTRING para tudo, acreditando que essa opção será a mais eficiente – mas essa crença é infundada.

Não há nenhuma vantagem inerente em usar VARSTRING em vez de STRING no sistema. STRING é o tipo de dados interno da string, sendo, portanto, o tipo mais eficiente a ser usado. O tipo VARSTRING foi projetado especificamente para estabelecer a interface com fontes de dados externas, embora também possa ser usado dentro do sistema.

Isso se aplica da mesma forma ao escolher entre usar UNICODE ou VARUNICODE.

STRING vs. QSTRING

Dependendo do uso que você fará dos dados, pode ou não ser importante reter a caixa original dos caracteres. Consequentemente, a diferenciação entre maiúsculas e minúsculas, a armazenagem dos dados de string em caixa alta é perfeitamente adequada, e o tipo QSTRING é sua escolha lógica em vez do STRING. No entanto, se você PRECISA manter os dados com diferenciação entre maiúsculas e minúsculas, o tipo STRING é a única escolha a ser feita.

A vantagem de QSTRING sobre STRING é uma taxa de compressão "instantânea" de 25% dos dados, já que os caracteres dos dados QSTRING são representados por seis bits cada em vez de oito. Isso é feito ao armazenar os dados em caixa alta e permitindo apenas caracteres alfanuméricos e um pequeno grupo de caracteres especiais (! " # $ % & ' ( ) * + , - . / ; < = > ? @ [\ ] ^ _).

Para strings menores que quatro bytes, não há vantagem em usar o QSTRING sobre o STRING, já que os campos ainda precisam estar alinhados em limites de bytes. Por isso, o menor QSTRING que deve ser usado é um QSTRING4 (quatro caracteres armazenados em três bytes em vez de quatro).

Strings de Largura Fixa vs. Largura Variável

Um campo de string ou parâmetro pode ser definido em um comprimento específico ao anexar o número de caracteres ao nome do tipo (como STRING20 para uma string de 20 caracteres). Também pode ser definido como comprimento variável quando o comprimento não foi definido (como STRING para uma string de comprimento variável).

Os campos de string ou parâmetros conhecidos por sempre terem um tamanho específico devem ser designados no tamanho exato necessário. Isso melhorará a eficiência e o desempenho, permitindo que o compilador seja otimizado para a string de tamanho específico e não sofra com a sobrecarga de calcular dinamicamente o comprimento variável no tempo de execução. O tipo de valor de comprimento variável (STRING, QSTRING, ou UNICODE) só deve ser usado quando o comprimento da string for variável ou desconhecido.

A função LENGTH pode ser usada para determinar o comprimento de uma string de comprimento variável especificada como um parâmetro para uma função. Uma string especificada para uma função na qual o parâmetro foi declarado como uma STRING20 sempre terá um comprimento de 20, independentemente de seu conteúdo. Por exemplo, uma STRING20 que contém 'ABC' terá um comprimento de 20, e não de 3 (exceto, é claro, se você incluir a função TRIM na expressão). Uma string declarada como STRING de comprimento variável e que contém 'ABC' terá um comprimento de 3.

```
STRING20 CityName := 'Orlando';  // LENGTH(CityName) is 20
STRING   CityName := 'Orlando';  // LENGTH(CityName) is 7
```

Tipos de Dados Definidos pelo Usuário

Há várias maneiras de definir seus próprios tipos de dados no ECL. As estruturas RECORD e TYPE são as mais comuns.

Estrutura RECORD

A estrutura RECORD pode ser vinculada a uma *struct* nas linguagens C/C++. Ela define um grupo de campos relacionados que são os campos de um recordset, esteja esse recordset em disco ou em uma TABLE temporária, ou seja, resultado de qualquer operação usando uma função TRANSFORM.

A estrutura RECORD é um tipo de dado definido por usuário porque, depois de definida como um atributo, é possível usar esse atributo como:

* o tipo de dados para parâmetros especificados para funções TRANSFORM

* o tipo de dados para um "campo" em outra estrutura RECORD (estruturas aninhadas)

* a estrutura de um campo DATASET secundário aninhado em outra estrutura RECORD

Aqui está um exemplo que mostra todos os três usos (contidos no arquivo RecStruct.ECL):

```
IMPORT ProgrammersGuide.DeclareData AS ProgGuide;

Layout_Person := RECORD
  UNSIGNED1 PersonID;
  STRING15  FirstName;
  STRING25  LastName;
END;
Person := DATASET([{1,'Fred','Smith'},
                   {2,'Joe','Blow'},
                   {3,'Jane','Smith'}],Layout_Person);

Layout_Accounts := RECORD
  STRING10  Account;
  UNSIGNED4 Balance;
END;
Layout_Accounts_Link := RECORD
  UNSIGNED1 PersonID;
  Layout_Accounts;               //nested RECORD structure
END;

Accounts := DATASET([{1,'45621234',452},
```

```
                    {1,'55621234',5000},
                    {2,'45629876',4215},
                    {3,'45628734',8525}],Layout_Accounts_Link);

Layout_Combined := RECORD
  Layout_Person;
  DATASET(Layout_Accounts) Accounts;    //nested child DATASET
END;

P_recs := PROJECT(Person, TRANSFORM(Layout_Combined,SELF := LEFT; SELF := []));

Layout_Combined CombineRecs(Layout_Combined L,
                            Layout_Accounts_Link R) := TRANSFORM
  SELF.Accounts := L.Accounts + ROW({R.Account,R.Balance}, Layout_Accounts);
  SELF := L;
END;                            //input and output types

NestedPeopleAccts := DENORMALIZE(P_recs,
                                 Accounts,
                                 LEFT.personid=RIGHT.personid,
                                 CombineRecs(LEFT,RIGHT));

OUTPUT(NestedPeopleAccts);
```

O Layout_Accounts_Link contém Layout_Accounts. Nenhum nome de campo foi definido, o que significa que ele simplesmente herda todos os campos nessa estrutura, na medida em que são definidos, e esses campos herdados são referenciados como se estivessem declarados de forma explícita na estrutura RECORD Layout_Accounts_Link, como por exemplo:

```
x := Accounts.Balance;
```

No entanto, se um nome tiver sido definido, então ele definiria uma estrutura aninhada e os campos nessa estrutura aninhada precisariam ser referenciados usando o nome da estrutura aninhada como parte do qualificador, da seguinte maneira:

```
//Assuming the definition was this:
Layout_Accounts_Link := RECORD
  UNSIGNED1          PersonID;
  Layout_Accounts    AcctStruct;      //nested RECORD with name
END;
  //then the field reference would have to be this:
x := Accounts.AcctStruct.Balance;
```

O atributo da estrutura RECORD Layout_Accounts define a estrutura do campo DATASET secundário no Layout_Combined. A estrutura RECORD Layout_Combined é então usada como entrada e resultado LEFT para a função TRANSFORM CombineRecs.

Estrutura TYPE

A estrutura TYPE é obviamente um tipo definido por usuário, uma vez que você define um tipo de dados que ainda não é compatível com a linguagem ECL. Sua finalidade é permitir a importação de dados em qualquer formato recebido, seu processamento em um dos formatos internos seguido da regravação dos dados no formato original em disco.

Ela funciona pela definição de funções de retorno de chamada específicas dentro da estrutura TYPE (LOAD, STORE, etc.) que o sistema utilizará para ler e gravar dados do e para o disco. A função de retorno LOAD lê os dados do disco e define seu tipo interno na medida em que você trabalhar com eles na forma de tipo de dados de retorno da função LOAD gravada.

```
GetXLen(DATA x,UNSIGNED len) := TRANSFER(((DATA4)(x[1..len])),UNSIGNED4);
xstring(UNSIGNED len) := TYPE
  EXPORT INTEGER PHYSICALLENGTH(DATA x) := GetXLen(x,len) + len;
```

```
  EXPORT STRING LOAD(DATA x) := (STRING)x[(len+1)..GetXLen(x,len) + len];
  EXPORT DATA STORE(STRING x):= TRANSFER(LENGTH(x),DATA4)[1..len] + (DATA)x;
END;

pstr    := xstring(1);    // typedef for user defined type
pppstr  := xstring(3);
nameStr := STRING20;     // typedef of a system type

namesRecord := RECORD
  pstr    surname;
  nameStr forename;
  pppStr  addr;
END;

ds := DATASET([{'TAYLOR','RICHARD','123 MAIN'},
               {'HALLIDAY','GAVIN','456 HIGH ST'}],
               {nameStr sur,nameStr fore, nameStr addr});

namesRecord  MoveData(ds L) := TRANSFORM
  SELF.surname  := L.sur;
  SELF.forename := L.fore;
  SELF.addr     := L.addr;
END;

out := PROJECT(ds,MoveData(LEFT));
OUTPUT(out);
```

Esse exemplo define um tipo de dados de "string Pascal", com o comprimento inicial armazenado como um a quatro bytes anexados aos dados.

Atributos TypeDef

O atributo TypeDef é outro tipo óbvio definido por usuário, uma vez que você define uma instância específica de um tipo de dados que já é compatível na linguagem ECL como um novo nome, seja para fins de conveniência de manutenção ou por motivos de legibilidade de código. O exemplo acima também demonstra o uso de atributos Type-Def.

Utilizando a Função GROUP

A função GROUP oferece funcionalidades importantes ao processar datasets de tamanho significativo. O conceito básico é que a função GROUP dividirá o dataset em um número de subconjuntos menores, mas o dataset que passou pela função GROUP ainda será tratado como uma entidade única no seu código ECL.

As operações em um dataset GROUPed (AGRUPADOS) são realizadas automaticamente e separadamente em cada subconjunto. Consequentemente, uma operação em um dataset GROUPed (AGRUPADO) aparecerá no código ECL como uma operação única, porém, a operação será realizada internamente pela execução em série da mesma operação em relação a cada subconjunto por vez. A vantagem dessa abordagem é que cada operação individual é muito menor e tem maior probabilidade de ser realizada sem vazamento no disco, o que significa que o tempo total para realizar todas as operações separadas normalmente será inferior do que realizar a mesma operação em todo o dataset (às vezes a diferença é bastante significativa).

GROUP vs. SORT

A função GROUP não classifica automaticamente os registros nos quais está operando – ela realizará o GROUP com base na ordem dos registros recebidos. Dessa forma, a classificação dos registros é normalmente realizada primeiro pelos campos nos quais você deseja aplicar a função GROUP (exceto em circunstâncias onde os campos GROUP são usados apenas para dividir uma operação única maior em diversas operações menores).

Para o conjunto de operações que usam as funções TRANSFORM (como ITERATE, PROJECT, ROLLUP, etc), operar em um dataset GROUPed onde a operação é realizada em cada fragmento (grupo) no grupo de registros, de forma independente, implica que o teste das condições de limite será diferente em comparação com um dataset que tenha passado pela função SORTed. Por exemplo, o código a seguir (contido no arquivo GROUPfunc.ECL) usa a função GROUP para classificar as contas pessoais com base na data de abertura e saldo. A conta com a data de abertura mais recente tem a maior classificação (se houver múltiplas contas abertas no mesmo dia, será usada aquela com o maior saldo). Não há necessidade de verificação de limite na função TRANSFORM porque ITERATE é novamente reiniciada com cada pessoa, de forma que o valor do campo L.Ranking para cada novo grupo de pessoas seja igual a zero (0).

```
IMPORT $;

accounts := $.DeclareData.Accounts;

rec := RECORD
  accounts.PersonID;
  accounts.Account;
  accounts.opendate;
  accounts.balance;
  UNSIGNED1 Ranking := 0;
END;

tbl := TABLE(accounts,rec);

rec RankGrpAccts(rec L, rec R) := TRANSFORM
  SELF.Ranking := L.Ranking + 1;
  SELF := R;
END;
GrpRecs  := SORT(GROUP(SORT(tbl,PersonID),PersonID),-Opendate,-Balance);
i1 := ITERATE(GrpRecs,RankGrpAccts(LEFT,RIGHT));
OUTPUT(i1);
```

O código a seguir usa apenas SORT para obter a mesma ordem de registro do código anterior. Observe o código de verificação de limites na função TRANSFORM. Isso é mandatório, uma vez que ITERATE realizará uma operação única em todo o dataset:

```
rec RankSrtAccts(rec L, rec R) := TRANSFORM
```

```
  SELF.Ranking := IF(L.PersonID = R.PersonID,L.Ranking + 1, 1);
  SELF := R;
END;
SortRecs := SORT(tbl,PersonID,-Opendate,-Balance);
i2 := ITERATE(SortRecs,RankSrtAccts(LEFT,RIGHT));
OUTPUT(i2);
```

A verificação dos diferentes limites em cada é exigida pela fragmentação criada pela função GROUP. ITERATE opera separadamente em cada fragmento no primeiro exemplo, e opera em todo o recordset no segundo.

Considerações sobre desempenho

Há também uma enorme vantagem em relação ao desempenho no uso da função GROUP. Por exemplo, SORT é uma operação de *n log n* exemplo, de forma que dividir os conjuntos grandes de registros em conjuntos menores pode melhorar significativamente o tempo necessário para realizar a operação de classificação.

Vamos supor que um dataset contenha 1 bilhão de registros de 1.000 bytes (1.000.000.000) e está sendo executado em um supercomputador de 100 nós. Supondo também que os dados estão igualmente distribuídos, temos então 10 milhões de registros por nó ocupando 1 gigabyte de memória em cada nó. Suponha que você precise classificar os dados por três campos: personID (ID pessoal), opendate (data de abertura) e balance (saldo). Há três maneiras possíveis de se fazer isso: SORT global, SORT local distribuída, ou SORT local distribuída que passou pela função GROUPed .

Aqui está um exemplo que demonstra todos os três métodos (contidos no arquivo GROUPfunc.ECL):

```
bf := NORMALIZE(accounts,
                CLUSTERSIZE * 2,
                TRANSFORM(RECORDOF(ProgGuide.Accounts),
                          SELF := LEFT));
ds0 := DISTRIBUTE(bf,RANDOM()) : PERSIST('~PROGGUIDE::PERSIST::TestGroupSort');
ds1 := DISTRIBUTE(ds,HASH32(personid));

// do a global sort
s1 := SORT(ds0,personid,opendate,-balance);
a  := OUTPUT(s1,,'~PROGGUIDE::EXAMPLEDATA::TestGroupSort1',OVERWRITE);

// do a distributed local sort
s3  := SORT(ds1,personid,opendate,-balance,LOCAL);
b   := OUTPUT(s3,,'~PROGGUIDE::EXAMPLEDATA::TestGroupSort2',OVERWRITE);

// do a grouped local sort
s4 := SORT(ds1,personid,LOCAL);
g2 := GROUP(s4,personid,LOCAL);
s5 := SORT(g2,opendate,-balance);
c  := OUTPUT(s5,,'~PROGGUIDE::EXAMPLEDATA::TestGroupSort3',OVERWRITE);
SEQUENTIAL(a,b,c);
```

Os conjuntos resultantes para todas essas operações SORT são idênticos. No entanto, o tempo necessário para produzi-los não é. O exemplo acima opera apenas em 10 milhões de registros de 46 bytes por nó, não em 1 bilhão de registros de 1.000 bytes anteriormente mencionados, mas certamente ilustra as técnicas.

Para o exemplo hipotético de um bilhão de registros, o desempenho da função Sort global é calculado pela fórmula: 1 bilhão de vezes o log de 1 bilhão (9) resultando em uma métrica de desempenho de 9 bilhões. O desempenho da função Sort local distribuída é calculado pela fórmula: 10 milhões de vezes o log de 10 milhões (7) resultando em uma métrica de desempenho de 70 milhões. Supondo que a operação GROUP tenha criado 1.000 subgrupos em cada nó, o desempenho da função Sort local que passou pela função GROUP é calculado pela fórmula: 1.000 vezes (10.000 vezes o log de 10.000 (4)) resultando em uma métrica de desempenho de 40 milhões.

Os próprios números de métricas de desempenho não são importantes, mas suas proporções indicam a diferença esperada em relação ao desempenho entre os métodos SORT. Isso significa que a função SORT local distribuída será cerca de 128 vezes mais rápida do que a SORT global (9 bilhões / 70 milhões); a SORT que passou pela função GROUP

será cerca de 225 vezes mais rápida do que a SORT global (9 bilhões / 40 bilhões), e a SORT que passou pela função GROUP será aproximadamente 1,75 vezes mais rápida que a SORT local distribuída (70 milhões / 40 milhões).

ECL Automatizado

Após ter definido os processos ECL padrão que você precisará realizar regularmente, é possível começar a automatizar esses processos. A automatização de processos elimina a necessidade de memorizar a ordem ou até mesmo a periodicidade.

Uma forma de automação normalmente envolve iniciar MACROs com a aplicação ECLPlus. Ao usar MACROs, é possível ter processos padrão que operam em diferentes entradas por vez, mas que produzem o mesmo resultado. Uma vez que o ECLPlus é uma aplicação de linha de comando, seu uso pode ser iniciado automaticamente de várias formas diferentes – arquivos em lote no DOS, de dentro de outra aplicação, ou...

Aqui está um exemplo. Esta MACRO (contida no arquivo DeclareData.ECL) usa dois parâmetros: o nome de um arquivo e o nome de um campo nesse arquivo para produzir uma contagem dos valores únicos no campo e um relatório de tabela de referência cruzada do número de instâncias de cada valor.

```
EXPORT MAC_CountFieldValues(infile,infield) := MACRO
  // Create the count of unique values in the infield
  COUNT(DEDUP(TABLE(infile,{infile.infield}),infield,ALL));

  // Create the crosstab report
  #UNIQUENAME(r_macro)
  %r_macro% := RECORD
    infile.infield;
    INTEGER cnt := COUNT(GROUP);
  END;
  #UNIQUENAME(y_macro)
  %y_macro% := TABLE(infile,%r_macro%,infield,FEW);
  OUTPUT(CHOOSEN(%y_macro%,50000));
ENDMACRO;
```

Ao usar #UNIQUENAME para gerar todos os nomes de atributos, essa MACRO pode ser usada múltiplas vezes na mesma tarefa. A MACRO pode ser testada no programa ECL IDE executando uma consulta como essa na janela do compilador ECL:

```
IMPORT ProgrammersGuide AS PG;
PG.DeclareData.MAC_CountFieldValues(PG.DeclareData.Person.file,gender);
```

Após ter testado completamente a MACRO e estar certo de que ela funciona corretamente, é possível automatizar o processo usando o ECLPlus.

Instale o programa ECLPlus em seu próprio diretório no mesmo PC que executa o ECL IDE, e crie um arquivo ECLPLUS.INI na mesma pasta com as configurações corretas para acessar o seu cluster (consulte a seção *Linha de comando ECL* no *PDF Ferramentas de cliente*). Em seguida, você pode abrir uma janela de prompt de comando e executar a mesma consulta da linha de comando da seguinte maneira:

```
C:\eclplus>eclplus
ecl=$ProgGuide.MAC_CountFieldValues(ProgrammersGuide.DeclareData.Person.File,gender)
```

Observe que você está usando a opção de linha de comando *ecl=* e não a opção *$Module.Attribute* . Essa é a maneira "correta" de expandir uma MACRO e executá-la pelo ECLplus. A opção *$Module.Attribute* é usada apenas para executar consultas da janela do compilador ECL que foram salvas como atributos no repositório (Executável da janela do compilador – código BWR) e não funciona com as MACROs.

Quando uma MACRO é expandida e executada, o resultado observado é parecido com esse em sua janela de prompt de comando:

```
Workunit W20070118-145647 submitted
[Result 1]
Result_1
```

```
2
[Result_2]
gender     cnt
 F        500000
 M        500000
```

Você pode redirecionar esse resultado para um arquivo usando a opção *output="filename"* na linha de comando da seguinte maneira:

```
C:\eclplus>eclplus ecl=$ProgGuide.MAC_CountFieldValues
( ProgrammersGuide.DeclareData.Person.File, gender) output="MyFile.txt"
```

Para arquivos de resultados maiores, você deve deixar a ação OUTPUT em seu código ECL gravar o conjunto de resultado em disco no supercomputador, e depois consolidar os dados aos nós em sua zona de entrada de arquivos (é possível usar a função File.Despray da biblioteca padrão para fazer isso a partir do seu código ECL).

Utilizando Arquivos Texto

Outra opção de automação é gerar um arquivo de texto que contenha o código ECL a ser executado e depois executar esse código a partir da linha de comando.

Por exemplo, você pode criar um arquivo contendo o seguinte:

```
IMPORT ProgrammersGuide AS PG;
PG.DeclareData.MAC_CountFieldValues(PG.DeclareData.Person.file,gender);
PG.DeclareData.MAC_CountFieldValues(PG.DeclareData.person.File,state)
```

Essas duas chamadas MACRO vão gerar o relatório de referência de tabela cruzada e contagem de ordinalidade de campo para os dois campos no mesmo arquivo. Você pode então executá-los da seguinte maneira (onde "test.ECL" é o nome do arquivo que você criou):

```
C:\eclplus>eclplus @test.ecl
```

Isso gerará resultados similares àqueles acima.

A vantagem desse método é a capacidade de incluir qualquer código de "configuração" do código ECL no arquivo antes dos acionamentos da MACRO, da seguinte forma (contido no arquivo RunText.ECL):

```
IMPORT ProgrammersGuide AS PG;
MyRec := RECORD
  STRING1 value1;
  STRING1 value2;
END;
D := DATASET([{'A','B'},
              {'B','C'},
              {'A','D'},
              {'B','B'},
              {'A','C'},
              {'B','D'},
              {'A','B'},
              {'C','C'},
              {'C','D'},
              {'A','A'}],MyRec);

PG.DeclareData.MAC_CountFieldValues(D,Value1)
PG.DeclareData.MAC_CountFieldValues(D,Value2)
```

Dessa forma, você consegue obter um resultado parecido com esse:

```
C:\eclplus>eclplus @test.ecl
Workunit W20070118-145647 submitted
[Result 1]
```

```
result_1
3
[Result 2]
value1  cnt
C        2
A        5
B        3
[Result 3]
result_3
4
[Result 4]
value2  cnt
D        3
C        3
A        1
B        3
```

Fica a seu critério decidir como esse arquivo de texto será usado. Para automatizar totalmente o processo, você pode programar uma aplicação daemon que vigie o diretório (como a zona de entrada de arquivos do ambiente HPCC) para detectar novos arquivos descartados (por quaisquer meios) e gerar o arquivo de código ECL adequado para processar o novo arquivo em algum modo padrão (normalmente usando acionamentos MACRO), e depois executá-lo a partir da linha de comando ECLplus como descrito acima. As possibilidades são infinitas.

"Falha" do Job

Às vezes, os jobs falham. E, às vezes, esse comportamento ocorre por concepção.

Por exemplo, tentar enviar um resultado inteiro gerado de volta ao programa ECL IDE quando tal resultado contém mais de 10 megabytes de dados levará a uma “falha” do job com o erro "Dataset muito grande para ser enviado à workunit (limite de 10 megabytes)." Essa “falha” do job é calculada naquela parte do sistema (e você pode redefinir este limite para cada workunit usando #OPTION), porque no momento em que estiver gravando toda essa quantidade de dados, a gravação deve ser feita em um arquivo para despray (consolidação de dados aos nós). Caso contrário, a armazenagem de dados em seu sistema será rapidamente preenchida.

Outros exemplos deste tipo de “falha” calculada pelo sistema: ao exceder os limites de SKEW ou de qualquer outro tempo de execução. Alguns desses limites podem ser redefinidos (mas, geralmente, essa NÃO é a melhor solução). Além disso, a “falha” calculada é um sinal de que há algo inerentemente errado com o job e de que, talvez, a abordagem usada precise ser repensada.

Entre em contato com a equipe de Suporte técnico ao surgir qualquer problema. Nós ajudaremos você a criar uma estratégia para fazer o que precisa ser feito sem gerar estas “falhas” calculadas pelo sistema.

Non-Random RANDOM

Haverá situações em que você precisará de um número aleatório e, após obtê-lo, você quer que aquele valor permaneça o mesmo durante toda a duração da workunit. Por exemplo, o "problema" com este código é que ele terá três OUTPUT de valores distintos (este código está em NonRandomRandom.ECL):

```
INTEGER1 Rand1 := (RANDOM() % 25) + 1;
OUTPUT(Rand1);
OUTPUT(Rand1);
OUTPUT(Rand1);
```

Para que o valor "random" (aleatório) continue sendo o mesmo durante a workunit, basta adicionar o serviço de fluxo de trabalho STORED à definição do atributo, como exemplificamos aqui (este código também consta no NonRandomRandom.ECL):

```
INTEGER1 Rand2 := (RANDOM() % 25) + 1 : STORED('MyRandomValue');
OUTPUT(Rand2);
OUTPUT(Rand2);
OUTPUT(Rand2);
```

Isso fará com que o valor "random" (aleatório) seja calculado uma vez e seja usado durante todo o restante da workunit.

O serviço do fluxo de trabalho GLOBAL faria a mesma coisa, mas usar STORED é mais vantajoso porque o valor "random" (aleatório) usado para a workunit será exibido na página do ECL Watch daquela tarefa. Isso permite um melhor debug do código, pois você verá qual valor "random" (aleatório) exato foi usado na tarefa.

Trabalhando com Dados XML

Os dados nem sempre são recebidos em arquivos simples, de comprimento fixo, fáceis de trabalhar e perfeitos: eles podem vir de várias formas. Um formato que é cada vez mais usado hoje em dia é o XML. ECL possui diversas opções para processar dados em XML – algumas óbvias e outras nem tanto.

OBSERVAÇÃO: XML a leitura e a análise podem consumir muita memória dependendo do uso. Em particular, se o XPATH especificado combinar com uma quantidade muito grande de dados, uma grande estrutura de dados será então fornecida para transformação. Dessa forma, quanto maior for a combinação, mais recursos serão consumidos por combinação. Por exemplo, se você possui um documento muito grande e combinar um elemento próximo da raiz que, virtualmente, engloba o documento todo, o documento inteiro será interpretado como uma estrutura referenciável que o ECL terá acesso.

Manipulação de Dados XML Simples

As opções XML no DATASET e OUTPUT permitem que você trabalhe facilmente com dados XML. Por exemplo, um arquivo XML que se parece com isso (esses dados foram gerados pelo código no GenData..ECL):

```
<?xml version=1.0 ...?>
<timezones>
<area>
  <code>
        215
  </code>
  <state>
        PA
  </state>
  <description>
        Pennsylvania (Philadelphia area)
  </description>
  <zone>
        Eastern Time Zone
  </zone>
</area>
<area>
  <code>
        216
  </code>
  <state>
        OH
  </state>
  <description>
        Ohio (Cleveland area)
  </description>
  <zone>
        Eastern Time Zone
  </zone>
</area>
</timezones>
```

Este arquivo pode ser declarado para uso em seu código ECL (uma vez que esse arquivo é declarado com o DATASET TimeZonesXML declarado na estrutura MODULE DeclareData) desta forma:

```
EXPORT TimeZonesXML :=
          DATASET('~PROGGUIDE::EXAMPLEDATA::XML_timezones',
                  {STRING code,
                   STRING state,
                   STRING description,
                   STRING timezone{XPATH('zone')}},
```

```
                  XML('timezones/area') )
```

Isso disponibiliza os dados contidos em cada tag XML no arquivo para uso em seu código ECL assim como qualquer dataset de arquivo simples. Os nomes de campo na estrutura RECORD (neste caso, embutidos na declaração DATASET) duplicam os nomes de tags no arquivo. O uso do modificador XPATH no campo timezone nos permite especificar que o campo vem da tag <zone>. Esse mecanismo nos permite nomear campos de forma diferente de seus nomes de tag.

Ao redefinir os campos como tipos STRING sem especificar seu comprimento, é possível ter a certeza de que todos os dados estão sendo obtidos – incluindo qualquer retorno de carro, alimentação de linha e tabulação no arquivo XML que estejam contidos dentro das tags de campo (conforme representadas neste arquivo). Esse OUTPUT simples mostra o resultado (esse e todos os exemplos de código neste artigo estão contidos no arquivo XMLcode..ECL).

```
IMPORT $;

ds := $.DeclareData.timezonesXML;

OUTPUT(ds);
```

Observe que o resultado exibido no programa ECL IDE contém colchetes nos dados – esses são retornos de carro, as alimentações de linha e as tabulações nos dados. É possível acabar com os erros de retorno de carro, alimentações de linha e tabulações simplesmente especificando os registros em uma operação PROJECT, da seguinte forma:

```
StripIt(STRING str) := REGEXREPLACE('[\r\n\t]',str,'$1');
RECORDOF(ds) DoStrip(ds L) := TRANSFORM
  SELF.code := StripIt(L.code);
  SELF.state := StripIt(L.state);
  SELF.description := StripIt(L.description);
  SELF.timezone := StripIt(L.timezone);
END;
StrippedRecs := PROJECT(ds,DoStrip(LEFT));
OUTPUT(StrippedRecs);
```

O uso da função REGEXREPLACE simplifica bastante o processo. Seu primeiro parâmetro é uma expressão regular Perl padrão que representa os caracteres que se busca: retorno de carro (\r), alimentação de linha (\n) e tabulação (\t).

É possível agora operar o recordset StrippedRecs (ou dataset ProgGuide.TimeZonesXML) assim como você faria com qualquer outro. Por exemplo, você pode querer simplesmente filtrar campos e registros desnecessários e gravar o resultado em um novo arquivo XML para ser especificado.

```
InterestingRecs := StrippedRecs((INTEGER)code BETWEEN 301 AND 303);
OUTPUT(InterestingRecs,{code,timezone},
       '~PROGGUIDE::EXAMPLEDATA::OUT::timezones300',
XML('area',HEADING('<?xml version=1.0 ...?>\n<timezones>\n','</timezones>')),OVERWRITE);
```

O arquivo XML resultante é mais ou menos assim:

```
<?xml version=1.0 ...?>
<timezones>
<area><code>301</code><zone>Eastern Time Zone</zone></area>
<area><code>302</code><zone>Eastern Time Zone</zone></area>
<area><code>303</code><zone>Mountain Time Zone</zone></area>
</timezones>
```

Manipulação de Dados XML Complexos

É possível criar resultados em XML mais complexos usando a opção CSV no OUTPUT em vez da opção XML. A opção XML produzirá apenas o estilo simples do XML mostrado acima. Porém, algumas aplicações exigem o uso de atributos XML dentro das tags. Este código demonstra como produzir esse formato:

```
CRLF := (STRING)x'0D0A';
```

```
OutRec := RECORD
  STRING Line;
END;
OutRec DoComplexXML(InterestingRecs L) := TRANSFORM
SELF.Line := '  <area code="' + L.code + '">' + CRLF +
             '    <zone>' + L.timezone + '</zone>' + CRLF +
             '  </area>';
END;
ComplexXML := PROJECT(InterestingRecs,DoComplexXML(LEFT));
OUTPUT(ComplexXML,,'~PROGGUIDE::EXAMPLEDATA::OUT::Complextimezones301',
CSV(HEADING('<?xml version=1.0 ...?>'+CRLF+'<timezones>'+CRLF,'</timezones>')),OVERWRITE);
```

A estrutura RECORD define um campo de resultado único para conter cada registro XML lógico criado com a função TRANSFORM. A operação PROJECT compila todos os registros de resultados individuais e, em seguida, a opção CSV na ação OUTPUT especifica os registros de cabeçalho e de rodapé do arquivo (neste caso, as tags do arquivo XML), exibindo o resultado mostrado aqui:

```
<?xml version=1.0 ...?>
<timezones>
  <area code="301">
    <zone>Eastern Time Zone</zone>
  </area>
  <area code="302">
    <zone>Eastern Time Zone</zone>
  </area>
  <area code="303">
    <zone>Mountain Time Zone</zone>
  </area>
</timezones>
```

Então, se o uso da opção CSV é o meio para OBTER OUTPUT formatos de dados XML complexos, como você pode acessar dados XML existentes de formato complexo e usar o ECL para trabalhar com eles?

A resposta está no uso da opção XPATH nas definições de campo na estrutura RECORD como abaixo:

```
NewTimeZones :=
 DATASET('~PROGGUIDE::EXAMPLEDATA::OUT::Complextimezones301',
         {STRING area {XPATH('<>')}},
         XML('timezones/area'));
```

A opção {XPATH('<>')} especificada basicamente "solicita tudo que está na tag XML, incluindo as próprias tags", de forma que você possa então usar o ECL para interpretar o texto e fazer o seu trabalho. Os registros de dados NewTimeZones se parecem com esse (já que incluem todos os retornos de carro/alimentação de linha) ao realizar uma operação OUTPUT simples e copiar o registro para um editor de texto:

```
<area code="301">
  <zone>Eastern Time Zone</zone>
</area>
```

Você pode então usar qualquer uma das funções de processamento de string no ECL ou as funções de Biblioteca de Serviços no StringLib ou UnicodeLib (consulte *Referência de biblioteca de serviços*) para trabalhar com o texto. No entanto, a ferramenta mais poderosa de análise de texto ECL é a função PARSE, que permite definir expressões regulares e/ou definições de atributos PATTERN do ECL para processar os dados.

Este exemplo usa a versão TRANSFORM de PARSE para obter dados XML:

```
{ds.code, ds.timezone} Xform(NewTimeZones L) := TRANSFORM
  SELF.code     := XMLTEXT('@code');
  SELF.timezone := XMLTEXT('zone');
END;
ParsedZones := PARSE(NewTimeZones,area,Xform(LEFT),XML('area'));
```

```
OUTPUT(ParsedZones);
```

Neste código, usamos o formato XML da PARSE e sua função associada XMLTEXT para analisar os dados da estrutura XML . O parâmetro para XMLTEXT é o XPATH para os dados nos quais estamos interessados (o principal subconjunto do padrão XPATH que o ECL suporta está documentado na Referência de Linguagem na discussão de estrutura RECORD).

Entrada com formatos XML complexos

Os dados XML podem vir em diversos formatos, e alguns deles utilizam "child dataset", de forma que uma determinada tag possa conter múltiplas instâncias de outras tags que contêm as próprias tags de campo individuais.

Aqui está um exemplo de uma estrutura com essa complexidade usando dados UCC. Um arquivamento individual pode conter uma ou mais transações que, por sua vez, podem conter múltiplos registros Debtor e SecuredParty:

```
<UCC>
  <Filing number='5200105'>
    <Transaction ID='5'>
  <StartDate>08/01/2001</StartDate>
  <LapseDate>08/01/2006</LapseDate>
  <FormType>UCC 1 FILING STATEMENT</FormType>
  <AmendType>NONE</AmendType>
  <AmendAction>NONE</AmendAction>
  <EnteredDate>08/02/2002</EnteredDate>
  <ReceivedDate>08/01/2002</ReceivedDate>
  <ApprovedDate>08/02/2002</ApprovedDate>
  <Debtor entityId='19'>
    <IsBusiness>true</IsBusiness>
    <OrgName><![CDATA[BOGUS LABORATORIES, INC.]]></OrgName>
    <Status>ACTIVE</Status>
    <Address1><![CDATA[334 SOUTH 900 WEST]]></Address1>
    <Address4><![CDATA[SALT LAKE CITY 45 84104]]></Address4>
    <City><![CDATA[SALT LAKE CITY]]></City>
    <State>UTAH</State>
    <Zip>84104</Zip>
    <OrgType>CORP</OrgType>
    <OrgJurisdiction><![CDATA[SALT LAKE CITY]]></OrgJurisdiction>
    <OrgID>654245-0142</OrgID>
    <EnteredDate>08/02/2002</EnteredDate>
  </Debtor>
  <Debtor entityId='7'>
    <IsBusiness>false</IsBusiness>
    <FirstName><![CDATA[FRED]]></FirstName>
    <LastName><![CDATA[JONES]]></LastName>
    <Status>ACTIVE</Status>
    <Address1><![CDATA[1038 E. 900 N.]]></Address1>
    <Address4><![CDATA[OGDEN 45 84404]]></Address4>
    <City><![CDATA[OGDEN]]></City>
    <State>UTAH</State>
    <Zip>84404</Zip>
    <OrgType>NONE</OrgType>
    <EnteredDate>08/02/2002</EnteredDate>
  </Debtor>
  <SecuredParty entityId='20'>
    <IsBusiness>true</IsBusiness>
    <OrgName><![CDATA[WELLS FARGO BANK]]></OrgName>
    <Status>ACTIVE</Status>
    <Address1><![CDATA[ATTN: LOAN OPERATIONS CENTER]]></Address1>
    <Address3><![CDATA[P.O. BOX 9120]]></Address3>
    <Address4><![CDATA[BOISE 13 83707-2203]]></Address4>
    <City><![CDATA[BOISE]]></City>
    <State>IDAHO</State>
    <Zip>83707-2203</Zip>
```

```
    <Status>ACTIVE</Status>
    <EnteredDate>08/02/2002</EnteredDate>
  </SecuredParty>
  <Collateral>
    <Action>ADD</Action>
    <Description><![CDATA[ALL ACCOUNTS]]></Description>
    <EffectiveDate>08/01/2002</EffectiveDate>
  </Collateral>
    </Transaction>
    <Transaction ID='375799'>
  <StartDate>08/01/2002</StartDate>
  <LapseDate>08/01/2006</LapseDate>
  <FormType>UCC 3 AMENDMENT</FormType>
  <AmendType>TERMINATION BY DEBTOR</AmendType>
  <AmendAction>NONE</AmendAction>
  <EnteredDate>02/23/2004</EnteredDate>
  <ReceivedDate>02/18/2004</ReceivedDate>
  <ApprovedDate>02/23/2004</ApprovedDate>
    </Transaction>
  </Filing>
</UCC>
```

O segredo para trabalhar com esse tipo de dados XML complexos são as estruturas RECORD que definem o layout dos dados XML .

```
CollateralRec := RECORD
  STRING Action        {XPATH('Action')};
  STRING Description   {XPATH('Description')};
  STRING EffectiveDate {XPATH('EffectiveDate')};
END;


PartyRec := RECORD
  STRING   PartyID         {XPATH('@entityId')};
  STRING   IsBusiness      {XPATH('IsBusiness')};
  STRING   OrgName         {XPATH('OrgName')};
  STRING   FirstName       {XPATH('FirstName')};
  STRING   LastName        {XPATH('LastName')};
  STRING   Status          {XPATH('Status[1]')};
  STRING   Address1        {XPATH('Address1')};
  STRING   Address2        {XPATH('Address2')};
  STRING   Address3        {XPATH('Address3')};
  STRING   Address4        {XPATH('Address4')};
  STRING   City            {XPATH('City')};
  STRING   State           {XPATH('State')};
  STRING   Zip             {XPATH('Zip')};
  STRING   OrgType         {XPATH('OrgType')};
  STRING   OrgJurisdiction {XPATH('OrgJurisdiction')};
  STRING   OrgID           {XPATH('OrgID')};
  STRING10 EnteredDate     {XPATH('EnteredDate')};
END;

TransactionRec := RECORD
  STRING            TransactionID  {XPATH('@ID')};
  STRING10          StartDate      {XPATH('StartDate')};
  STRING10          LapseDate      {XPATH('LapseDate')};
  STRING            FormType       {XPATH('FormType')};
  STRING            AmendType      {XPATH('AmendType')};
  STRING            AmendAction    {XPATH('AmendAction')};
  STRING10          EnteredDate    {XPATH('EnteredDate')};
  STRING10          ReceivedDate   {XPATH('ReceivedDate')};
  STRING10          ApprovedDate   {XPATH('ApprovedDate')};
  DATASET(PartyRec) Debtors        {XPATH('Debtor')};
  DATASET(PartyRec) SecuredParties {XPATH('SecuredParty')};
  CollateralRec     Collateral     {XPATH('Collateral')}
```

```
END;

UCC_Rec := RECORD
  STRING                  FilingNumber {XPATH('@number')};
  DATASET(TransactionRec) Transactions {XPATH('Transaction')};
END;
UCC := DATASET('~PROGGUIDE::EXAMPLEDATA::XML_UCC',UCC_Rec,XML('UCC/Filing'));
```

Começando por baixo, essas estruturas RECORD são combinadas para criar o layout UCC_Rec final que define todo o formato desses dados XML .

A opção XML na declaração DATASET final especifica o XPATH para a tag de registro (arquivamento), e depois as definições de "campo" do DATASET secundário nas estruturas RECORD processam as questões de múltipla instância. Uma vez que o ECL faz distinção entre maiúsculas e minúsculas, assim como a sintaxe XML , é necessário usar o XPATH para definir todas as tags de campo. A estrutura RECORD PartyRec funciona com ambos os campos DATASET secundários Debtors e SecuredParties, já que ambos contêm as mesmas tags e informações.

Depois de definir o layout, como se extrai os dados para uma estrutura relacional normalizada para que se possa trabalhar com ela no supercomputador? NORMALIZE é a resposta. NORMALIZE precisa saber quantas vezes ela deve acionar sua TRANSFORM, para que a função TABLE possa ser usada para obter as contagens, da seguinte maneira:

```
XactTbl := TABLE(UCC,{INTEGER XactCount := COUNT(Transactions), UCC});

OUTPUT(XactTbl)
```

Essa função TABLE obtém as contagens de múltiplos registros de transação por arquivamento, de forma que possamos usar NORMALIZE para extraí-las em uma tabela própria.

```
Out_Transacts := RECORD
   STRING            FilingNumber;
   STRING            TransactionID;
   STRING10          StartDate;
   STRING10          LapseDate;
   STRING            FormType;
   STRING            AmendType;
   STRING            AmendAction;
   STRING10          EnteredDate;
   STRING10          ReceivedDate;
   STRING10          ApprovedDate;
   DATASET(PartyRec) Debtors;
   DATASET(PartyRec) SecuredParties;
   CollateralRec     Collateral;
END;

Out_Transacts Get_Transacts(XactTbl L, INTEGER C) := TRANSFORM
  SELF.FilingNumber := L.FilingNumber;
  SELF              := L.Transactions[C];
END;

Transacts := NORMALIZE(XactTbl,LEFT.XactCount,Get_Transacts(LEFT,COUNTER));

OUTPUT(Transacts);
```

Essa NORMALIZE extrai todas as transações em um conjunto de registros separado com apenas uma transação por registro e com as informações primárias (o número de arquivamento) anexadas. No entanto, cada registro aqui ainda contém múltiplos registros secundários Debtor e SecuredParty.

```
PartyCounts := TABLE(Transacts,
                     {INTEGER DebtorCount := COUNT(Debtors),
                      INTEGER PartyCount := COUNT(SecuredParties),
                      Transacts});
```

```
OUTPUT(PartyCounts);
```

A função TABLE obtém as contagens dos múltiplos registros Debtor e SecuredParty para cada transação.

```
Out_Parties := RECORD
   STRING   FilingNumber;
   STRING   TransactionID;
   PartyRec;
END;

Out_Parties Get_Debtors(PartyCounts L, INTEGER C) := TRANSFORM
  SELF.FilingNumber  := L.FilingNumber;
  SELF.TransactionID := L.TransactionID;
  SELF               := L.Debtors[C];
END;
TransactDebtors := NORMALIZE( PartyCounts,
                              LEFT.DebtorCount,
                              Get_Debtors(LEFT,COUNTER));

OUTPUT(TransactDebtors);
```

O NORMALIZE extrai todos os Debtors para um conjunto de registros separado.

```
Out_Parties Get_Parties(PartyCounts L, INTEGER C) := TRANSFORM
  SELF.FilingNumber  := L.FilingNumber;
  SELF.TransactionID := L.TransactionID;
  SELF               := L.SecuredParties[C];
END;

TransactParties := NORMALIZE(PartyCounts,
                             LEFT.PartyCount,
                             Get_Parties(LEFT,COUNTER));

OUTPUT(TransactParties);
```

Essa NORMALIZE extrai todas as SecuredParties para um conjunto de registros separado. Com isso, dividimos todos os registros secundários em sua própria estrutura relacional normalizada com a qual podemos trabalhar facilmente.

"Piping" para Ferramenta de Terceiros

Outra forma de trabalhar com dados XML é usar ferramentas de terceiros adaptadas para uso no supercomputador, para ter a vantagem de trabalhar com tecnologia já consagrada e o benefício de executar essa tecnologia em paralelo em todos os nós de supercomputador de uma só vez.

A técnica é simples: basta definir o arquivo de entrada como um fluxo de dados e usar a opção PIPE no DATASET para processar os dados em seu formato nativo. Após ter concluído o processamento, é possível gerar o OUTPUT do resultado em qualquer formato resultante da ferramenta terceirizada, algo como o seguinte código de exemplo (não funcional):

```
Rec := RECORD
  STRING1  char;
END;

TimeZones := DATASET('timezones.xml',Rec,PIPE('ThirdPartyTool.exe'));

OUTPUT(TimeZones,,'ProcessedTimezones.xml');
```

O segredo para essa técnica é a definição do campo STRING1. Isso produz a entrada e a saída de apenas um fluxo de dados de 1 byte por vez, o qual passa pela ferramenta terceirizada e retorna para o código ECL em seu formato nativo. Você não precisa nem saber qual é o formato. Você também pode usar essa técnica com a opção PIPE no OUTPUT.

Trabalhando com BLOBs

O suporte BLOB (Objetos Binários Grandes) no ECL começa com o tipo de valor DATA. Ele pode conter qualquer tipo de dados, tornando a opção perfeita para usar dados BLOB.

Há basicamente três questões ao trabalhar com dados BLOB:

1) Como inserir os dados no HPCC (sparying).

2) Como trabalhar com os dados depois que já estiverem no HPCC.

3) Como retirar os dados do HPCC (despraying).

Spraying de dados BLOB

No arquivo HPCCClientTools.PDF, há um capítulo dedicado ao programa DFUplus.exe. Essa é uma ferramenta de linha de comando com opções específicas que permitem que você realize os processos de spray e despray de arquivos em BLOBs no HPCC. Em todos os exemplos abaixo, vamos supor que você tenha um arquivo DFUPLUS.INI na mesma pasta que o executável com o conteúdo padrão descrito na seção do PDF.

O segredo para que uma operação de spray seja gravada nos BLOBs é o uso da opção *prefix=Filename,Filesize* (Compilar). Por exemplo, a linha de comando a seguir realiza o spray de todos os arquivos .JPG e .BMP do diretório c:\import da máquina 10.150.51.26 em um arquivo lógico único denominado LE::imagedb:

```
C:\>dfuplus action=spray srcip=10.150.51.26 srcfile=c:\import\*.jpg,c:\import\*.bmp
           dstcluster=le_thor dstname=LE::imagedb overwrite=1
           PREFIX=FILENAME,FILESIZE nosplit=1
```

Ao usar caracteres curinga (* e ?) para fazer o spray de vários arquivos de origem (*srcfile*) para um único *dstname*, você (DEVE) MUST usar ambas as opções *filename* e *filesize* (FILENAME,FILESIZE) se precisar realizar o despray dos conteúdos de cada registro no *dstname* de volta para os vários arquivos dos quais eles vieram originalmente.

Trabalhando Dados BLOB

Depois de ter realizado o spray dos dados no HPCC, você precisa definir a estrutura RECORD e o DATASET. A estrutura RECORD a seguir define o resultado do spray acima:

```
imageRecord := RECORD
  STRING filename;
  DATA   image;
       //first 4 bytes contain the length of the image data
  UNSIGNED8  RecPos{virtual(fileposition)};
END;
imageData := DATASET('LE::imagedb',imageRecord,FLAT);
```

O segredo para essa estrutura é o uso de tipos de valor STRING e DATA de comprimento variável. O campo filename recebe o nome completo do arquivo .JPG ou .BMP original que está agora contido no campo de imagem. Os quatro primeiros bytes do campo de imagem contêm um valor inteiro que especifica o número de bytes no arquivo original que estão agora no campo de imagem.

O tipo de valor DATA é usado aqui para o campo BLOB porque os formatos JPG e BMP são essencialmente dados binários. Porém, se o BLOB tivesse dados XML de múltiplos arquivos, ele poderia ser definido como um tipo de valor STRING . Neste caso, os quatro primeiros bytes do campo ainda conteriam um valor inteiro especificando o número de bytes no arquivo original, seguidos pelos dados XML do arquivo.

O limite de tamanho máximo para qualquer valor DATA é 4GB.

A adição do campo RecPos (um campo de "ponteiro de registro" ECL padrão) permite que criemos um INDEX, como este:

```
imageKey := INDEX(imageData,{filename,fpos},'LE::imageKey');
BUILDINDEX(imageKey);
```

Ter um INDEX permite que você trabalhe com o arquivo imageData nas operações JOIN ou FETCH com chave. É claro que também é possível realizar nos arquivos de dados BLOB qualquer operação que você faria com qualquer outro arquivo no ECL.

Despraying de Dados BLOB

O programa DFUplus.exe também permite que você realize o despray de arquivos BLOB do HPCC, dividindo-os novamente em arquivos individuais dos quais foram originados. O segredo para que uma operação de despray seja gravada nos BLOBs para separar os arquivos é usar a opção *splitprefix=Filename,Filesize* Por exemplo, a linha de comando a seguir realiza o despray de todos os dados BLOB para o diretório c:\import\despray da máquina 10.150.51.26 a partir de um arquivo lógico único denominado LE::imagedb:

```
C:\>dfuplus action=despray dstip=10.150.51.26 dstfile=c:\import\despray\*.*
            srcname=LE::imagedb PREFIX=FILENAME,FILESIZE nosplit=1
```

Depois que esse comando é executado, você deve ter o mesmo grupo de arquivos, que foram originalmente distribuídos aos nós, recriados em um diretório separado.

Utilizando Chaves ECL (Arquivos Index)

As operações ETL (extrair, transformar e carregar – processamento de ingestão de dados padrão) no ECL normalmente operam em oposição a todos, ou à maioria, dos registos de um determinado dataset, fazendo com que o uso das chaves (arquivos INDEX) sejam de pouca utilidade. Muitas consultas realizam a mesma operação.

No entanto, a entrega de dados de produção para usuários finais raramente requer o acesso a todos os registros em um dataset. Os usuários finais sempre buscam acesso "instantâneo" para os dados desejados e, muitas vezes, esses dados são um pequeno subconjunto do grupo total de registros disponíveis. Consequentemente, o uso de chaves (INDEXes) passou a ser obrigatório.

As definições dos atributos a seguir usadas pelos exemplos de códigos neste artigo são declaradas no atributo de estrutura MODULE DeclareData no arquivo DeclareData.ECL :

```
EXPORT Person := MODULE
  EXPORT File := DATASET('~PROGGUIDE::EXAMPLEDATA::People',Layout_Person, THOR);
  EXPORT FilePlus := DATASET('~PROGGUIDE::EXAMPLEDATA::People',
                             {Layout_Person,
                              UNSIGNED8 RecPos{VIRTUAL(fileposition)}}, THOR);
END;
EXPORT Accounts := DATASET('~PROGGUIDE::EXAMPLEDATA::Accounts',
                           {Layout_Accounts_Link,
                            UNSIGNED8 RecPos{VIRTUAL(fileposition)}}, THOR);
EXPORT PersonAccounts := DATASET('~PROGGUIDE::EXAMPLEDATA::PeopleAccts',
                                 {Layout_Combined,
                                  UNSIGNED8 RecPos{virtual(fileposition)}},THOR);

EXPORT IDX_Person_PersonID := INDEX(Person.FilePlus,{PersonID,RecPos},
                                    '~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID');
EXPORT IDX_Accounts_PersonID := INDEX(Accounts,{PersonID,RecPos},
                                      '~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID');

EXPORT IDX_Accounts_PersonID_Payload :=
        INDEX(Accounts,
              {PersonID},
              {Account,OpenDate,IndustryCode,AcctType,
               AcctRate,Code1,Code2,HighCredit,Balance,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::Accounts.PersonID.Payload');

EXPORT IDX_PersonAccounts_PersonID :=
        INDEX(PersonAccounts,{PersonID,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::PeopleAccts.PersonID');

EXPORT IDX__Person_LastName_FirstName :=
        INDEX(Person.FilePlus,{LastName,FirstName,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::People.LastName.FirstName');
EXPORT IDX__Person_PersonID_Payload :=
        INDEX(Person.FilePlus,{PersonID},
              {FirstName,LastName,MiddleInitial,
               Gender,Street,City,State,Zip,RecPos},
              '~PROGGUIDE::EXAMPLEDATA::KEYS::People.PersonID.Payload');
```

Embora seja possível usar um INDEX como se fosse um DATASET, há apenas duas operações no ECL que usam chaves diretamente: FETCH e JOIN.

FETCH simples

A operação FETCH é o uso mais simples de um INDEX. Sua finalidade é localizar registros de um dataset usando um INDEX para acessar diretamente apenas os registros especificados.

O código de exemplo abaixo (contido no arquivo IndexFetch.ECL) ilustra a forma comum:

```
IMPORT $;

F1 := FETCH($.DeclareData.Person.FilePlus,
            $.DeclareData.IDX_Person_PersonID(PersonID=1),
            RIGHT.RecPos);
OUTPUT(F1);
```

Você perceberá que o DATASET nomeado como o primeiro parâmetro não possui filtro, enquanto o INDEX nomeado como segundo parâmetro possui um filtro. Isso sempre acontece com a operação FETCH. O propósito do INDEX no ECL é sempre possibilitar o acesso "direto" a registros individuais no dataset de base, de forma que seja sempre necessário filtrar o INDEX para definir o conjunto exato de registros a serem localizados. Devido a isso, a filtragem do dataset de base não é necessária.

Como você pode ver, não há uma função TRANSFORM neste código. Para usos mais comuns de FETCH, uma função de transformação é desnecessária, embora seja certamente adequado se os dados de resultado exigirem formatação, como neste exemplo (também contido no arquivo IndexFetch.ECL):

```
r := RECORD
  STRING FullName;
  STRING Address;
  STRING CSZ;
END;

r Xform($.DeclareData.Person.FilePlus L) := TRANSFORM
  SELF.Fullname := TRIM(L.Firstname) + TRIM(' ' + L.MiddleInitial) + ' ' + L.Lastname;
  SELF.Address  := L.Street;
  SELF.CSZ      := TRIM(L.City) + ', ' + L.State + ' ' + L.Zip;
END;

F2 := FETCH($.DeclareData.Person.FilePlus,
            $.DeclareData.IDX_Person_PersonID(PersonID=1),
            RIGHT.RecPos,
            Xform(LEFT));
OUTPUT(F2);
```

Mesmo com a função TRANSFORM, esse código ainda é basicamente uma operação direta de localização de registros.

Full-keyed JOIN

Ao contrário da simplicidade de FETCH, o uso de INDEXes nas operações JOIN é um pouco mais complexo. A forma mais óbvia é o JOIN, de “full-keyed”, especificado pela opção KEYED , que nomeia um INDEX no conjunto de registros direito (o segundo parâmetro JOIN). A finalidade para essa forma é tratar de situações nas quais o conjunto de registros esquerdo (nomeado como o primeiro parâmetro no JOIN) é um dataset bastante pequeno que precisa ser combinado a um dataset grande e indexado (o conjunto de registros direito). Ao usar a opção KEYED, a operação JOIN utiliza o INDEX especificado para localizar os registros direitos correspondentes. Isso significa que a condição de combinação precisa usar os principais campos no INDEX para localizar os registros correspondentes.

Esse código de exemplo (contido no arquivo IndexFullKeyedJoin.ECL) ilustra o uso habitual de uma combinação com chaves completas:

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;

r1 Xform1($.DeclareData.Person.FilePlus L,
          $.DeclareData.Accounts R) := TRANSFORM
```

```
  SELF := L;
  SELF := R;
END;
J1 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.Accounts,
           LEFT.PersonID=RIGHT.PersonID,
           Xform1(LEFT,RIGHT),
           KEYED($.DeclareData.IDX_Accounts_PersonID));

OUTPUT(J1,ALL);
```

O arquivo de Contas direito contém cinco milhões de registros e, com a condição de filtro especificada, o conjunto de registro Pessoa esquerdo contém exatamente cem registros. Um JOIN padrão entre esses dois normalmente exigiria que todos os cinco milhões de registros de conta fossem lidos para produzir o resultado. No entanto, ao usar a opção KEYED, a árvore binária INDEX será usada para localizar as entradas com os valores de campo de chave adequados e obter os ponteiros para o conjunto exato de registros de contas necessário para produzir o resultado correto. Isso significa que apenas aqueles registros lidos do arquivo direito são os de fato contidos no resultado.

Half-keyed JOIN

A operação half-keyed JOIN é uma versão mais simples, onde o INDEX é o conjunto de registros direito no JOIN. Assim como acontece com full-keyed JOIN, a condição de combinação precisa usar os principais campos no INDEX para realizar seu trabalho. A finalidade de half-keyed JOIN é a mesma da versão de chave completa.

Na verdade, full-keyed JOIN é, nos bastidores, basicamente o mesmo que um JOIN de meia chave e um FETCH para localizar os registros de dataset de base. Com isso, um half-keyed JOIN e um FETCH são semântica e funcionalmente equivalentes, como mostrado neste código de exemplo (contido no arquivo IndexHalfKeyedJoin..ECL file):

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;
r2 := RECORD
  $.DeclareData.Layout_Person;
  UNSIGNED8 AcctRecPos;
END;

r2 Xform2($.DeclareData.Person.FilePlus L,
          $.DeclareData.IDX_Accounts_PersonID R) := TRANSFORM
  SELF.AcctRecPos := R.RecPos;
  SELF := L;
END;

J2 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.IDX_Accounts_PersonID,
           LEFT.PersonID=RIGHT.PersonID,
           Xform2(LEFT,RIGHT));

r1 Xform3($.DeclareData.Accounts L, r2 R) := TRANSFORM
  SELF := L;
  SELF := R;
END;
F1 := FETCH($.DeclareData.Accounts,
            J2,
            RIGHT.AcctRecPos,
            Xform3(LEFT,RIGHT));

OUTPUT(F1,ALL);
```

Esse código produz o mesmo conjunto de resultado que o exemplo anterior.

A vantagem de usar JOINs de meia chave em vez da versão completa é em casos onde você possa precisar realizar vários JOINs para executar qualquer processo que esteja em execução. O uso da forma de meia chave permite que você realize todos os JOINs antes do FETCH explicitamente para localizar os registros de resultado final, tornando assim o código mais eficiente.

Índices de Payload

Há uma forma estendida do INDEX que permite que cada entrada carregue uma "carga útil" – dados adicionais não incluídos no conjunto de campos-chave. Esses campos adicionais podem simplesmente ser campos extras do dataset de base (não precisam fazer parte da chave de busca) ou podem conter o resultado de alguma computação preliminar (campos computados). Uma vez que os dados em um INDEX são sempre compactados (usando a compressão LZW), carregar carga de conteúdo extra não força o sistema incorretamente.

Um INDEX de carga de conteúdo exige duas estruturas RECORD como o segundo e terceiro parâmetros da declaração INDEX. O segundo parâmetro da estrutura RECORD lista os principais campos nos quais o INDEX é desenvolvido (os campos de busca), enquanto o terceiro parâmetro da estrutura RECORD define os campos de carga de conteúdo adicionais.

O campo de ponteiro de registro **virtual(fileposition)** sempre precisa ser o último campo listado em qualquer tipo de INDEX; por isso, ao definir uma chave de carga de conteúdo, ele é sempre o último campo no terceiro parâmetro da estrutura RECORD.

Este código de exemplo (contido no arquivo IndexHalfKeyedPayloadJoin.ECL) duplica novamente os resultados anteriores, porém usando apenas o JOIN de meia chave (sem o FETCH) ao marcar o uso de uma chave de carga de conteúdo:

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;

r1 Xform($.DeclareData.Person.FilePlus L, $.DeclareData.IDX_Accounts_PersonID_Payload R) :=
  TRANSFORM
    SELF := L;
    SELF := R;
END;

J2 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.IDX_Accounts_PersonID_Payload,
           LEFT.PersonID=RIGHT.PersonID,
           Xform(LEFT,RIGHT));

OUTPUT(J2,ALL);
```

É possível ver que isso torna o código mais enxuto. Ao eliminar a operação FETCH, também se elimina o acesso de disco associado a ela, agilizando o processo. O requisito, obviamente, é pré-desenvolver as chaves de carga de conteíudo para que FETCH não seja mais necessário.

Campos Computados nas Chaves de Payload

Há um truque para colocar campos computados na carga útil. Uma vez que um "campo computado", por definição, não existe no dataset, a técnica necessária para sua criação e uso é desenvolver o conteúdo do INDEX antecipadamente.

O código de exemplo a seguir (contido no arquivo IndexPayloadFetch.ECL) ilustra como fazer isso ao desenvolver o conteúdo de alguns campos computados (derivados dos registros secundários) em uma TABLE onde o INDEX é desenvolvido:

```
IMPORT $;

PersonFile := $.DeclareData.Person.FilePlus;
AcctFile   := $.DeclareData.Accounts;
IDXname := '~$.DeclareData::EXAMPLEDATA::KEYS::Person.PersonID.CompPay';

r1 := RECORD
  PersonFile.PersonID;
  UNSIGNED8 AcctCount := 0;
  UNSIGNED8 HighCreditSum := 0;
  UNSIGNED8 BalanceSum := 0;
  PersonFile.RecPos;
END;

t1 := TABLE(PersonFile,r1);
st1 := DISTRIBUTE(t1,HASH32(PersonID));


r2 := RECORD
  AcctFile.PersonID;
  UNSIGNED8 AcctCount := COUNT(GROUP);
  UNSIGNED8 HighCreditSum := SUM(GROUP,AcctFile.HighCredit);
  UNSIGNED8 BalanceSum := SUM(GROUP,AcctFile.Balance);
END;

t2 := TABLE(AcctFile,r2,PersonID);
st2 := DISTRIBUTE(t2,HASH32(PersonID));

r1 countem(t1 L, t2 R) := TRANSFORM
  SELF := R;
  SELF := L;
END;

j := JOIN(st1,st2,LEFT.PersonID=RIGHT.PersonID,countem(LEFT,RIGHT),LOCAL);

Bld := BUILDINDEX(j,
                  {PersonID},
                  {AcctCount,HighCreditSum,BalanceSum,RecPos},
                  IDXname,OVERWRITE);


i := INDEX(PersonFile,
           {PersonID},
           {UNSIGNED8 AcctCount,UNSIGNED8 HighCreditSum,UNSIGNED8 BalanceSum,RecPos},
           IDXname);

f := FETCH(PersonFile,i(PersonID BETWEEN 1 AND 100),RIGHT.RecPos);

Get := OUTPUT(f,ALL);

SEQUENTIAL(Bld,Get);
```

A primeira função TABLE localiza todos os valores dos principais campos do dataset Person para o INDEX e cria campos vazios para conter os valores computados. Observe bem que o valor de campo RecPos virtual(fileposition) também é localizado neste ponto.

A segunda função TABLE calcula os valores que vão nos campos computados. Os valores neste exemplo são provenientes do dataset Accounts relacionado. Esses valores de campo computados vão permitir que o INDEX de carga de conteúdo final no dataset Person produza esses valores de conjunto de registro secundário sem nenhum código adicional (ou acesso de disco).

A operação JOIN combina o resultado de duas TABLEs em sua forma final. Esses são os dados a partir dos quais o INDEX é desenvolvido.

A ação BUILDINDEX grava o INDEX no disco. A parte complicada é então declarar o INDEX em relação ao dataset de base (não ao resultado JOIN). Assim sendo, é essencial para essa técnica desenvolver o INDEX em relação a um conjunto derivado/computado de dados e depois declarar o INDEX em relação ao dataset de base do qual os dados foram extraídos.

Para demonstrar o uso de um INDEX, esse código de exemplo só realiza uma simples operação de FETCH para localizar o resultado combinado com todos os campos do dataset Person juntamente com todos os valores de campo computados. No uso "normal", esse tipo de chave de carga útil seria usado em uma operação half-keyed JOIN.

Campos Computados nas Chaves de Pesquisa

Há uma situação na qual é necessário o uso de um campo computado como uma chave de busca – quando o campo que você deseja procurar está em um tipo de dados REAL ou DECIMAL. Nenhum desses dois é válido para uso como uma chave de busca. Consequentemente, fazer da chave de busca um campo de STRING computado contendo o valor para pesquisa é uma forma de superar essa limitação.

O segredo para os campos computados na carga útil é o mesmo para chaves de busca – desenvolver o conteúdo do INDEX antecipadamente O exemplo a seguir (contido no arquivo IndexREALkey.ECL) ilustra como fazer isso ao desenvolver o conteúdo dos campos de chave de busca computados nos quais o INDEX é desenvolvido usando TABLE e PROJECT:

```
IMPORT $;

r := RECORD
  REAL8      Float := 0.0;
  DECIMAL8_3 Dec   := 0.0;
  $.DeclareData.person.file;
END;
t := TABLE($.DeclareData.person.file,r);

r XF(r L) := TRANSFORM
  SELF.float := L.PersonID / 1000;
  SELF.dec := L.PersonID / 1000;
  SELF := L;
END;
p := PROJECT(t,XF(LEFT));

DSname   := '~PROGGUIDE::EXAMPLEDATA::KEYS::dataset';
IDX1name := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestIDX1';
IDX2name := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestIDX2';
OutName1 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout1';
OutName2 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout2';
OutName3 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout3';
OutName4 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout4';
OutName5 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout5';
OutName6 := '~PROGGUIDE::EXAMPLEDATA::KEYS::realkeytestout6';

DSout := OUTPUT(p,,DSname,OVERWRITE);

ds := DATASET(DSname,r,THOR);

idx1 := INDEX(ds,{STRING13 FloatStr := REALFORMAT(float,13,3)},{ds},IDX1name);
idx2 := INDEX(ds,{STRING13 DecStr := (STRING13)dec},{ds},IDX2name);

Bld1Out := BUILD(idx1,OVERWRITE);
Bld2Out := BUILD(idx2,OVERWRITE);

j1 := JOIN(idx1,idx2,LEFT.FloatStr = RIGHT.DecStr);
j2 := JOIN(idx1,idx2,KEYED(LEFT.FloatStr = RIGHT.DecStr));
j3 := JOIN(ds,idx1,KEYED((STRING10)LEFT.float = RIGHT.FloatStr));
j4 := JOIN(ds,idx2,KEYED((STRING10)LEFT.dec = RIGHT.DecStr));
```

```
j5 := JOIN(ds,idx1,KEYED((STRING10)LEFT.dec = RIGHT.FloatStr));
j6 := JOIN(ds,idx2,KEYED((STRING10)LEFT.float = RIGHT.DecStr));

JoinOut1 := OUTPUT(j1,,OutName1,OVERWRITE);
JoinOut2 := OUTPUT(j2,,OutName2,OVERWRITE);
JoinOut3 := OUTPUT(j3,,OutName3,OVERWRITE);
JoinOut4 := OUTPUT(j4,,OutName4,OVERWRITE);
JoinOut5 := OUTPUT(j5,,OutName5,OVERWRITE);
JoinOut6 := OUTPUT(j6,,OutName6,OVERWRITE);

SEQUENTIAL(DSout,Bld1Out,Bld2Out,JoinOut1,JoinOut2,JoinOut3,JoinOut4,JoinOut5,JoinOut6);
```

Esse código começa com algumas definições de nome de arquivo. A estrutura de registro adiciona dois campos ao conjunto de campos existente de nosso dataset de base: um campo REAL8 denominado "float" e um campo DECIMAL12_6 denominado "dec." Esses campos conterão nossos dados REAL e DECIMAL que desejamos pesquisar. O PROJECT da TABLE insere valores nesses dois campos (neste caso, simplesmente dividindo o arquivo PersonID por 1.000 para alcançar um valor de ponto flutuante exclusivo para ser usado).

A definição IDX1 INDEX cria a chave de busca REAL como um campo computado STRING13 usando a função REALFORMAT para justificar à direita o valor de ponto flutuante em uma STRING de 13 caracteres. Isso padroniza o valor com o número exato de casas decimais especificado na função REALFORMAT.

A definição IDX2 INDEX cria a chave de busca DECIMAL como um campo computado STRING13 ao atribuir os dados DECIMAL para uma STRING13. Usar o operador de representação de tipos apenas justifica à esquerda o valor na string. Ele também pode incluir zeros à direita de forma que não há garantia de que o número de casas decimais será sempre o mesmo.

Uma vez que há dois métodos distintos de elaborar as strings de chave de busca, as próprias strings não são iguais, embora os valores usados para criá-las sejam os mesmos. Isso significa que você não pode esperar uma "mistura e combinação" entre as duas – é necessário usar cada INDEX com o método usado para criá-la. É por isso que as duas operações JOIN que demonstram seu uso utilizam o mesmo método para criar o valor de comparação de string conforme usado para criar o INDEX. Dessa maneira, você tem a certeza de que terá valores correspondentes.

Utilizando um INDEX como DATASET

As chaves de carga útil também podem ser usadas para operações de tipo DATASET padrão. Nesse tipo de uso, o INDEX atua como se fosse um dataset, com a vantagem de conter dados comprimidos e um índice btree. A principal diferença neste tipo de uso é a utilização de KEYED e WILD nos filtros INDEX , que permite a leitura do INDEX para utilizar o btree em vez de realizar uma busca completa na tabela.

O código de exemplo a seguir (contido no arquivo IndexAsDataset.ECL)ilustra o uso de um INDEX como se ele fosse um DATASET, e compara o desempenho relativo do INDEX em relação ao uso do DATASET:

```
IMPORT $;

OutRec := RECORD
  INTEGER   Seq;
  QSTRING15 FirstName;
  QSTRING25 LastName;
  STRING2   State;
END;

IDX  := $.DeclareData.IDX__Person_LastName_FirstName_Payload;
Base := $.DeclareData.Person.File;

OutRec XF1(IDX L, INTEGER C) := TRANSFORM
  SELF.Seq := C;
  SELF := L;
END;
```

```
O1 := PROJECT(IDX(KEYED(lastname='COOLING'),
                  KEYED(firstname='LIZZ'),
              state='OK'),
              XF1(LEFT,COUNTER));
OUTPUT(O1,ALL);

OutRec XF2(Base L, INTEGER C) := TRANSFORM
  SELF.Seq := C;
  SELF := L;
END;

O2 := PROJECT(Base(lastname='COOLING',
                   firstname='LIZZ',
              state='OK'),
              XF2(LEFT,COUNTER));
OUTPUT(O2,ALL);
```

As duas operações PROJECT produzirão exatamente o mesmo resultado, mas a primeira utiliza um INDEX e a segunda um DATASET. A única diferença significativa entre as duas é o uso de KEYED no filtro INDEX. Isso indica que a leitura do index deve usar o btree para localizar o conjunto específico de registros de nó folha para leitura. A versão DATASET precisa ler todos os registros no arquivo para localizar o correto, fazendo deste um processo bem mais lento.

Se verificar as medidas de tempo de tarefa no ECL Watch, você perceberá uma diferença. Neste caso de testes, a diferença pode não parecer grande (não há muitos dados de testes), mas em suas aplicações reais a diferença entre uma operação de leitura de índice e uma verificação completa de tabela deve ser significativa.

Trabalhando com Superarquivos

Visão Geral de Superarquivo

Primeiro, vamos definir alguns termos:

Logical File	Uma entidade lógica única cujas múltiplas partes físicas (uma em cada nó do cluster) são gerenciadas internamente pelo utilitário de arquivos distribuídos (DFU).
Dataset	Um arquivo lógico declarado como um DATASET.
SuperFile	Uma lista gerenciada de subarquivos (arquivos lógicos) tratados como uma entidade lógica única. Os subarquivos não precisam de declarações DATASET (embora possam ter). Um superarquivo precisa ser declarado como um DATASET para ser usado no ECL, e é tratado no código ECL como qualquer outro dataset. As complexidades de gerenciar múltiplos subarquivos são deixadas para o DFU (exatamente como ele gerencia as partes físicas de cada subarquivo).

Cada subarquivo em um superarquivo precisa ter o mesmo tipo de estrutura (THOR, CSV, ou XML) e o mesmo layout do campo. O próprio subarquivo pode ser um superarquivo, permitindo que você crie hierarquias multinível que possibilitam fácil manutenção. Todas as funções que criam e mantêm os superarquivos estão na biblioteca padrão de arquivos (consulte a *Referência de biblioteca padrão*).

A principal vantagem de usar superarquivos é a fácil manutenção do conjunto de subarquivos. Isso significa que atualizar os dados que uma consulta de fato lê pode ser tão simples quanto adicionar um novo subarquivo a um superarquivo existente.

Funções de Existência do Superarquivo

As funções a seguir tratam da criação, exclusão e detecção da existência de superarquivos:

```
CreateSuperFile()
DeleteSuperFile()
SuperFileExists()
```

É necessário criar um superarquivo usando a função CreateSuperFile() antes de poder realizar qualquer uma das outras operações de superarquivo no arquivo. A função SuperFileExists() informa se existe um superarquivo com o nome especificado, enquanto DeleteSuperFile() remove um superarquivo do sistema.

Funções de Consulta do Superarquivo

As funções a seguir fornecem informações sobre um determinado superarquivo:

```
GetSuperFileSubCount()
GetSuperFileSubName()
FindSuperFileSubName()
SuperFileContents()
LogicalFileSuperOwners()
```

A função GetSuperFileSubCount() permite que você determine o número de subarquivos em um determinado superarquivo. A função GetSuperFileSubName() retorna o nome do subarquivo em uma determinada posição na lista de subarquivos. A função FindSuperFileSubName() retorna a posição ordinal de um determinado subarquivo na lista de subarquivos. A função SuperFileContents() retorna um grupo de registros de nomes de subarquivo lógicos contidos no superarquivo. A função LogicalFileSuperOwners retorna uma lista de todos os superarquivos que contêm um subarquivo especificado.

Funções de Manutenção do Superarquivo

As funções a seguir permitem que você mantenha a lista de subarquivos que compõem um superarquivo.

```
AddSuperFile()
RemoveSuperFile()
ClearSuperFile()
SwapSuperFile()
ReplaceSuperFile()
```

A função AddSuperFile() adiciona um subarquivo ao superarquivo. A função RemoveSuperFile() remove um subarquivo do superarquivo. A função ClearSuperFile() apaga todos os subarquivos do superarquivo. A função SwapSuperFile() alterna todos os subarquivos entre os dois superarquivos. A função ReplaceSuperFile() substitui um subarquivo no superarquivo por outro.

Todas essas funções precisam ser acionadas em um período de transação a fim de garantir que não haja problemas com o uso de superarquivo.

Transações de Superarquivos

A funções de manutenção de superarquivo (apenas elas) precisam ser acionadas em um determinado período de transação se houver a possibilidade de que outro processo tente usar o superarquivo durante a manutenção do subarquivo. O período de transação bloqueia todas as outras operações durante a duração da transação. Dessa forma, o trabalho de manutenção pode ser feito sem causar problemas com nenhuma consulta que possa usar o superarquivo. Isso significa duas coisas:

1) A ação SEQUENTIAL (SEQUENCIAL) precisa ser usada a fim de garantir a execução dos acionamentos de função dentro do período de transação.

2) As funções StartSuperFileTransaction() e FinishSuperFileTransaction() são usadas para "bloquear" o superarquivo durante a manutenção e sempre envolvem os acionamentos de função de manutenção de superarquivo na ação SEQUENTIAL.

Qualquer função além das funções de manutenção acima listadas que possa estar presente dentro de um período de transação pode aparecer como parte da transação, mas não é. Isso pode causar confusão se você, por exemplo, incluir um acionamento para ClearSuperFile() (que é válido para uso no período de transação) e segui-lo com um acionamento para DeleteSuperFile() (que não é válido para uso no período de transação); isso causará um erro, uma vez que a operação ocorrerá fora do período de transação e antes que a função ClearSuperFile() tenha chance de realizar seu trabalho.

Outras Funções Úteis

As funções a seguir, embora não sejam especificamente projetadas para uso em superarquivo, são normalmente úteis para criar e manter superarquivos:

```
RemoteDirectory()
ExternalLogicalFilename()
LogicalFileList()
LogicalFileSuperOwners()
```

O uso dessas funções será descrito no próximo conjunto de artigos sobre superarquivos.

Criando e Mantendo SuperFiles

Criando Dados

Primeiro, precisamos criar alguns arquivos lógicos para inserir em um superarquivo.

Os nomes dos arquivos a seguir para os novos subarquivos são declarados no atributo de estrutura MODULE Declare-Data:

```
EXPORT BaseFile := '~PROGGUIDE::SUPERFILE::Base';
EXPORT SubFile1 := '~PROGGUIDE::SUPERFILE::People1';
EXPORT SubFile2 := '~PROGGUIDE::SUPERFILE::People2';
EXPORT SubFile3 := '~PROGGUIDE::SUPERFILE::People3';
EXPORT SubFile4 := '~PROGGUIDE::SUPERFILE::People4';
EXPORT SubFile5 := '~PROGGUIDE::SUPERFILE::People5';
EXPORT SubFile6 := '~PROGGUIDE::SUPERFILE::People6';
```

O código a seguir (no arquivo SuperFile1.ECL) cria os arquivos que vamos usar para compilar superarquivos:

```
IMPORT $;
IMPORT Std;

s1 := $.DeclareData.Person.File(firstname[1] = 'A');
s2 := $.DeclareData.Person.File(firstname[1] BETWEEN 'B' AND 'C');
s3 := $.DeclareData.Person.File(firstname[1] BETWEEN 'D' AND 'J');
s4 := $.DeclareData.Person.File(firstname[1] BETWEEN 'K' AND 'N');
s5 := $.DeclareData.Person.File(firstname[1] BETWEEN 'O' AND 'R');
s6 := $.DeclareData.Person.File(firstname[1] BETWEEN 'S' AND 'Z');

Rec := $.DeclareData.Layout_Person;

IF(~Std.File.FileExists($.DeclareData.SubFile1),
   OUTPUT(s1,,$.DeclareData.SubFile1));

IF(~Std.File.FileExists($.DeclareData.SubFile2),
   OUTPUT(s2,,$.DeclareData.SubFile2));

IF(~Std.File.FileExists($.DeclareData.SubFile3),
   OUTPUT(s3,,$.DeclareData.SubFile3));

IF(~Std.File.FileExists($.DeclareData.SubFile4),
   OUTPUT(s4,,$.DeclareData.SubFile4));

IF(~Std.File.FileExists($.DeclareData.SubFile5),
   OUTPUT(s5,,$.DeclareData.SubFile5));

IF(~Std.File.FileExists($.DeclareData.SubFile6),
   OUTPUT(s6,,$.DeclareData.SubFile6));
```

Esse código obtém dados do dataset ProgGuide.Person.File (criado pelo código no GenData.ECL e declarado no atributo de estrutura MODULE ProgGuide no módulo Default) e grava seis amostras discretas separadas em seus próprios arquivos lógicos, mas apenas se elas ainda não existirem. Vamos usar esses arquivos lógicos para compilar alguns superarquivos.

Um Exemplo Simples

Vamos começar com um exemplo simples sobre como criar e usar um superarquivo. Essa declaração de dataset está na estrutura MODULE ProgGuide (contida no módulo Default). Isso declara o superarquivo como um DATASET que pode ser referenciado no código ECL:

```
EXPORT SuperFile1 := DATASET(BaseFile,Layout_Person,FLAT);
```

Vamos então criar e adicionar subarquivos a um superarquivo (esse código está contido no arquivo SuperFile2.ECL):

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.CreateSuperFile($.DeclareData.BaseFile),
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.BaseFile,$.DeclareData.SubFile1),
  Std.File.AddSuperFile($.DeclareData.BaseFile,$.DeclareData.SubFile2),
  Std.File.FinishSuperFileTransaction());
```

Se a tarefa falhar com uma mensagem de erro "logical name progguide::superfile::base already exists", abra o arquivo SuperFileRestart.ECL e execute-o; em seguida, tente o código acima novamente. Depois de ter conseguido executar esse código em uma janela do compilador, você criou o superarquivo e adicionou dois subarquivos a ele.

O atributo de declaração DATASET SuperFile1 disponibiliza o superarquivo para uso assim como qualquer outro DATASET faria – esse é o segredo para usar superarquivos. Isso significa que os seguintes tipos de ações podem ser executadas em um superarquivo, assim como com qualquer outro dataset:

```
IMPORT $;
COUNT($.DeclareData.SuperFile1(PersonID <> 0));
OUTPUT($.DeclareData.SuperFile1);
```

Devido aos arquivos lógicos anteriormente compilados, os resultados da COUNT devem ser 317.000. A condição de filtro sempre será verdadeira, de forma que a COUNT retornada será o número total de registros no superarquivo. O filtro do registro (PersonID <> 0) é necessário para que a COUNT real seja realizada toda vez que o resultado não for um valor de atalho armazenado internamente pelo ECL Agent. Obviamente, o OUTPUT produz os 100 primeiros registros no superarquivo.

Aninhando Superarquivos

A técnica de aninhar superarquivos (um superarquivo que contém um subarquivo que é, na verdade, outro superarquivo) é uma técnica que permite que novos dados recebidos periodicamente (diariamente, a cada hora ou outros) sejam disponibilizados "instantaneamente" no sistema. Uma vez que o código ECL que se refere a um superarquivo sempre faz referência à declaração DATASET, a única mudança necessária para disponibilizar novos dados para consultas é adicionar os novos dados como um subarquivo. Uma vez que a adição de um subarquivo é sempre realizada em uma transação de superarquivo, quaisquer consultas são bloqueadas enquanto a atualização está em andamento.

Também está implícito nessa técnica a implantação e consolidação periódicas de novos dados em arquivos compostos. Isso é necessário porque o limite prático para o número de arquivos físicos que você deve combinar em um superarquivo é de aproximadamente 100 (cem), já que toda vez que se faz referência ao superarquivo todo subarquivo precisa ter sido fisicamente aberto e lido no disco e, quanto mais subarquivos houver, mais recursos de sistema operacional são usados para ter acesso aos dados.

Dessa forma, é necessário executar periodicamente um processo que combine fisicamente todos os arquivos lógicos incrementais e combiná-los em um arquivo lógico único que substitua todos eles. A consolidação periódica dos dados em superarquivos é um processo simples de uso do OUTPUT para gravar conteúdos completos do superarquivo em um arquivo lógico único e novo. Depois que todos os dados estão em um único arquivo, uma transação de superarquivo pode apagar o conjunto antigo de subarquivos e depois adicionar o novo arquivo lógico de "base".

Exemplo de Superarquivo Aninhado

Aqui está um exemplo de como aninhar superarquivos. Este exemplo supõe que você receba novos dados diariamente. Ele também supõe que você queira implantar novos dados diariamente e semanalmente. Os nomes dos arquivos a seguir para os novos subarquivos são declarados no atributo de estrutura MODULE DeclareData:

```
EXPORT AllPeople := '~PROGGUIDE::SUPERFILE::AllPeople';
EXPORT WeeklyFile := '~PROGGUIDE::SUPERFILE::Weekly';
EXPORT DailyFile := '~PROGGUIDE::SUPERFILE::Daily';
```

A criação de três ou mais superarquivos precisa ser apenas uma vez; depois, é necessário adicionar os subarquivos a eles (este código está contido no arquivo SuperFile3.ECL):

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.CreateSuperFile($.DeclareData.AllPeople),
  Std.File.CreateSuperFile($.DeclareData.WeeklyFile),
  Std.File.CreateSuperFile($.DeclareData.DailyFile),
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.AllPeople,$.DeclareData.BaseFile),
  Std.File.AddSuperFile($.DeclareData.AllPeople,$.DeclareData.WeeklyFile),
  Std.File.AddSuperFile($.DeclareData.AllPeople,$.DeclareData.DailyFile),
  Std.File.FinishSuperFileTransaction());
```

Agora o superarquivo AllPeople contém os superarquivos BaseFile, WeklyFile e DailyFile como subarquivos, criando uma hierarquia de superarquivos, sendo que apenas um ainda não contém dados reais. O superarquivo Base contém todos os dados atualmente conhecidos, como a hora da compilação dos arquivos lógicos. Os superarquivos Weekly e Daily terão atualizações provisórias de dados conforme são recebidos, acabando com a necessidade de recompilar todo o banco de dados sempre que um novo conjunto de dados for recebido.

Um aviso importante neste esquema é que um determinado arquivo lógico real (arquivo de dados real) deve estar contido em exatamente um dos superarquivos aninhados por vez. Caso contrário haverá registros duplicados no superarquivo de base. Dessa forma, é necessário ter cuidado em como se mantém a hierarquia a fim de não permitir que o mesmo arquivo lógico seja referido por mais de um dos superarquivos aninhados por vez, fora de um quadro de transação.

Ao receber novos arquivos lógicos no decorrer do dia, é possível adicioná-los ao superarquivo diário da seguinte maneira (este código está contido no SuperFile4.ECL):

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile3),
  Std.File.FinishSuperFileTransaction());
```

Isso anexa o arquivo lógico ProgGuide.SubFile3 à lista de subarquivos no SuperFile DailyFile. Isso significa que a próxima consulta que usar um dataset SuperFile1 estará utilizando os dados mais recentes – atualizados no último minuto.

Essa declaração de dataset está na estrutura MODULE DeclareData (contida no módulo Default). Isso declara o superarquivo aninhado como um DATASET que pode ser referenciado no código ECL:

```
EXPORT SuperFile2 := DATASET(AllPeople,Layout_Person,FLAT);
```

Execute a seguinte ação:

```
IMPORT ProgrammersGuide AS PG;
COUNT(PG.DeclareData.SuperFile2(PersonID <> 0));
```

O resultado de COUNT agora deve ser 451.000.

Edite o código do SuperFile4.ECL para adicionar no ProgGuide.SubFile4, da seguinte maneira:

```
IMPORT $;
```

```
IMPORT Std;

SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile4),
  Std.File.FinishSuperFileTransaction());
```

A reexecução da ação COUNT acima, deve agora resultar em 620.000.

Uma vez por dia, é possível implantar todos os subarquivos no WeeklyFile e apagar o DailyFile para o processamento de ingestão de dados do dia seguinte da seguinte maneira (este código está contido no arquivo SuperFile5.ECL):

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.WeeklyFile,$.DeclareData.DailyFile,,TRUE),
  Std.File.ClearSuperFile($.DeclareData.DailyFile),
  Std.File.FinishSuperFileTransaction());
```

Isso move as referências de todos os subarquivos do DailyFile para o WeeklyFile (o quatro parâmetro para a função AddSuperFile sendo TRUE copia as referências de um superarquivo para outro), e depois apaga o DailyFile.

Consolidação de dados

Uma vez que o limite prático para o número de arquivos lógicos que você deve combinar em um superarquivo é limitado a cem, será necessário executar um processo que combine fisicamente todos os arquivos lógicos incrementais e os combine em um único arquivo lógico que substitua todos os demais das seguinte maneira:

```
IMPORT $;
IMPORT Std;

OUTPUT($.DeclareData.SuperFile2,,'~$.DeclareData::SUPERFILE::People14',OVERWRITE)
```

Isso gravará um novo arquivo contendo todos os registros de todos os subarquivos no superarquivo.

Depois de fazer isso, será necessário apagar todos os superarquivos de componentes e adicionar o novo arquivo de dados com todos os dados existentes no BaseFile, da seguinte maneira (este código está contido no SuperFile6.ECL):

```
IMPORT $;
IMPORT Std;
SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.ClearSuperFile($.DeclareData.BaseFile),
  Std.File.ClearSuperFile($.DeclareData.WeeklyFile),
  Std.File.ClearSuperFile($.DeclareData.DailyFile),
  Std.File.AddSuperFile($.DeclareData.BaseFile,'~$.DeclareData::SUPERFILE::People14'),
  Std.File.FinishSuperFileTransaction());
```

Essa ação apaga o superarquivo de base, adiciona a referência ao novo arquivo lógico completo e depois apaga todo os superarquivos incrementais.

A reexecução da ação COUNT acima, deve ainda resultar em 620.000.

Novamente, edite o código do arquivo SuperFile4.ECL para adicionar os arquivos ProgGuide.SubFile5 e ProgGuide.SubFile6 ao DailyFile, da seguinte maneira:

```
IMPORT $;
IMPORT Std;
```

```
SEQUENTIAL(
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile5),
  Std.File.AddSuperFile($.DeclareData.DailyFile,$.DeclareData.SubFile6),
  Std.File.FinishSuperFileTransaction());
```

Depois de fazer isso, a reexecução da ação COUNT acima, deve agora resultar em 1.000.000.

Obtendo componentes do superarquivo

Essa macro (no atributo de estrutura MODULE DeclareData) demonstra uma técnica para listar os subarquivos de componentes de um superarquivo:

```
IMPORT STD;
EXPORT MAC_ListSFsubfiles(SuperFile) := MACRO

#UNIQUENAME(SeedRec)
%SeedRec% := DATASET([{''}], {STRING name});

#UNIQUENAME(Xform)
TYPEOF(%SeedRec%) %Xform%(%SeedRec% L, INTEGER C) :=
          TRANSFORM
SELF.name :=
          Std.File.GetSuperFileSubName(SuperFile,C);
END;

OUTPUT(NORMALIZE(%SeedRec%,
Std.File.GetSuperFileSubCount(SuperFile),
%Xform%(LEFT,COUNTER)));
ENDMACRO;
```

A técnica interessante aqui é o uso de NORMALIZE para acionar a função TRANSFORM de forma iterativa até que todos os subarquivos no estejam listados no superarquivo. É possível acionar essa macro em uma janela de compilador como essa (este código está contido no arquivo SuperFile7.ECL):

```
IMPORT $;
IMPORT Std;

$.DeclareData.MAC_ListSFsubfiles($.DeclareData.AllPeople);
```

Isso retornará uma lista de todos os subarquivos no superarquivo especificado. No entanto, esse tipo de código não é mais necessário, uma vez que o modo padrão da função SuperFileContents() agora retorna exatamente o mesmo resultado, da seguinte forma:

```
IMPORT $;
IMPORT Std;
OUTPUT(Std.File.SuperFileContents($.DeclareData.AllPeople));
```

A função SuperFileContents() tem uma vantagem sobre a macro: ela tem uma opção para retornar os subarquivos de qualquer superarquivo aninhado (algo que a macro não oferece). Dessa forma, a função se parece com o seguinte:

```
IMPORT $;
IMPORT Std;
OUTPUT(Std.File.SuperFileContents($.DeclareData.AllPeople,TRUE));
```

Indexando em Superarquivos

Superarquivos vs. Superchaves

Um superarquivo pode conter arquivos INDEX em vez de arquivos DATASET, tornando-o uma Superchave. Os mesmos processos e princípios de criação e manutenção são aplicáveis conforme previamente descrito no tópico *Como criar e manter superarquivos* .

Porém, **uma Superchave pode não conter subarquivos INDEX que fazem referência direta aos subarquivos de um superarquivo usando o mecanismo do "indicador de registro" {virtual(fileposition)}** (usado pelas operações FETCH e JJOIN full-keyed). Isso acontece porque o campo {virtual(fileposition)} é um campo virtual (existe apenas quando o arquivo é lido a partir do disco) que contém a posição relativa do byte de cada registro em uma única entidade lógica.

As definições dos atributos a seguir usadas pelos exemplos de códigos neste artigo são declaradas no atributo de estrutura MODULE DeclareData:

```
EXPORT i1name := '~PROGGUIDE::SUPERKEY::IDX1';
EXPORT i2name := '~PROGGUIDE::SUPERKEY::IDX2';
EXPORT i3name := '~PROGGUIDE::SUPERKEY::IDX3';
EXPORT SFname := '~PROGGUIDE::SUPERKEY::SF1';
EXPORT SKname := '~PROGGUIDE::SUPERKEY::SK1';
EXPORT ds1 := DATASET(SubFile1,{Layout_Person,UNSIGNED8 RecPos {VIRTUAL(fileposition)}},THOR);
EXPORT ds2 := DATASET(SubFile2,{Layout_Person,UNSIGNED8 RecPos {VIRTUAL(fileposition)}},THOR);
EXPORT i1 := INDEX(ds1,{personid,RecPos},i1name);
EXPORT i2 := INDEX(ds2,{personid,RecPos},i2name);
EXPORT sf1 := DATASET(SFname,{Layout_Person,UNSIGNED8 RecPos {VIRTUAL(fileposition)}},THOR);
EXPORT sk1 := INDEX(sf1,{personid,RecPos},SKname);
EXPORT sk2 := INDEX(sf1,{personid,RecPos},i3name );
```

Há uma problema

A maneira mais fácil de ilustrar o problema é executar o código a seguir (este código está contido no IndexSuperFile1.ECL) que usa dois subarquivos do tópico Como criar e manter superarquivos . *Criando e Mantendo Superarquivos*

```
IMPORT $;

OUTPUT($.DeclareData.ds1);
OUTPUT($.DeclareData.ds2);
```

Você notará que os valores RecPos retornados de ambos os datasets são exatamente os mesmos (0, 89, 178, etc.). Isso é esperado, já que ambos possuem a mesma estrutura RECORD de comprimento fixo. O problema está no uso deste campo ao compilar INDEXes individuais para os dois datasets. Ele funciona perfeitamente como INDEXes individuais em DATASETs individuais.

Por exemplo, é possível usar este código para compilar e testar INDEXes individuais (contido em IndexSuperFile2..ECL):

```
IMPORT $;

Bld := PARALLEL(BUILDINDEX($.DeclareData.i1,OVERWRITE),BUILDINDEX
                          ($.DeclareData.i2,OVERWRITE));

F1 := FETCH($.DeclareData.ds1,
            $.DeclareData.i1(personid=$.DeclareData.ds1[1].personid),
            RIGHT.RecPos);
F2 := FETCH($.DeclareData.ds2,
```

```
            $.DeclareData.i2(personid=$.DeclareData.ds2[1].personid),
            RIGHT.RecPos);

Get := PARALLEL(OUTPUT(F1),OUTPUT(F2));
SEQUENTIAL(Bld,Get);
```

Neste caso, dois registros diferentes são retornados pelas duas operações FETCH. Porém, ao criar um superarquivo e uma superchave e em seguida tentar usá-los para realizar as duas mesmas operações FETCHes novamente, ambos retornam o mesmo registro, como mostrado por este código (contido no IndexSuperFile3.ECL):

```
IMPORT $;
IMPORT Std;

BldSF := SEQUENTIAL(
  Std.File.CreateSuperFile($.DeclareData.SFname),
  Std.File.CreateSuperFile($.DeclareData.SKname),
  Std.File.StartSuperFileTransaction(),
  Std.File.AddSuperFile($.DeclareData.SFname,$.DeclareData.SubFile1),
  Std.File.AddSuperFile($.DeclareData.SFname,$.DeclareData.SubFile2),
  Std.File.AddSuperFile($.DeclareData.SKname,$.DeclareData.i1name),
  Std.File.AddSuperFile($.DeclareData.SKname,$.DeclareData.i2name),
  Std.File.FinishSuperFileTransaction());

F1  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk1(personid=$.DeclareData.ds1[1].personid),
             RIGHT.RecPos);
F2  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk1(personid=$.DeclareData.ds2[1].personid),
             RIGHT.RecPos);
Get := PARALLEL(OUTPUT(F1),OUTPUT(F2));
SEQUENTIAL(BldSF,Get);
```

Após combinar os DATASETS em um superarquivo e combinar os INDEXes em uma superchave, obtém-se múltiplas entradas na superchave (com valores de campo de chave diferentes) apontadas para o mesmo registro físico no superarquivo, uma vez que os valores do ponteiro do registro são os mesmos.

E a Solução é ...

Para solucionar este problema é preciso criar um único INDEX no superarquivo, como mostrado através deste código (contido no IndexSuperFile4.ECL):

```
IMPORT $;

F1  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk2(personid=$.DeclareData.ds1[1].personid),
             RIGHT.RecPos);
F2  := FETCH($.DeclareData.sf1,
             $.DeclareData.sk2(personid=$.DeclareData.ds2[1].personid),
             RIGHT.RecPos);
Get := PARALLEL(OUTPUT(F1),OUTPUT(F2));

SEQUENTIAL(BUILDINDEX($.DeclareData.sk2,OVERWRITE),Get);
```

Ao usar um único INDEX em vez de uma superchave, as operações FETCH localizarão novamente os registros corretos.

Utilizando Superchaves

Um superarquivo cujos subarquivos são INDEXes (e não DATASETs) corresponde a uma Superchave. Conforme descrito no tópico anterior, *Como indexar em Superarquivos* , há um problema com o uso de uma superchave para tentar indexar em um superarquivo. Assim sendo, para que servem as superchaves afinal?

No tópico *Como usar chaves ECL (arquivos INDEX)* , doi demonstrada a técnica de criar e usar INDEXes que contêm campos de conteúdo. Ao colocar os campos de conteúdo no próprio INDEX, elimina-se a necessidade de acessar diretamente o dataset de base do qual o INDEX foi compilado. Assim sendo, o problema se torna discutível.

As superchaves são compiladas com chaves de conteúdo. E já que a operação primária para uma chave de conteúdo é o HLAF-KEYED JOINS, esse também se torna o principal uso operacional da superchave.

Ambos superarquivos e superchaves podem ser usados em operações no Thor ou Roxie.

As definições dos atributos a seguir usadas pelos exemplos de códigos neste artigo são declaradas no atributo de estrutura MODULE DeclareData:

```
EXPORT SubKey1 := '~PROGGUIDE::SUPERKEY::Accounts1';
EXPORT SubKey2 := '~PROGGUIDE::SUPERKEY::Accounts2';
EXPORT SubKey3 := '~PROGGUIDE::SUPERKEY::Accounts3';
EXPORT SubKey4 := '~PROGGUIDE::SUPERKEY::Accounts4';
EXPORT SubIDX1 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX1';
EXPORT SubIDX2 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX2';
EXPORT SubIDX3 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX3';
EXPORT SubIDX4 := '~PROGGUIDE::SUPERKEY::KEY::AcctsIDX4';
EXPORT AcctSKname :=
       '~PROGGUIDE::SUPERKEY::KEY::AcctsSK';
EXPORT AcctSK := INDEX(Accounts,{PersonID},
```

Construindo Superchaves

Antes que uma superchave possa ser criada, primeiro é necessário ter alguns INDEXes para usar. O código a seguir (contido no arquivo SuperKey1.ECL) cria quatro chaves de conteúdo individuais a partir do dataset Account:

```
IMPORT $;
IMPORT Std;

s1 := $.DeclareData.Accounts(Account[1] = '1');
s2 := $.DeclareData.Accounts(Account[1] = '2');
s3 := $.DeclareData.Accounts(Account[1] = '3');
s4 := $.DeclareData.Accounts(Account[1] IN ['4','5','6','7','8','9']);

Rec := $.DeclareData.Layout_Accounts_Link;
RecPlus := {Rec,UNSIGNED8 RecPos{virtual(fileposition)}};
d1 := DATASET($.DeclareData.SubKey1,RecPlus,THOR);
d2 := DATASET($.DeclareData.SubKey2,RecPlus,THOR);
d3 := DATASET($.DeclareData.SubKey3,RecPlus,THOR);
d4 := DATASET($.DeclareData.SubKey4,RecPlus,THOR);

i1 := INDEX(d1,{PersonID},
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX1);
i2 := INDEX(d2,{PersonID},
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX2);
i3 := INDEX(d3,{PersonID},
```

```
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX3);
i4 := INDEX(d4,{PersonID},
            {Account,OpenDate,IndustryCode,AcctType,AcctRate,
             Code1,Code2,HighCredit,Balance,RecPos},
            $.DeclareData.SubIDX4);

BldDat := PARALLEL(
        IF(~Std.File.FileExists($.DeclareData.SubKey1),
           OUTPUT(s1,
                 {PersonID,Account,OpenDate,IndustryCode,AcctType,
                  AcctRate,Code1,Code2,HighCredit,Balance},
                 $.DeclareData.SubKey1)),

        IF(~Std.File.FileExists($.DeclareData.SubKey2),
           OUTPUT(s2,
                  {PersonID,Account,OpenDate,IndustryCode,AcctType,
                   AcctRate,Code1,Code2,HighCredit,Balance},
                  $.DeclareData.SubKey2)),

        IF(~Std.File.FileExists($.DeclareData.SubKey3),
           OUTPUT(s3,
                  {PersonID,Account,OpenDate,IndustryCode,AcctType,
                   AcctRate,Code1,Code2,HighCredit,Balance},
                  $.DeclareData.SubKey3)),

        IF(~Std.File.FileExists($.DeclareData.SubKey4),
           OUTPUT(s4,
                  {PersonID,Account,OpenDate,IndustryCode,AcctType,
                   AcctRate,Code1,Code2,HighCredit,Balance},
                  $.DeclareData.SubKey4)));

BldIDX := PARALLEL(
        IF(~Std.File.FileExists($.DeclareData.SubIDX1),
           BUILDINDEX(i1)),

        IF(~Std.File.FileExists($.DeclareData.SubIDX2),
           BUILDINDEX(i2)),

        IF(~Std.File.FileExists($.DeclareData.SubIDX3),
           BUILDINDEX(i3)),

        IF(~Std.File.FileExists($.DeclareData.SubIDX4),
           BUILDINDEX(i4)));

SEQUENTIAL(BldDat,BldIDX);
```

Esse código compila sequencialmente os arquivos lógicos ao usar subconjuntos de registros do dataset Accounts e gravar esses registros para arquivos em disco. Depois que os arquivos lógicos existem fisicamente, as ações BUILDINDEX gravam as chaves de conteúdo em disco.

Um ponto interessante para esse código é o uso da função Std.File.FileExists para detectar se esses arquivos já foram criados. O código na próxima seção também usa a função Std.File.SuperFileExists para detectar se o superarquivo já foi criado e criá-lo somente caso ele ainda não exista. Essa técnica permite que o código de exemplo neste artigo seja executado de modo correto, independentemente do usuário já ter passado pelos exemplos ou não.

Criando uma Superchave

O processo de criação de uma superchave é exatamente o mesmo daquele usado para criar um superarquivo. O código a seguir (contido no arquivo SuperKey2.ECL) cria uma superchave e adiciona as duas primeiras chaves de conteúdo a ela:

```
IMPORT $;
IMPORT Std;

SEQUENTIAL(
   IF(~Std.File.SuperFileExists($.DeclareData.AcctSKname),
      Std.File.CreateSuperFile($.DeclareData.AcctSKname)),
   Std.File.StartSuperFileTransaction(),
   Std.File.ClearSuperFile($.DeclareData.AcctSKname),
   Std.File.AddSuperFile($.DeclareData.AcctSKname,$.DeclareData.SubIDX1),
   Std.File.AddSuperFile($.DeclareData.AcctSKname,$.DeclareData.SubIDX2),
   Std.File.FinishSuperFileTransaction());
```

Utilizando uma Superchave

Depois de ter uma superchave pronta para usar, é possível usá-la nos JOINs de meia chave, como demonstrado neste código (contido no arquivo SuperKey3.ECL):

```
IMPORT $;

r1 := RECORD
  $.DeclareData.Layout_Person;
  $.DeclareData.Layout_Accounts;
END;

r1 Xform($.DeclareData.Person.FilePlus L, $.DeclareData.AcctSK R) := TRANSFORM
  SELF := L;
  SELF := R;
END;

J3 := JOIN($.DeclareData.Person.FilePlus(PersonID BETWEEN 1 AND 100),
           $.DeclareData.AcctSK,
           LEFT.PersonID=RIGHT.PersonID,
           Xform(LEFT,RIGHT));

OUTPUT(J3,ALL);
```

Manutenção de superchaves

Uma superchave é simplesmente um superarquivo cujos subarquivos componentes são chaves de conteúdo. Assim sendo, compilar e manter uma superchave constitui exatamente o mesmo processo já descrito no tópico *Como criar e manter superarquivos* . A única diferença significa está na maneira pela qual você cria os subarquivos de componente, cujo processo já foi descrito no tópico *Como usar chaves ECL (arquivos INDEX)* .

Trabalhando com Roxie

Visão Geral do Roxie

Vamos começar com algumas definições:

Thor	Um cluster do HPCC projetado especificamente para manipular processos com uma quantidade enorme de dados (ETL). Essa é uma ferramenta de preparo de dados de back-office e não é voltada para consultas em nível de produção para o usuário final. Consulte os manuais de operação do HPCC para obter a documentação completa.
Roxie	Um cluster do HPCC projetado especificamente para atender consultas padrão, proporcionando uma taxa de produção de mais de mil respostas por segundo (a taxa de respostas real para qualquer consulta depende, obviamente, de sua complexidade). Essa é uma ferramenta de nível de produção projetada para aplicações de missão crítica. Consulte os manuais de operação do HPCC para obter a documentação completa.
hThor	Uma plataforma de P&D projetada para desenvolvimento interativo e iterativo e testes de consultas Roxie. Esse não é um cluster separado, mas uma implementação "acumulada" do ECL Agent e Thor. Consulte os manuais de operação do HPCC para obter a documentação completa.

Thor

Os clusters Thor são usados para realizar todo o trabalho de preparo "pesado" do dados para processar os dados brutos em formatos padrão. Após a conclusão deste processo, os usuários finais podem consultar esses dados padronizados para recolher informações reais. No entanto, usuários finais normalmente querem esses resultados "imediatamente ou para ontem" – e normalmente mais de um usuário final quer o resultado ao mesmo tempo. A plataforma Thor só funciona em uma consulta por vez, o que a torna seu uso impraticável por usuários finais, e é essa a razão pela qual a plataforma Roxie foi criada.

Roxie

Clusters Roxie podem processar milhares de usuários finais simultaneamente e proporcionar a todos eles a percepção dos resultados que desejam obter "imediatamente ou para ontem". Eles fazem isso ao permitir que os usuários finais executem apenas consultas padrão e pré-compiladas que tenham sido desenvolvidas especificamente para serem usadas pelo usuário final no cluster Roxie. Normalmente, essas consultas usam índices e, dessa forma, oferecem um desempenho extremamente rápido. No entanto, usar o cluster Roxie como uma ferramenta de desenvolvimento é impraticável, já que todas as suas consultas precisam ser pré-compiladas e os dados usados precisam ter sido previamente implementados. Dessa forma, o processo de testes e desenvolvimento de consultas é realizado usando o Doxie.

hThor

hThor não é um cluster independente, mas uma instância do ECL Agent (que opera em um servidor único) que emula a operação de um cluster Roxie. Assim como as consultas Thor, as consultas hThor são compiladas a cada execução. As consultas hThor acessam os dados diretamente a partir dos discos de um cluster Thor sem interferir nas operações do Thor. Isso faz dele uma ferramenta adequada para criar consultas voltadas para uso em um cluster Roxie.

Como Desenvolver Consultas Roxie

Para começar a criar consultas para uso em clusters Roxie, o primeiro passo é decidir que dados consultar e como indexar esses dados para que usuários finais vejam seu resultado em tempo mínimo. O processo de colocação dos

dados em sua forma mais útil e de indexação é realizado em um cluster Thor. Os artigos anteriores sobre indexação e superarquivos deverão orientá-lo sobre como fazer isso.

Depois que os dados estão prontos para uso, é possível criar sua consulta. As consultas para clusters Roxie sempre contêm no mínimo uma ação – normalmente, uma OUTPUT simples para retornar o conjunto de resultados.

As consultas Roxie usam um SOAP (Protocolo Simples de Acesso a Objetos) ou interface JSON (Notação de Objeto em JavaScrip) para "especificar" valores de dados. Os valores especificados pela interface se resumem nas definições com o serviço de fluxo de trabalho STORED. Seu código ECL pode então usar essas definições para determinar os valores especificados e retornar o resultado adequado para o usuário final.

Aqui está um exemplo simples de uma estrutura de consulta Roxie (contida no arquivo RoxieOverview1.ECL):

```
IMPORT $;

EXPORT RoxieOverview1 := FUNCTION

STRING30 lname_value := '' : STORED('LastName');
STRING30 fname_value := '' : STORED('FirstName');

IDX  := $.DeclareData.IDX__Person_LastName_FirstName;
Base := $.DeclareData.Person.FilePlus;

Fetched := IF(fname_value = '',
              FETCH(Base, IDX(LastName=lname_value), RIGHT.RecPos),
              FETCH(Base, IDX(LastName=lname_value, FirstName=fname_value), RIGHT.RecPos));

RETURN OUTPUT(CHOOSEN(Fetched,2000));

END;
```

Observe que FUNCTION não recebe parâmetros. Em vez disso, as definições lname_value e fname_value possuem o serviço de fluxo de trabalho STORED que fornece nomes de armazenamento. A interface SOAP/JSON usa os nomes de armazenamento para especificar os valores, uma vez que a opção STORED abre um espaço de armazenamento na tarefa onde a interface pode colocar os valores a serem especificados para o serviço.

Esse código usa a função FETCH por ser o exemplo mais simples do uso de um INDEX no ECL. Geralmente, as consultas Roxie usam operações JOIN de meia chave com chaves de conteúdo (o artigo *Consultas Complexas Roxie* aborda esse problema). Note que o OUTPUT possui CHOOSEN, um exemplo simples de como se certificar de que é preciso limitar a quantidade máxima de dados que pode ser retornada da consulta para uma quantidade "cabível" – não faz tanto sentido ter uma consulta Roxie que possa acabar retornando 10 bilhões de registros para o PC de um usuário final (qualquer um que realmente precise de tantos dados deve usar o Thor, e não o Roxie).

Testando Consultas

Depois de gravar sua consulta, você, naturalmente, vai querer testá-la. É aí que o hThor entra em cena. hThor é um sistema de testes interativos que pode ser usado antes da implementação de consultas no Roxie. A maneira mais fácil de descrever o processo é acompanhá-lo usando essa consulta simples como exemplo.

1. Abra o arquivo Samples\ProgrammersGuide\RoxieOverview1.ECL

Agora você está pronto para publicar essa consulta no hThor.

2. Selecione "hthor" na lista suspensa Target

3. Clique na seta para baixo no botão Submit e selecione Compile

4. Abra a workunit compilada e selecione a aba ECL Watch

5. Pressione o botão Publish

Abra a página do ECL Watch (não use o ECL IDE – abra a página no Internet Explorer). Abra a página do ECL Watch (não use o ECL IDE – abra a página no Internet Explorer). O IP do ECL Watch é o mesmo IP usado para configurar o ECL IDE para acessar o ambiente no qual está trabalhando. A porta é 8010.

6. Clique em System Servers (Servidores do Sistema) na seção Topology

7. Localize a seção **ESP Servers**

8. Clique no link do nome do ESP Server para exibir sua lista de serviços e suas portas

9. Note o número de porta ao lado do tipo de serviço **wsecl** (normalmente é 8002, mas pode ser definida para um valor diferente)

Após ter descoberto o IP e a porta para seu serviço wsecl (o serviço que faz com que o hthor "finja" ser um Roxie), é possível acessar e executar a consulta.

10. Uma maneira fácil de fazer isso é clicar com o botão direito no link wsecl e abri-lo em uma nova aba ou janela (ou você pode editar a barra de endereços do Internet Explorer para indicar o IP:port correto.

11. Pressione a tecla Enter

Uma caixa de login será exibida – seu ID e senha são os mesmos usados no ECL IDE. Depois de fazer login, será exibida uma lista do grupo de consultas em forma de árvore à esquerda.

12. Clique na ramificação hthor

Uma lista de todas as consultas publicadas no hthor é exibida na árvore. Neste caso, há apenas uma.

13. Clique na ramificação RoxieOverview1.1

É exibida uma página da Web contendo dois controles de entrada e um botão **Submit** .

14. Digite um sobrenome do grupo de sobrenomes usado pelo código no GenData.ECL para gerar os arquivos de dados para este *Guia do Programador (Progammers's Guide)*

COOLING é um bom exemplo para ser usado. Observe que por se tratar de um exemplo extremamente simples, será necessário digitá-lo em totalmente caixa alta ALL CAPS; caso contrário, FETCH irá falhar ao tentar localizar os registros correspondentes (isso se deve apenas à simplicidade desse código ECL e não a alguma falta inerente no sistema).

15. Pressione o botão **Submit** .

As consultas são pré-compiladas ao serem publicadas, de forma que após um segundo, você deve ver um resultado em XML contendo 1.000 registros.

Implantando Consultas para o Roxie

Depois de ter realizado testes suficientes no hThor , para garantir que a consulta funciona como o esperado, o único passo necessário é implementá-la e testá-la também no Roxie (apenas para estar totalmente seguro de que tudo está funcionando da maneira correta). Após ter realizado o teste no Roxie, é possível informar aos usuários que a consulta está disponível para uso.

O processo de implementação do Roxie é feito da mesma maneira que foi realizado para o hThor, exceto pelo fato de que é necessário definir uma lista de Destino para o Roxie.

Depois de implementar a consulta, é possível testá-la da mesma maneira que foi testada no hThor, porém o novo serviço aparecerá em seu Roxie na lista de árvore.

Consultas habilitadas em SOAP

As consultas destinadas ao uso no Roxie devem primeiramente ser habilitadas para SOAP. O código ECL necessário para se fazer isso é o serviço de fluxo de trabalho STORED. As consultas Roxie devem estar contidas na estrutura FUNCTION ou devem simplesmente ser executáveis.

Chave ECL para SOAP

O requisito do código ECL para os parâmetros de entrada ativados no SOAP é o uso do serviço do fluxo de trabalho STORED. Cada nome do parâmetro SOAP deve constituir no nome STORED para uma definição ECL. O serviço de fluxo de trabalho STORED cria um espaço de armazenagem de dados na tarefa que a interface do SOAP usa para preencher os dados “especificados”. O código ECL simplesmente usa essas definições STORED para determinar se os dados foram especificados a partir daquele “parâmetro” e em que esses dados consistem. O tipo de dados do parâmetro SOAP especificado está implícito pela definição do STORED.

Para o exemplo de código a seguir, você deve criar duas definições com os nomes STORED duplicando o nome do parâmetro SOAP, desta forma:

```
STRING30 lname_value := '' : STORED('LastName');

STRING30 fname_value := '' : STORED('FirstName');
```

Por padrão, estes ficarão em branco e o serviço do fluxo de trabalho STORED abre um espaço na workunit para armazenar o valor. O Enterprise Service Platform (ESP) lida com as tarefas da interface SOAP adicionando os valores adequados ao espaço de armazenamento criado pelo STORED. Diante disso, o código ECL precisa apenas usar as definições (neste caso Lname e Fname) para concluir a consulta. Isso simplifica bastante o lado ECL da equação.

Juntando Tudo

Após entender os requisitos, a consulta ativada no SOAP terá a seguinte forma (contida no SOAPenabling.ECL):

```
IMPORT ProgrammersGuide.DeclareData AS ProgGuide;

EXPORT SOAPenabling() := FUNCTION
  STRING30 lname_value := '' : STORED('LastName');
  STRING30 fname_value := '' : STORED('FirstName');
  IDX  := ProgGuide.IDX__Person_LastName_FirstName;
  Base := ProgGuide.Person.FilePlus;
  Fetched := IF(fname_value = '',
           FETCH(Base,IDX(LastName=lname_value),RIGHT.RecPos),
                FETCH(Base,IDX(LastName=lname_value,
      FirstName=fname_value),RIGHT.RecPos));
  RETURN OUTPUT(CHOOSEN(Fetched,2000));
END;
```

Técnicas de Consultas Complexas

As técnicas de codificação ECL usadas nas consultas Roxie podem ser bastante complexas, usando múltiplas chaves, chaves de contéudo, HALF-KEYED JOINS, a função KEYDIFF e vários outros recursos da linguagem ECL. No entanto, todas essas técnicas compartilham do mesmo objetivo: maximizar o desempenho da consulta para que seu resultado seja entregue do modo mais eficiente possível, maximizando a taxa de produtividade de transação total para o Roxie que atende a consulta.

Seleção de Chave Baseada na Entrada

Tudo começa com a arquitetura de seus dados e as chaves compiladas a partir deles. Normalmente, um único dataset teria múltiplos índices para fornecer vários métodos de acesso aos dados. Dessa forma, uma das principais técnicas usadas em consultas Roxie é detectar qual dos conjuntos de possíveis valores foi especificado para a consulta e, baseado nesses valores, escolher o INDEX correto a ser usado.

A base para detectar quais valores foram especificados para a consulta é determinada pelos atributos STORED definidos para receber os valores especificados. A interface SOAP preenche automaticamente esses atributos com quaisquer valores que tenham sido especificados para a consulta. Isso significa que o código de consulta precisa apenas interrogar esses parâmetros quanto a presença de valores diferentes dos de seus padrões.

Este exemplo demonstra a técnica:

```
IMPORT $;
EXPORT PeopleSearchService() := FUNCTION
  STRING30 lname_value := '' : STORED('LastName');
  STRING30 fname_value := '' : STORED('FirstName');
  IDX  := $.IDX__Person_LastName_FirstName;
  Base := $.Person.FilePlus;

  Fetched := IF(fname_value = '',
               FETCH(Base,IDX(LastName=lname_value),RIGHT.RecPos),
               FETCH(Base,IDX(LastName=lname_value,FirstName=fname_value),RIGHT.RecPos));
  RETURN OUTPUT(CHOOSEN(Fetched,2000));
END;
```

Essa consulta é escrita supondo que o parâmetro LastName sempre será especificado, de forma que IF precisa apenas detectar se um FirstName também foi inserido pelo usuário. Em caso afirmativo, o filtro no parâmetro de índice para FETCH precisa apenas filtrar esse valor, caso contrário o FETCH precisa apenas filtrar o índice com o valor LastName.

Esse código pode ser escrito de várias maneiras. Aqui está uma alternativa:

```
IMPORT $;
EXPORT PeopleSearchService() := FUNCTION
  STRING30 lname_value := '' : STORED('LastName');
  STRING30 fname_value := '' : STORED('FirstName');
  IDX  := $.IDX__Person_LastName_FirstName;
  Base := $.Person.FilePlus;
  IndxFilter := IF(fname_value = '',
                   IDX.LastName=lname_value,
                   IDX.LastName=lname_value AND IDX.FirstName=fname_value);
  Fetched := FETCH(Base,IDX(IndxFilter),RIGHT.RecPos);
  RETURN OUTPUT(CHOOSEN(Fetched,2000));
END;
```

Neste exemplo, IF apenas compila a expressão de filtro correta a ser usada por FETCH. O uso desta forma facilita a leitura e a manutenção do código ao removeras múltiplas formas possíveis da lógica de filtro da função que a utiliza.

KEYED JOINS

Embora a função FETCH tenha sido projetada especificamente para acesso indexados aos dados, na prática, a operação HALF-KEYED JOINs é normalmente usada em consultas Roxie. A principal razão para isso é a flexibilidade oferecida por JOIN.

As vantagens de uso de operações KEYED JOINs em qualquer consulta são discutidas de maneira mais detalhada no tópico *Como usar chaves ECL (arquivos INDEX)* . Essas vantagens garantem benefícios tremendos para as consultas Roxie. Devido à natureza do Roxie, a melhor vantagem de JOINs com chave vem do uso de HALF-KEYED JOINS que utilizam índices payload (acabando com a necessidade de operações FETCH adicionais).

Limitando a saída

Uma das principais considerações para desenvolver uma consulta Roxie é a quantidade de dados que pode acabar sendo retornada da consulta. Uma vez que as operações JOIN podem acabar resultando em enormes datasets, deve-se ter o cuidado de limitar o número de registros que qualquer consulta possa retornar para um número razoável de acordo com o tipo específico de consulta. Aqui estão algumas técnicas para se fazer isso:

* As funções CHOOSEN e LIMIT devem ser usadas para limitar leituras de índice para o número máximo.
* JOINs com chave devem usar a opção ATMOST, KEEP, ou LIMIT.
* Quando um dataset secundário aninhado é definido, ele deve ter a opção MAXCOUNT definida no campo DATASET secundário da estrutura RECORD, e o código que compila o dataset secundário aninhado deve usar CHOOSEN com um valor que corresponda exatamente ao MAXCOUNT.

Todas essas técnicas ajudarão a garantir que, quando o usuário final espera obter cerca de dez resultados, ele não acabe com dez milhões.

SOAPCALL do Thor para o Roxie

Após ter testado e implementado no Roxie as consultas habilitadas para SOAP, você precisa conseguir usá-las. Muitas consultas do Roxie podem ser iniciadas através de alguma interface de usuário projetada especificamente para possibilitar que usuários finais insiram critérios de busca e obtenham resultados, um por vez. No entanto, às vezes é necessário recuperar dados em modo de lote, onde a mesma consulta é executada uma vez em relação a cada registro de um dataset. Isso faz do Thor um bom candidato à plataforma de solicitação usando o SOAPCALL.

Um registro de entrada, um conjunto de registros de saída (One Record Input, Record Set Return)

Este código de exemplo (contido no arquivo Soapcall1.ECL) aciona o serviço previamente implementado no artigo **Visão geral do Roxie** (você precisará alterar o atributo IP neste código para o IP e porta adequados do Roxie para o qual será implementado):

```
IMPORT $;

OutRec1 := $.DeclareData.Layout_Person;
RoxieIP := 'http://127.0.0.1:8002/WsEcl/soap/query/roxie/roxieoverview1.1';
svc     := 'RoxieOverview1.1';

InputRec := RECORD
  STRING30 LastName  := 'KLYDE';
  STRING30 FirstName := '';
END;
//1 rec in, recordset out
ManyRec1 := SOAPCALL(RoxieIP,
                     svc,
                     InputRec,
                     DATASET(OutRec1));
OUTPUT(ManyRec1);
```

Este exemplo mostra como você faria um SOAPCALL para o serviço especificando-o como um conjunto único de parâmetros para recuperar apenas os registros que estão relacionados ao conjunto de parâmetros especificados. O serviço recebe um conjunto único de dados de entrada e retorna apenas aqueles registros que atendam a esses critérios. O resultado esperado dessa consulta é um conjunto retornado dos 1.000 registros cujo campo LastName contém "KLYDE".

Um conjunto de registros de entrada, um conjunto de registros de saída. (Record Set Input, Record Set Return)

Este próximo código de exemplo (contido no arquivo Soapcall2.ECL) também aciona o mesmo serviço que o exemplo anterior (lembre-se, você precisará alterar o atributo IP neste código para o IP e porta adequados do Roxie para o qual foi implementado):

```
IMPORT $;

OutRec1 := $.DeclareData.Layout_Person;
RoxieIP := 'http://127.0.0.1:8002/WsEcl/soap/query/roxie/roxieoverview1.1';
svc     := 'RoxieOverview1.1';
//recordset in, recordset out
InRec := RECORD
  STRING30 LastName {XPATH('LastName')};
```

```
  STRING30 FirstName{XPATH('FirstName')};
END;

InputDataset := DATASET([{'TRAYLOR','CISSY'},
                         {'KLYDE','CLYDE'},
                         {'SMITH','DAR'},
                         {'BOWEN','PERCIVAL'},
                         {'ROMNEY','GEORGE'}],Inrec);

ManyRec2 := SOAPCALL(InputDataset,
      RoxieIP,
      svc,
      Inrec,
      TRANSFORM(LEFT),
      DATASET(OutRec1),
      ONFAIL(SKIP));
OUTPUT(ManyRec2);
```

Este exemplo especifica um dataset que contém múltiplos conjuntos de parâmetros nos quais o serviço vai operar, retornando um grupo de registros único com todos os registros retornados por cada conjunto de parâmetros. Nesta forma, a função TRANSFORM permite que SOAPCALL opere como um PROJECT para produzir registros de entrada que proporcionam os parâmetros de entrada para o serviço.

O serviço opera em cada registro no dataset de entrada por vez, combinando os resultados de cada um deles em um conjunto único de resultados retornados. A opção ONFAIL indica que se houver qualquer tipo de erro, o registro deve simplesmente ser ignorado. O resultado esperado dessa consulta é um conjunto retornado de três registros para apenas os três registros que corresponderem aos critérios de entrada (CISSY TRAYLOR, CLYDE KLYDE, e PERCIVAL BOWEN).

Considerações sobre desempenho: PARALLEL

A forma do primeiro exemplo usa uma linha única como sua entrada. Quando apenas um URL for especificado, SOAPCALL envia a solicitação para um URL e aguarda uma resposta. Se vários URLs forem especificados, o SOAPCALL envia uma solicitação para o primeiro URL na lista, aguarda uma resposta, envia uma solicitação para o segundo URL, e assim por diante. A opção PARALLEL controla a simultaneidade, de forma que se PARALLEL (*n*) for especificada, as solicitações são enviadas simultaneamente de cada nó Thor com até *n* solicitações em voo de uma vez a partir de cada nó.

A forma do segundo exemplo usa um dataset como sua entrada. Quando apenas um URL é especificado, o comportamento padrão é enviar duas solicitações com a primeira e segunda linhas simultaneamente, aguardar uma resposta, enviar a terceira linha e assim por diante em todo o dataset, com até duas solicitações em voo por vez. Se PARALLEL (*n*) for especificada, ela envia *n* solicitações com as primeiras *n* linhas simultaneamente de cada nó Thor e assim por diante, com até *n* solicitações em voo de uma vez a partir cada nó.

Em um mundo ideal, você especificaria um valor PARALLEL que multiplica no mínimo o número de URLs do Roxie, de forma que cada host disponível possa funcionar simultaneamente. Além disso, se estiver usando um dataset como entrada, talvez você queira tentar um valor várias vezes o número de URLs. No entanto, isso pode causar contenção de rede (tempo-limite e conexões perdidas) se for definido como muito elevado.

A opção PARALLEL deve ser adicionada ao código de ambos os exemplos anteriores para ver que efeito os diferentes valores podem ter em seu ambiente.

Considerações sobre desempenho: MERGE

A opção MERGE controla o número de linhas por solicitação para a forma que utiliza um dataset (MERGE não se aplica às formas de SOAPCALL que usam uma única linha como entrada). Se MERGE(*m*) for especificada, cada solicitação contém até *m* linhas em vez de uma única linha.

Se a simultaneidade (configuração da opção PARALLEL) for inferior ou igual ao número de URLs, então cada URL normalmente verá apenas uma solicitação por vez (supondo que todos os hosts operem em uma velocidade aproximada). Nesse caso, você pode querer escolher um valor máximo de MERGE que o host e a rede possam suportar: um valor muito alto e uma solicitação enorme podem cancelar ou desacelerar um serviço, enquanto um valor muito baixo aumenta desnecessariamente a sobrecarga enviando muitas solicitações pequenas no lugar de outras maiores. Se a simultaneidade for superior ao número de URLs, então cada URL terá múltiplas solicitações por vez e essas considerações ainda serão aplicadas.

Supondo que o host processe uma única solicitação em série, há uma consideração adicional. Você precisa assegurar que o valor MERGE seja inferior ao número de linhas no dataset a fim de assegurar que você esteja utilizando a paralelização dos hosts. Se o valor MERGE for superior ou igual ao número de linhas de entrada, você então envia todo o dataset de entrada em uma única solicitação e o host processará as linhas em série.

Você deve adicionar a opção MERGE ao código do segundo exemplo para ver que efeito os diferentes valores podem ter em seu ambiente.

Um Exemplo do Mundo Real

Um cliente pediu ajuda para resolver um problema – como comparar duas strings e determinar se a primeira contém todas as palavras contidas na segunda, em qualquer ordem, quando há um número indeterminado de palavras em cada string. Esse é um problema bastante simples no ECL. Usar JOIN e ROLLUP seria uma abordagem, ou ainda consultas de child datasets aninhadas (não compatíveis com o Thor no momento da solicitação de ajuda, embora talvez a situação possa ter mudado no momento em que você estiver lendo isso). Todo o código a seguir está contido no arquivo Soapcall3.ECL.

A primeira necessidade seria criar uma função para extrair todas as palavras discretas de uma string. Esse é o tipo de job na qual a função PARSE se sobressai. E é exatamente isso que o código faz:

```
ParseWords(STRING LineIn) := FUNCTION
  PATTERN Ltrs := PATTERN('[A-Za-z]');
  PATTERN Char := Ltrs | '-' | '\'';
  TOKEN   Word := Char+;
          ds   := DATASET([{LineIn}],{STRING line});
  RETURN PARSE(ds,line,Word,{STRING Pword := MATCHTEXT(Word)});
END;
```

Essa FUNCTION (contida no Soapcall3.ECL) recebe uma string de entrada e produz um resultado de conjunto de registros de todas as palavras contidas nessa string. Ela define um atributo PATTERN (Char) de caracteres permitidos em uma palavra como o conjunto de todas as letras maiúsculas e minúsculas (definida por PATTERN Ltrs), o hífen e o apóstrofo. Qualquer outro caractere além desses será ignorado.

Em seguida, ela define uma Palavra como um ou mais caractere de padrão Char permitido. Esse padrão é definido como um TOKEN de forma que apenas a correspondência da palavra completa seja retornada, e não todas as correspondências alternativas possíveis (isto é, retornando apenas o SOAP, em vez de SOAP, SOA, SO, e S – todas as correspondências alternativas que um PATTERN poderia gerar).

Esse atributo DATASET embutido de um registro (ds) cria o "arquivo" de entrada a ser processado pela função PARSE, produzindo o conjunto de registro de resultado com todas as palavras discretas da string de entrada.

Em seguida, precisamos de uma consulta Roxie para comparar as duas strings (também contidas no arquivo Soapcall3.ECL):

```
EXPORT Soapcall3() := FUNCTION
  STRING UID     := '' : STORED('UIDstr');
  STRING LeftIn  := '' : STORED('LeftInStr');
  STRING RightIn := '' : STORED('RightInStr');
  BOOLEAN TokenMatch := FUNCTION
    P1 := ParseWords(LeftIn);
```

```
    P2 := ParseWords(RightIn);
    SetSrch := SET(P1,Pword);
    ProjRes := PROJECT(P2,
                       TRANSFORM({BOOLEAN Fnd},
                                 SELF.Fnd := LEFT.Pword IN SetSrch));
    AllRes  := DEDUP(SORT(ProjRes,Fnd));
    RETURN COUNT(AllRes) = 1 AND AllRes[1].Fnd = TRUE;
  END;
  RETURN OUTPUT(DATASET([{UID,TokenMatch}],{STRING UID,BOOLEAN res}));
END;
```

Esta consulta espera receber três porções de dados: uma string com um identificador para comparação (para finalidades de contexto no resultado) e as duas strings cujas palavras serão comparadas.

A FUNCTION passa as strings de entrada para a função ParseWords a fim de criar dois conjuntos de registros de palavras a partir destas strings. A função SET então redefine o primeiro conjunto de registros como um SET para que o operador IN possa ser usado.

A operação PROJECT realiza todo o trabalho real. Ela especifica cada palavra por vez a partir da segunda string de entrada até sua função TRANSFORM embutida, que produz um resultado Booleano para essa palavra – TRUE ou FALSE está presente no conjunto de palavras da primeira string de entrada ou não?

Para determinar se todas as palavras na segunda string estavam contidas na primeira, a função SORT/DEDUP classifica todos os valores Booleanos resultantes e depois remove todas as entradas em duplicidade. Haverá apenas um ou dois registros restantes: um TRUE e um FALSE, ou um único registro TRUE ou FALSE.

A expressão RETURN detecta qual das três situações ocorreu. Dois registros restantes indicam que algumas palavras, e não todas, estavam presentes. Um único registro indica a presença de todas ou de nenhuma palavra, e se o valor desse registro for TRUE, significa que todas as palavras estavam presentes e a FUNCTION retorna TRUE. Todos os outros casos retornam como FALSE.

O OUTPUT usa um DATASET embutido de um registro para formatar o resultado. O identificador especificado é novamente especificado juntamente com o resultado Booleano da comparação. O identificador se torna importante quando a consulta é acionada múltiplas vezes no Roxie para processar por meio de um dataset de strings para comparar em um modo de lote, uma vez que os resultados podem não ser retornados na mesma ordem que os registros de entrada. Se fosse usado apenas de forma interativa, esse identificador não seria necessário.

Depois de ter salvo a consulta no Repositório, é possível testá-la com o hThor e/ou implementá-la no Roxie (o hThor pode ser usado para testes, mas o Roxie é muito mais rápido em se tratando de produção). De qualquer forma, você pode usar SOAPCALL para acessá-lo dessa forma (a única diferença seria o IP e a porta de destino para a consulta (contidas no Soapcall4.ECL):

```
RoxieIP := 'http://127.0.0.1:8002/WsEcl/soap/query/roxie/soapcall3.1'; //Roxie
svc     := 'soapcall3.1';

InRec := RECORD
  STRING UIDstr{XPATH('UIDstr')};
  STRING LeftInStr{XPATH('LeftInStr')};
  STRING RightInStr{XPATH('RightInStr')};
END;
InDS := DATASET([
   {'1','the quick brown fox jumped over the lazy red dog','quick fox red dog'},
   {'2','the quick brown fox jumped over the lazy red dog','quick fox black dog'},
   {'3','george of the jungle lives here','fox black dog'},
   {'4','fred and wilma flintstone','fred flintstone'},
   {'5','yomama comeonah','brake chill'} ],InRec);

RS := SOAPCALL(InDS,
               RoxieIP,
               svc,
```

```
              InRec,
              TRANSFORM(LEFT),
              DATASET({STRING UIDval{XPATH('uid')},BOOLEAN CompareResult{XPATH('res')}}));

OUTPUT(RS);
```

É óbvio que **você deve primeiramente alterar o IP e a porta neste código para os valores corretos para seu ambiente**. O IP e a porta adequados para uso podem ser localizados consultando a página Servidores do Sistema do seu ECL Watch. Se o destino for o Doxie (também conhecido como ECL Agent ou hthor), use o IP do ESP Server do Thor e a porta para seu serviço wsecl. Se o destino for o Roxie use o IP do ESP Server do Roxie e a porta para seu serviço wsecl. É possível que ambos ESP Servers estejam na mesma caixa. Se for este o caso, a diferença será apenas na atribuição de portas de cada um.

O segredo dessa consulta SOAPCALL está na estrutura RECORD InRec com suas definições XPATH. Elas precisam corresponder exatamente aos nomes da parte e aos nomes STORED dos atributos de recepção de parâmetro da consulta (NB que eles distinguem maiúsculas de minúsculas, uma vez que o XPATH é XML e XML sempre faz essa distinção). Isso é o que mapeia os campos de dados de entrada pela interface SOAP para os atributos da consulta.

Esse SOAPCALL recebe um conjunto de registros como entrada e produz um conjunto de registros como seu resultado, tornando-a bastante semelhante ao segundo exemplo acima. Uma pequena mudança em relação ao exemplo anterior deste tipo é o uso do TRANSFORM abreviado em vez da função TRANSFORM completa. Observe também que XPATH para o primeiro campo na estrutura RECORD embutida do parâmetro do DATASET contém um "uid" em minúsculas enquanto se refere obviamente ao campo OUTPUT da consulta denominado "UID" – o XML retornado do serviço SOAP usa nomes de tags em minúsculas para os campos de dados retornados.

Ao executar isso, será exibido um resultado TRUE para os registros um e quatro, e FALSE para todos os demais.

Controlando Consultas Roxie

Há várias funções ECL projetadas especificamente para ajudar a otimizar as consultas para execução no Roxie. Esses incluem PRELOAD, ALLNODES, THISNODE, LOCAL, e NOLOCAL Entender como todas essas funções operam conjuntamente pode fazer uma enorme diferença no desempenho de suas consultas Roxie.

Como os grafos são executados

Gravar consultas eficientes para o Roxie ou o Thor pode exigir a compreensão sobre como os diferentes clusters operam. Isso gera três perguntas:

Como o grafo é executado: em um nó único ou em todos os nós em paralelo?

Como os datasets são acessados por cada nó executando o grafo: apenas as partes que são locais ao nó ou todas as partes em todos os nós?

Uma operação é coordenada com a mesma operação em outros nós, ou cada nó opera de modo independente?

As consultas "normalmente" são executadas em cada tipo de cluster da seguinte maneira:

Thor	Os grafos são executados em múltiplos nós filhos em paralelo.
	As leituras de índice/disco são feitas localmente por cada nó filho.
	Todos os demais acessos ao disco (FETCH, keyed JOIN, etc.) são feitos de forma eficiente em todos os nós.
	A coordenação com operações em outros nós é controlada pela presença ou ausência da opção LOCAL.
	Não há suporte para consultas secundárias (isso pode ser mudado nas próximas versões).
hthor	Os grafos executados em um nó único do ECL Agent (servidor Roxie).
	Todas as partes do dataset/índice são acessadas pelo acesso direto ao disco rígido do nó com os dados – sem outra interação com os outros nós.
	Consultas secundárias sempre são executadas no mesmo nó como principais.
Roxie	Os grafos executados em um nó único (servidor Roxie).
	Todas as partes do dataset/índice são acessadas pelo acesso direto ao disco rígido do nó com os dados – sem outra interação com os outros nós.
	Consultas secundárias podem ser executadas em um único nó do Agent em vez de um nó de servidor Roxie.

ALLNODES vs. THISNODE

No Roxie, os grafos são executados em um nó único do servidor Roxie, a menos que a função ALLNODES() seja usada. ALLNODES() faz com que a parte da consulta incluída seja executada em todos os nós de agente em paralelo. Os resultados são calculados de forma independente em cada nó e depois são agregados, sem ordenação dos registros. Ela é geralmente usada para realizar um processamento remoto complexo que exige apenas o acesso ao índice local, reduzindo significativamente o tráfego de rede entre os nós.

Por padrão, tudo que estiver em ALLNODES() será executado em todos os nós; às vezes, porém, a consulta ALLNODES() exige alguma entrada ou argumentos que não devem ser executados em todos os nós – por exemplo, o melhor palpite anterior sobre os resultados, ou algumas informações que controlam a consulta paralela. A função THISNODE() pode ser usada nos elementos acerca que serão ser avaliados pelo nó atual.

Um uso típico seria:

```
bestSearchResults := ALLNODES(doRemoteSearch(THISNODE(searchWords),THISNODE(previousResults)))
```

Onde 'searchWords' e 'previousResults' são calculados de forma efetiva no nó atual e depois especificados como parâmetros para cada instância do doRemoteSearch(), sendo executado em paralelo em todos os nós.

LOCAL vs. NOLOCAL

A opção LOCAL disponível em várias funções (como JOIN, SORT, etc.) e as funções LOCAL() e NOLOCAL() controlam se os grafos executados em um determinado nó acessam todas as partes de um arquivo/índice ou apenas aquelas associadas ao nó específico (LOCAL). Geralmente, dentro do contexto ALLNODES(), busca-se apenas acessar as partes de índices locais a partir de um único nó, uma vez que cada nó está processando suas partes associadas de forma independente. Especificar que uma leitura de índice ou uma JOIN com chave seja LOCAL significa que apenas a parte local é usada em cada nó. A leitura local de um INDEX de parte única será avaliada no primeiro nó agent (ou o nó farmer se não estiver em ALLNODES).

A avaliação local pode ser especificada de duas maneiras:

1) Como uma operação de dataset:

```
LOCAL(MyIndex)(myField = searchField)
```

2) Como uma opção na operação:

```
JOIN(... ,LOCAL)
FETCH(... ,LOCAL)
```

A função LOCAL(*dataset*) faz com que toda operação no *dataset* acesse a chave/arquivo em nível local. Por exemplo,

```
LOCAL(JOIN(index1, index2,...))
```

será lido como index1 e index2 localmente. A regra é aplicada recursivamente até atingir um dos seguintes:

Uso da função NOLOCAL()

Um atributo não local – a operação permanece fora do local, mas itens secundários ainda são marcados como locais conforme necessário

GLOBAL() ou THISNODE() ou operação de fluxo de trabalho – já que será avaliada em um contexto diferente

Uso da função ALLNODES() (como em uma consulta secundária aninhada)

Observe que:

JOIN(x, LOCAL(index1)...) é tratado da mesma forma que JOIN(x, index1, ..., local).

LOCAL também é suportado como uma opção em um INDEX, mas a função LOCAL() é preferida uma vez que ela geralmente depende do contexto no qual um índice é usado para saber se o acesso deve ou não ser local.

Um atributo não local é suportado sempre que um atributo LOCAL for permitido – para substituir uma função LOCAL() delimitadora,

O uso de LOCAL para indicar que o acesso do dataset/chave é local não entra em conflito com o seu uso para controlar a coordenação de uma operação com outros nós, uma vez que não há operação que possivelmente seja coordenada com outros nós e que também acesse índices ou datasets.

Índices NOROOT

A função ALLNODES() é especialmente útil se houver mais de um índice co-distribuído em um determinado valor para que todas as informações relacionadas a um determinado valor de campo de chave estejam associadas ao mesmo nó. No entanto, os índices geralmente são classificados de forma global. **Adicionar a opção NOROOT em uma ação BUILD ou declaração INDEX indica que o índice não é classificado em nível global, e que não há índice de raiz para indicar qual parte do índice conterá uma determinada entrada.**

Bibliotecas de Consultas

Uma biblioteca de consulta é um conjunto de atributos, agrupados em uma unidade autocontida, que permite que o código seja compartilhado entre diferentes tarefas. Isso reduz o tempo necessário para implementar um conjunto de atributos, podendo reduzir a pegada de memória para o grupo de consultas no Roxie que usam a biblioteca. Também é possível atualizar uma biblioteca de consultas sem precisar reimplementar todas as consultas que a utilizam.

As bibliotecas de consulta não são compatíveis com o Thor, porém podem vir a ter compatibilidade futuramente.

Uma biblioteca de consulta é definida por duas estruturas – uma INTERFACE para definir os parâmetros do percurso e um MODULE que implementa a INTERFACE.

Definição da biblioteca INTERFACE

Para criar uma biblioteca de consulta, a primeira exigência é uma definição de seus parâmetros de entrada com uma estrutura de INTERFACE que recebe os parâmetros:

```
NamesRec := RECORD
  INTEGER1  NameID;
  STRING20  FName;
  STRING20  LName;
END;

FilterLibIface1(DATASET(namesRec) ds, STRING search) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;
```

Esse exemplo define a INTERFACE para uma biblioteca que usa duas entradas – um DATASET (com o formato de layout especificado) e uma STRING – e fornece a capacidade de gerar dois resultados de DATASET.

Para a maioria das consultas de biblioteca, pode ser preferível usar uma INTERFACE individual para definir os parâmetros de entrada. O uso de uma INTERFACE torna os parâmetros especificados mais claros e facilita a sincronização contínua da interface e da implementação. Este exemplo define a mesma interface de biblioteca mostrada acima:

```
NamesRec := RECORD
    INTEGER1  NameID;
    STRING20  FName;
    STRING20  LName;
END;

IFilterArgs := INTERFACE //defines passed parameters
  EXPORT DATASET(namesRec) ds;
  EXPORT STRING search;
END;
FilterLibIface2(IFilterArgs args) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;
```

Definições da biblioteca MODULE

Uma biblioteca de consulta é uma definição de estrutura de MODULE que implementa uma determinada definição de INTERFACE da biblioteca. Os parâmetros especificados ao MODULE precisam corresponder exatamente à definição da INTERFACE da biblioteca, e o MODULE precisa conter definições de atributos EXPORT compatíveis para cada um dos resultados especificados na INTERFACE da biblioteca. A opção LIBRARY na definição MODULE especi-

fica a INTERFACE da biblioteca que está sendo implementada. Este exemplo define uma implementação para as INTERFACEs acima:

```
FilterDsLib1(DATASET(namesRec) ds,
             STRING search) := MODULE,LIBRARY(FilterLibIface1)
  EXPORT matches := ds(Lname = search);
  EXPORT others := ds(Lname != search);
END;
```

e para a variedade que compõe uma INTERFACE como seu parâmetro único:

```
FilterDsLib2(IFilterArgs args) := MODULE,LIBRARY(FilterLibIface2)
  EXPORT matches := args.ds(Lname = args.search);
  EXPORT others := args.ds(Lname != args.search);
END;
```

Construindo uma Biblioteca Externa

Uma biblioteca de consulta pode ser interna ou externa. Uma biblioteca interna seria usada principalmente em consultas do hthor para testes e depuração antes da implementação no Roxie. Embora seja possível usar bibliotecas de consulta internas em consultas do Roxie, as vantagens são obtidas no uso da versão externa.

Uma biblioteca de consulta externa é criada pela ação BUILD, que compila a biblioteca de consulta em sua própria tarefa. O nome da biblioteca é o nome do trabalho associado à tarefa. Como consequência, a #WORKUNIT normalmente seria usada para dar um nome de trabalho significativo à tarefa, como neste exemplo:

```
#WORKUNIT('name','Ppass.FilterDsLib');
BUILD(FilterDsLib1);
```

Este código compila a biblioteca para a versão de parâmetros de INTERFACE do código acima:

```
#WORKUNIT('name','Ipass.FilterDsLib');
BUILD(FilterDsLib2);
```

O sistema mantém um catálogo das mais recentes versões de cada biblioteca de consulta que é atualizado sempre que uma biblioteca é compilada. O hthor usa isso para determinar bibliotecas de consulta ao executar uma consulta (como o Thor também fará quando for compatível com bibliotecas de consulta). O Roxie usa o mecanismo de alias de consulta da mesma maneira.

Usando uma biblioteca de consulta

A sintaxe para usar uma biblioteca de consulta é levemente diferente, dependendo de a biblioteca ser externa ou interna. No entanto, ambos os métodos usam a função LIBRARY.

A função LIBRARY retorna a implementação de um MODULE com os parâmetros adequados especificados para a instância na qual você deseja usar, que pode ser usada para acessar os atributos EXPORT da biblioteca.

Bibliotecas internas

Uma biblioteca interna gera o código de biblioteca como uma unidade separada, mas depois inclui essa unidade na tarefa de consulta. Ela não possui a vantagem de reduzir o tempo de compilação ou o uso de memória no Roxie, mas retém a estrutura da biblioteca, o que significa que alterações no código não podem afetar mais ninguém que esteja usando o sistema. Isso torna as bibliotecas um método de testes perfeito.

A sintaxe para uso de uma biblioteca de consultas interna simplesmente passa o nome do atributo MODULE da biblioteca dentro de uma chamada de função INTERNAL no primeiro parâmetro para a função LIBRARY, como neste exemplo (para a versão que não usa uma INTERFACE como parâmetro):

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                      NamesRec);
lib1 := LIBRARY(INTERNAL(FilterDsLib1),FilterLibIface1(NamesTable, 'Holliday'));
```

Neste caso, o resultado é um MODULE com dois atributos EXPORTed – correspondências e outros – que podem ser usados da mesma forma que qualquer outro MODULE, como neste exemplo:

```
OUTPUT(lib1.matches);
OUTPUT(lib1.others);
```

e o código muda para esse, para a variedade que usa uma INTERFACE:

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                      NamesRec);
SearchArgs := MODULE(IFilterArgs)
  EXPORT DATASET(namesRec) ds := NamesTable;
  EXPORT STRING search := 'Holliday';
END;
lib3 := LIBRARY(INTERNAL(FilterDsLib2),FilterLibIface2(SearchArgs));
OUTPUT(lib3.matches);
OUTPUT(lib3.others);
```

Bibliotecas Externas

Depois que a biblioteca é implementada como uma biblioteca externa (o uso da ação BUILD para criar a biblioteca é feito em uma tarefa separada), a função LIBRARY não exige mais o uso da função INTERNAL para especificar a biblioteca. Em vez disso, ela usa uma string constante que contém o nome da tarefa criada pelo BUILD como seu primeiro parâmetro, como este:

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                     NamesRec);
lib2 := LIBRARY('Ppass.FilterDsLib',FilterLibIface1(NamesTable, 'Holliday'));
OUTPUT(lib2.matches);
OUTPUT(lib2.others);
```

Ou, para a versão INTERFACE :

```
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                      NamesRec);

SearchArgs := MODULE(IFilterArgs)
  EXPORT DATASET(namesRec) ds := NamesTable;
  EXPORT STRING search := 'Holliday';
END;

lib4 := LIBRARY('Ipass.FilterDsLib',FilterLibIface2(SearchArgs));
OUTPUT(lib4.matches);
OUTPUT(lib4.others);
```

Um aviso rápido sobre o uso de bibliotecas externas: se estiver desenvolvendo um atributo dentro de uma biblioteca que seja compartilhado por outras pessoas, então é necessário garantir que suas mudanças de desenvolvimento não

invalidem outras consultas. Isso significa que você precisa usar um nome de biblioteca diferente no desenvolvimento. O método mais simples é provavelmente usar um atributo diferente para a implementação de biblioteca durante o desenvolvimento. Outra maneira para evitar isso é desenvolver/testar com bibliotecas internas e apenas desenvolver e implementar a biblioteca externa quando estiver pronto para colocar a consulta em produção.

Se as bibliotecas forem aninhadas, o processo ficará mais complicado. Se estiver trabalhando em uma biblioteca C, que é acionada a partir de uma biblioteca A, que é então acionada a partir de uma consulta, então você precisará usar nomes de biblioteca diferentes para as bibliotecas C e A. Caso contrário, ou você não acionará seu código modificado na biblioteca C ou todos que usarem a biblioteca A acionarão o seu código modificado. Você pode optar por tornar as bibliotecas A e C internas, mas não poderá ter os benefícios do tempo reduzido de compilação associado às bibliotecas externas.

Além disso, lembre-se de que suas alterações ocorrem na biblioteca, e não na consulta. Não é incomum pensar porque as alterações no ECL não estão tendo efeito e depois perceber que você estava recompilando/implementando o item incorreto.

Dicas da Biblioteca de Consultas

É possível deixar o código mais limpo tornando a chamada LIBRARY um atributo de função da seguinte maneira:

```
FilterDataset(DATASET(namesRecord) ds,
              STRING search) := LIBRARY('Ppass.FilterDsLib',FilterLibIface1(ds, search));
```

O uso da biblioteca então se torna:

```
FilterDataset(myNames, 'Holliday');
```

O nome da biblioteca (especificado como o primeiro parâmetro para a função LIBRARY) não precisa ser um valor constante, mas não deve ser alterado enquanto a consulta estiver sendo executada. Isso significa que você pode selecionar condicionalmente diferentes versões de uma biblioteca.

Por exemplo, é provável que você queira separar as bibliotecas para processar dados FCRA e não FCRA, já que combinar os dois pode gerar um código ineficiente ou não processável. O código para selecionar entre as duas implementações seria assim:

```
LibToUse := IF(isFCRA,'special.lookupFRCA','special.lookupNoFCRA);
MyResults := LIBRARY(LibToUse, InterfaceCommonToBoth(args));
```

Restrições

O sistema reportará um erro se você tentar usar uma implementação de uma biblioteca de consulta que tenha uma INTERFACE diferente daquela especificada na função LIBRARY.

Há uma restrição especialmente importante no ECL que pode ser incluída em uma biblioteca: serviços de fluxo de trabalho não podem ser incluídos como INDEPENDENT, PERSIST, SUCCESS, e especialmente, STORED. Os atributos STORED não fazem sentido dentro de uma biblioteca de consulta, uma vez que uma biblioteca de consulta deve ser independente de ambas as consultas que a utilizam e de outras bibliotecas de consulta. Em vez de usar atributos STORED (da forma que as consultas Roxie habilitadas para SOAP usam) para especificar parâmetros para as consultas de biblioteca, os parâmetros precisam ser especificados de forma explícita na biblioteca de consulta – seja como um parâmetro individual ou como parte de uma definição de INTERFACE que oferece os argumentos. A consulta que usa a biblioteca de consulta pode utilizar variáveis armazenadas e depois mapear essas variáveis armazenadas para os parâmetros esperados pelas bibliotecas de consulta.

As bibliotecas de consulta atualmente só podem realizar o EXPORT de datasets, das linhas de dados e das expressões de valores únicos. Especificamente, elas não podem realizar ações EXPORT (como OUTPUT), estruturas TRANSFORM ou de outras estruturas MODULE.

Notas sobre a implementação

Há alguns itens que valem a pena ser mencionados sobre a implementação. No Roxie, antes de executar a consulta, todos os gráficos de biblioteca são ampliados para o gráfico de consulta. Quaisquer datasets que são fornecidos como parâmetros para a biblioteca (ou um dataset dentro de um parâmetro de interface) são conectados diretamente ao local em que são usados na biblioteca de consulta, e apenas os resultados usados são avaliados. Isso significa que o uso de uma biblioteca de consulta deve ter pouquíssima sobrecarga em comparação com a inclusão do código ECL diretamente na consulta. OBSERVAÇÃO: Datasets dentro de parâmetros de linha não são transmitidos, de forma que especificar uma ROW que contenha um dataset como um parâmetro para a biblioteca não é tão eficiente quanto usar uma INTERFACE.

A implementação no hthor não é tão eficiente. Parâmetros de dataset tão totalmente avaliados e especificados para a biblioteca como um bloco de unidade completo, e todos os resultados são avaliados. O Thor ainda não é compatível com as bibliotecas de consulta.

O outro item a ser observado é que se a biblioteca A usa a biblioteca C, e se a biblioteca B também usa a biblioteca C com os mesmos parâmetros, as denominações das diferentes bibliotecas não serão combinadas. No entanto, se um atributo exportado de uma instância da biblioteca C for especificado para as bibliotecas A e B, então as denominações para a biblioteca C serão combinadas. A maneira pela qual os atributos atualmente tendem a ser estruturados no repositório, por exemplo, calcular get_Dids() e especificar isso em outros atributos, significa que é improvável que cause qualquer problema na prática.

Estrutura sugerida

Antes de programar várias bibliotecas, vale a pena investir tempo definindo como os atributos são estruturados para uma biblioteca a fim de que todas as bibliotecas no sistema estejam consistentes. Aqui estão algumas diretrizes para usar durante a fase de concepção da biblioteca de consulta:

Convenções de Nomenclatura

Gostaria também de sugerir a criação de uma convenção de nomenclatura consistente antes de desenvolver várias bibliotecas. Em especial, é preciso criar uma convenção para os nomes dos argumentos, para a definição da biblioteca, para a implementação do módulo, para a implementação da biblioteca e para o atributo que encapsula o uso da biblioteca. (Por exemplo, algo como IXArgs, Xinterface, DoX, Xlibrary e X()).

Use uma INTERFACE para definir parâmetros

Este mecanismo (exemplo mostrado abaixo) fornece a documentação para os parâmetros exigidos por um serviço. Isso significa que o código dentro da implementação irá acessá-los como args.xxx ou options.xxx de forma que ficará claro quando os parâmetros estão sendo acessados. Isso também simplifica algumas das sugestões a seguir.

Ocultar a biblioteca

Tornar a chamada LIBRARY um atributo funcional (exemplo também mostrado abaixo) significa que você pode facilmente modificar todos os usos de uma biblioteca se estiver desenvolvendo uma nova versão. Da mesma forma, é possível alternar facilmente para uso de uma biblioteca interna em vez de alterar apenas uma linha de código.

Use estruturas herdadas de MÓDULOS

Use uma estrutura MODULE (sem a opção LIBRARY) que implemente a INTERFACE da biblioteca, e um MODULE individual derivado do primeiro para implementar a LIBRARY utilizando esse módulo de serviço. Ao ocultar a LIBRARY e usar uma implementação de MODULE individual é possível remover facilmente a biblioteca em con-

junto. Além disso, usar uma implementação separada das definições de biblioteca significa que você pode facilmente gerar múltiplas variantes da mesma biblioteca a partir da mesma definição.

```
NamesRec := RECORD
    INTEGER1  NameID;
    STRING20  FName;
    STRING20  LName;
END;
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
                      NamesRec);

 //define an INTERFACE for the passed parameters
IFilterArgs := INTERFACE
  EXPORT DATASET(namesRec) ds;
  EXPORT STRING search;
END;

 //then define an INTERFACE for the query library
FilterLibIface2(IFilterArgs args) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;

 //implement the INTERFACE
FilterDsLib(IFilterArgs args) := MODULE
  EXPORT matches := args.ds(Lname = args.search);
  EXPORT others := args.ds(Lname != args.search);
END;

 //then derive that MODULE to implement the LIBRARY
FilterDsLib2(IFilterArgs args) := MODULE(FilterDsLib(args)),LIBRARY(FilterLibIface2)
END;

 //make the LIBRARY call a function
FilterDs(IFilterArgs args) := LIBRARY(INTERNAL(FilterDsLib2),FilterLibIface2(args));
 //easily modified to eliminate the LIBRARY, if desired
 // FilterDs(IFilterArgs args) := FilterDsLib2(args));
 //define the parameters to pass as the interface
SearchArgs := MODULE(IFilterArgs)
  EXPORT DATASET(namesRec) ds := NamesTable;
  EXPORT STRING search := 'Holliday';
END;

 //use the LIBRARY, passing the parameters
OUTPUT(FilterDs(SearchArgs).matches);
OUTPUT(FilterDs(SearchArgs).others);
```

Smart Stepping (Escalonamento Inteligente)

Visão geral

O Escalonamento inteligente é um conjunto de técnicas de indexação que, juntas, compõem um método de realização de operações de junção/junção de mesclagem com variação *n*, onde *n* é definido como dois ou mais datasets. O Escalonamento inteligente permite que o supercomputador combine registros de forma eficiente a partir múltiplas fontes de dados filtradas, incluindo subconjuntos do mesmo dataset. Ele é especialmente eficiente quando as correspondência são esparsas e não correlatas. O Escalonamento inteligente também é compatível com a junção de registros de datasets M de N.

Antes da criação do Escalonamento inteligente, a localização das interseções de registros de múltiplos datasets era feita ao extrair as possíveis correspondências de um dataset e depois combinando o conjunto candidato a cada um dos outros datasets sucessivamente. As junções usariam diversos mecanismos, incluindo consultas de índice, ou fariam a leitura das possíveis correspondências de um dataset e, depois, juntam-se a eles. Isso significa que a única maneira de unir múltiplos datasets exigia que, no mínimo, um dataset fosse lido inteiramente e depois unido aos outros. Isso poderia ser bastante ineficaz se o programador não tomasse cuidado para selecionar a ordem mais eficiente de leitura dos datasets. Infelizmente, é impossível saber com antecedência qual ordem seria a melhor. Muitas vezes, também é impossível ordenar as junções para que os dois termos menos frequentes sejam unidos. A implementação eficiente das variedades de junção M de N foi particularmente difícil.

Com a tecnologia de Escalonamento inteligente, essas junções de múltiplos datasets se tornam uma operação única e eficiente em vez de uma série de operações múltiplas. O Escalonamento inteligente só pode ser usado no contexto onde a condição da junção é principalmente um teste de semelhança entre as colunas nos datasets de entrada, e os datasets de entrada precisam ter o resultado classificado por essas colunas.

O Escalonamento inteligente também oferece uma maneira eficiente de transmitir informações de um dataset classificadas por qualquer ordem de classificação à direita. Anteriormente, se você tivesse um dataset classificado (muitas vezes um índice) que precisasse ser filtrado por alguns componentes à esquerda, e depois classificar as linhas resultantes pelos componentes à direita, seria necessário fazer a leitura de todo o resultado filtrado para depois classificar o resultado.

O Escalonamento inteligente pode usar quantidades significativas de armazenamento temporário caso não seja usado da maneira correta. Por isso, deve-se ter atenção para que ele seja usado de forma devida.

Ordenação de campos à direita

A função STEPPED oferece a capacidade de classificação pelos principais campos de componentes de chave à direita de modo muito mais eficiente do que a classificação após a filtragem (o único método até então disponível para fazer isso). Os campos de chaves à direita escalonados permitem que as linhas classificadas sejam retornadas sem a leitura de todo o dataset.

Antes da criação do Escalonamento inteligente, um dataset ou índice classificado poderia produzir linhas filtradas de forma eficiente, ou linhas classificadas na mesma ordem daquela original, mas não era capaz de produzir de modo eficiente linhas classificadas por uma ordem de classificação à direita do índice (sejam elas filtradas ou não). O método de filtragem seguido de pós-classificação exigia que todas as linhas do dataset fossem lidas antes que quaisquer linhas classificadas pudessem ser localizadas. O Escalonamento inteligente permite que os dados classificados sejam lidos imediatamente (e, dessa forma, parcialmente).

A maneira mais fácil de ver o efeito é com este exemplo (contido no arquivo SmartStepping1.ECL – este código precisa ser executado no hthor ou Roxie, não no Thor):

```
IMPORT $;
IDX := $.DeclareData.IDX__Person_State_City_Zip_LastName_FirstName_Payload;
Filter := IDX.State = 'LA' AND IDX.City = 'ABBEVILLE';
 //filter by the leading index elements
 //and sort the output by a trailing element
OUTPUT(SORT(IDX(Filter),FirstName),ALL);  //the old way
OUTPUT(STEPPED(IDX(Filter),FirstName),ALL); //Smart Stepping
```

O método anterior de realização deste procedimento implicava na produção do conjunto de resultado filtrado, seguido do uso de SORT para alcançar a ordem de classificação desejada. O novo método é bastante similar, usando STEPPED em vez de SORT, e ambos OUTPUTs produzem o mesmo resultado, mas a eficiência dos métodos pelos quais os resultados são alcançados é bastante diferente.

Depois de executar esse código com sucesso e obter o resultado, observe a página Graphs.

Observe que o primeiro subgráfico do OUTPUT contém três atividades: a leitura do índice, a classificação e o resultado. Porém o segundo subgráfico do OUTPUT contém apenas duas atividades: a leitura do índice e o resultado. Todo o trabalho do Escalonamento inteligente para produzir o resultado é feito pela leitura de índice. Então, se você acessar a página do ECL Watch da tarefa e procurar as medidas de tempo, você deve ver que o tempo do segundo gráfico 1-1 do OUTPUT é significativamente inferior ao do primeiro gráfico 1-2:

Isso demonstra o tipo de vantagem de desempenho que o Escalonamento inteligente pode ter sobre métodos anteriores. É claro que a vantagem de desempenho real é vista quando você solicita apenas os primeiros *n* registros, como neste exemplo (contido no arquivo SmartStepping1a.ECL):

```
IMPORT $;
IDX := $.DeclareData.IDX__Person_State_City_Zip_LastName_FirstName_Payload;
Filter := IDX.State = 'LA' AND IDX.City = 'ABBEVILLE';
OUTPUT(CHOOSEN(SORT(IDX(Filter),FirstName),5));     //the old way
OUTPUT(CHOOSEN(STEPPED(IDX(Filter),FirstName),5));  //Smart Stepping
```

Após executar esse código, verifique as medidas de tempo na página do ECL Watch. Você deve ver novamente uma grande diferença de desempenho entre os dois métodos, mesmo com uma pequena quantidade de dados.

N-ary JOINs

A finalidade principal do Escalonamento inteligente é possibilitar que operações de fusão/junção com variação *n* sejam realizadas do modo mais eficiente possível. Para isso, o conceito de um grupo de datasets (ou índices) foi adicionado à linguagem. Isso permite que JOIN seja estendida para operar em múltiplos datasets, e não apenas dois.

Por exemplo, nestes dados (contidos no arquivo SmartStepping2.ECL)

```
Rec := RECORD,MAXLENGTH(4096)
  STRING1 Letter;
  UNSIGNED1 DS;
  UNSIGNED1 Matches := 1;
  UNSIGNED1 LastMatch := 1;
  SET OF UNSIGNED1 MatchDSs := [1];
END;

ds1 := DATASET([{'A',1},{'B',1},{'C',1},{'D',1},{'E',1}],Rec);
ds2 := DATASET([{'A',2},{'B',2},{'H',2},{'I',2},{'J',2}],Rec);
ds3 := DATASET([{'B',3},{'C',3},{'M',3},{'N',3},{'O',3}],Rec);
ds4 := DATASET([{'A',4},{'B',4},{'R',4},{'S',4},{'T',4}],Rec);
ds5 := DATASET([{'B',5},{'V',5},{'W',5},{'X',5},{'Y',5}],Rec);
```

Para realizar uma junção interna de todos os cinco datasets usando o Escalonamento inteligente, o código é esse (também contido no arquivo SmartStepping2.ECL):

```
SetDS := [ds1,ds2,ds3,ds4,ds5];
```

```
Rec XF(Rec L,DATASET(Rec) Matches) := TRANSFORM
  SELF.Matches := COUNT(Matches);
  SELF.LastMatch := MAX(Matches,DS);
  SELF.MatchDSs := SET(Matches,DS);
  SELF := L;
END;
j1 := JOIN( SetDS,STEPPED(LEFT.Letter=RIGHT.Letter),XF(LEFT,ROWS(LEFT)),SORTED(Letter));

O1 := OUTPUT(j1);
```

Sem usar o Escalonamento inteligente, o código é esse (também contido no arquivo SmartStepping2.ECL):

```
Rec XF1(Rec L,Rec R,integer MatchSet) := TRANSFORM
  SELF.Matches := L.Matches + 1;
  SELF.LastMatch := MatchSet;
  SELF.MatchDSs := L.MatchDSs + [MatchSet];
  SELF := L;
END;
j2 := JOIN( ds1,ds2,LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,2));
j3 := JOIN( j2,ds3, LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,3));
j4 := JOIN( j3,ds4, LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,4));
j5 := JOIN( j4,ds5, LEFT.Letter=RIGHT.Letter,XF1(LEFT,RIGHT,5));
O2 := OUTPUT(SORT(j5,Letter));
```

Ambos os exemplos produzem o mesmo resultado de um registro, mas, sem o Escalonamento inteligente, são necessárias quatro operações separadas de JOINs para alcançar o objetivo e, no código "real", você pode precisar de uma TRANSFORM separada para cada, dependendo do resultado buscado.

Além da junção interna padrão entre todos os datasets, a forma Escalonamento inteligente de JOIN também é compatível com o mesmo tipo de junções LEFT OUTER e LEFT ONLY que a operação padrão de JOIN. No entanto, esta forma também é compatível com as junções *M* de *N* (MOFN), onde os registros correspondentes precisam aparecer em um número mínimo especificado de datasets, e pode opcionalmente especificar um máximo onde eles apareçam, como nestes exemplos (também contidos no arquivo SmartStepping2.ECL):

```
j6 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            LEFT OUTER);
j7 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            LEFT ONLY);
j8 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            MOFN(3));
j9 := JOIN( SetDS,
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter),
            MOFN(3,4));
O3 := OUTPUT(j6);
O4 := OUTPUT(j7);
O5 := OUTPUT(j8);
O6 := OUTPUT(j9);
```

A função RANGE também está disponível para limitar quais datastes no grupo de datasets serão processados, como neste exemplo (também contido no aruqivo SmartStepping2.ECL):

```
j10 := JOIN( RANGE(SetDS,[1,3,5]),
```

```
            STEPPED(LEFT.Letter=RIGHT.Letter),
            XF(LEFT,ROWS(LEFT)),
            SORTED(Letter));
O7 := OUTPUT(j10);

SEQUENTIAL(O1,O2,O3,O4,O5,O6,O7);
```

Este recurso pode ser útil em situações nas quais você possa não ter todas as informações para a seleção de todos os datasets no grupo.

O próximo exemplo demonstra o uso mais provável para essa tecnologia em um ambiente real – localizar o grupo de registros principais onde os registros secundários relacionados existem de forma a combinar com um grupo de critérios de filtro especificados. É exatamente o que esse exemplo (contido no arquivo SmartStepping3.ECL) faz:

```
LinkRec := RECORD
 UNSIGNED1 Link;
END;
DS_Rec := RECORD(LinkRec)
  STRING10 Name;
  STRING10 Address;
END;
Child1_Rec := RECORD(LinkRec)
  UNSIGNED1 Nbr;
END;
Child2_Rec := RECORD(LinkRec)
  STRING10 Car;
END;
Child3_Rec := RECORD(LinkRec)
  UNSIGNED4 Salary;
END;
Child4_Rec := RECORD(LinkRec)
  STRING1
```

Usar essa forma de herança de estrutura de RECORD simplifica bastante a definição da conexão entre os arquivos principal e secundário. Observe que todos esses arquivos possuem formatos diferentes.

```
ds := DATASET([{1,'Fred','123 Main'},{2,'George','456 High'},
               {3,'Charlie','789 Bank'},{4,'Danielle','246 Front'},
               {5,'Emily','613 Boca'},{6,'Oscar','942 Frank'},
               {7,'Felix','777 John'},{8,'Adele','543 Bank'},
               {9,'Johan','123 Front'},{10,'Ludwig','212 Front'}],
              DS_Rec);

Child1 := DATASET([{1,5},{2,8},{3,11},{4,14},{5,17},
                   {6,20},{7,23},{8,26},{9,29},{10,32}],Child1_Rec);

Child2 := DATASET([{1,'Ford'},{2,'Ford'},{3,'Chevy'},
                   {4,'Lexus'},{5,'Lexus'},{6,'Kia'},
                   {7,'Mercury'},{8,'Jeep'},{9,'Lexus'},
                   {9,'Ferrari'},{10,'Ford'}],
                  Child2_Rec);


Child3 := DATASET([{1,10000},{2,20000},{3,155000},{4,800000},
                   {5,250000},{6,75000},{7,200000},{8,15000},
                   {9,80000},{10,25000}],
                  Child3_Rec);

Child4 := DATASET([{1,'House'},{2,'House'},{3,'House'},{4,'Apt'},
                   {5,'Apt'},{6,'Apt'},{7,'Apt'},{8,'House'},
                   {9,'Apt'},{10,'House'}],
                  Child4_Rec);

TblRec := RECORD(LinkRec),MAXLENGTH(4096)
```

```
  UNSIGNED1 DS;
  UNSIGNED1 Matches := 0;
  UNSIGNED1 LastMatch := 0;
  SET OF UNSIGNED1 MatchDSs := [];
END;

Filter1 := Child1.Nbr % 2 = 0;
Filter2 := Child2.Car IN ['Ford','Chevy','Jeep'];
Filter3 := Child3.Salary < 100000;
Filter4 := Child4.Domicile = 'House';

t1 := PROJECT(Child1(Filter1),TRANSFORM(TblRec,SELF.DS:=1,SELF:=LEFT));
t2 := PROJECT(Child2(Filter2),TRANSFORM(TblRec,SELF.DS:=2,SELF:=LEFT));
t3 := PROJECT(Child3(Filter3),TRANSFORM(TblRec,SELF.DS:=3,SELF:=LEFT));
t4 := PROJECT(Child4(Filter4),TRANSFORM(TblRec,SELF.DS:=4,SELF:=LEFT));
```

A operação PROJECT é uma maneira simples de transformar os resultados para todos esses arquivos de formatos diferentes em um layout de padrão único que pode ser usado pela operação de Escalonamento inteligente JOIN.

```
SetDS := [t1,t2,t3,t4];

TblRec XF(TblRec L,DATASET(TblRec) Matches) := TRANSFORM
  SELF.Matches := COUNT(Matches);
  SELF.LastMatch := MAX(Matches,DS);
  SELF.MatchDSs := SET(Matches,DS);
  SELF := L;
END;

j1 := JOIN( SetDS,STEPPED(LEFT.Link=RIGHT.Link),XF(LEFT,ROWS(LEFT)),SORTED(Link));

OUTPUT(j1);

OUTPUT(ds(link IN SET(j1,link)));
```

O primeiro OUTPUT exibe apenas o mesmo tipo de resultado que o exemplo anterior. O segundo OUTPUT produz um grupo de resultados "real" dos registros de dataset de base que correspondem aos critérios de filtro para cada um dos datasets secundários.

Resolvendo Desafios

Produto Cartesiano de Dois Datasets

Um produto cartesiano é o produto de dois conjuntos não-vazios na forma de pares ordenados. Como exemplo, se pegarmos um conjunto de valores (A, B e C) e um segundo conjunto de valores (1, 2 e 3), o produto cartesiano desses dois conjuntos seria o conjunto de pares ordenados A1, A2, A3, B1, B2, B3, C1, C2 e C3.

Para produzir esse tipo de resultado de qualquer um dos dois datasets de entrada, o código ECL seria algo parecido com (contido em Cartesian.ECL):

```
OutFile1 := '~PROGGUIDE::OUT::CP1';

rec := RECORD
  STRING1 Letter;
END;
Inds1 := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                  {'F'},{'G'},{'H'},{'I'},{'J'},
                  {'K'},{'L'},{'M'},{'N'},{'O'},
                  {'P'},{'Q'},{'R'},{'S'},{'T'},
                  {'U'},{'V'},{'W'},{'X'},{'Y'}],
                 rec);

Inds2 := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
                  {'F'},{'G'},{'H'},{'I'},{'J'},
                  {'K'},{'L'},{'M'},{'N'},{'O'},
                  {'P'},{'Q'},{'R'},{'S'},{'T'},
                  {'U'},{'V'},{'W'},{'X'},{'Y'}],
                 rec);

CntInDS2 := COUNT(Inds2);
SetInDS2 := SET(inds2,letter);
outrec := RECORD
  STRING1 LeftLetter;
  STRING1 RightLetter;
END;

outrec CartProd(rec L, INTEGER C) := TRANSFORM
  SELF.LeftLetter := L.Letter;
  SELF.RightLetter := SetInDS2[C];
END;

//Run the small datasets
CP1 := NORMALIZE(Inds1,CntInDS2,CartProd(LEFT,COUNTER));
OUTPUT(CP1,,OutFile1,OVERWRITE);
```

A estrutura principal desse código é o NORMALIZE que vai gerar o produto cartesiano. Os dois datasets de entrada possuem vinte e cinco registros cada, de forma que o número de registros resultantes será seiscentos e vinte e cinco (vinte e cinco ao quadrado).

Cada registro no dataset de entrada LEFT do NORMALIZE executará o TRANSFORM uma vez para cada entrada no SET de valores. Tornar os valores um SET é o segredo para permitir que o NORMALIZE realize esta operação – caso contrário, seria necessário realizar um JOIN onde a condição de combinação é a palavra-chave TRUE para realizar essa tarefa. No entanto, ao testar isso com datasets dimensionáveis (como na próxima instância do código abaixo), a versão NORMALIZE era aproximadamente 25% mais rápida do que o uso do JOIN. Se houver mais de um campo, múltiplos SETs podem então ser definidos e o processo continua o mesmo.

Esse próximo exemplo realiza a mesma operação acima, mas gera primeiramente dois datasets dimensionáveis para trabalhar (também contidos em Cartesian.ECL):

```
InFile1 := '~PROGGUIDE::IN::CP1';
InFile2 := '~PROGGUIDE::IN::CP2';
OutFile2 := '~PROGGUIDE::OUT::CP2';

//generate data files
rec BuildFile(rec L, INTEGER C) := TRANSFORM
  SELF.Letter := Inds2[C].Letter;
END;

GenCP1 := NORMALIZE(InDS1,CntInDS2,BuildFile(LEFT,COUNTER));
GenCP2 := NORMALIZE(GenCP1,CntInDS2,BuildFile(LEFT,COUNTER));
GenCP3 := NORMALIZE(GenCP2,CntInDS2,BuildFile(LEFT,COUNTER));

Out1 := OUTPUT(DISTRIBUTE(GenCP3,RANDOM()),,InFile1,OVERWRITE);
Out2 := OUTPUT(DISTRIBUTE(GenCP2,RANDOM()),,InFile2,OVERWRITE);

// Use the generated datasets in a cartesian join:

ds1 := DATASET(InFile1,rec,thor);
ds2 := DATASET(InFile2,rec,thor);

CntDS2 := COUNT(ds2);
SetDS2 := SET(ds2,letter);

CP2  := NORMALIZE(ds1,CntDS2,CartProd(LEFT,COUNTER));
Out3 := OUTPUT(CP2,,OutFile2,OVERWRITE);
SEQUENTIAL(Out1,Out2,Out3)
```

Neste caso, o uso de NORMALIZE para gerar os datasets é o mesmo tipo de uso previamente descrito no artigo "Como criar dados de exemplo". Depois disso, o processo para chegar ao produto cartesiano é exatamente o mesmo que o do exemplo anterior.

Aqui está um exemplo de como essa mesma operação pode ser feita usando JOIN (também contido no Cartesian.ECL):

```
// outrec joinEm(rec L, rec R) := TRANSFORM
    // SELF.LeftLetter := L.Letter;
    // SELF.RightLetter := R.Letter;
// END;

// ds4 := JOIN(ds1, ds2, TRUE, joinEM(LEFT, RIGHT), ALL);
// OUTPUT(ds4);
```

Registros Contendo Qualquer Conjuntos de Palavras

Parte do problema de limpeza de dados é a possível presença de profanação ou nomes de personagens de desenhos animados nos dados. Isso pode se tornar um problema sempre que você estiver trabalhando com dados originados do registro direto por usuários finais em um site. O código a seguir (contido no arquivo BadWordSearch.ECL) vai detectar a presença de qualquer conjunto de palavras obscenas em um determinado campo:

```
IMPORT std;

SetBadWords := ['JUNK', 'GARBAGE', 'CRUD'];
BadWordDS := DATASET(SetBadWords,{STRING10 word});

SearchDS := DATASET([{1,'FRED','FLINTSTONE'},
                     {2,'GEORGE','KRUEGER'},
                     {3,'CRUDOLA','BAR'},
                     {4,'JUNKER','KNIGHT'},
                     {5,'GARBAGEGUY','MANGIA'},
                     {6,'FREDDY','KRUEGER'},
                     {7,'TIM','TINY'},
                     {8,'JOHN','JONES'},
                     {9,'MIKE','JETSON'}],
                    {UNSIGNED6 ID,STRING10 firstname,STRING10 lastname});

outrec := RECORD
  SearchDS.ID;
  SearchDS.firstname;
  BOOLEAN FoundWord;
END;

{BOOLEAN Found} FindWord(BadWordDS L, STRING10 inword) := TRANSFORM
  SELF.Found := Std.Str.Find(inword,TRIM(L.word),1)>0;
END;

outrec CheckWords(SearchDS L) := TRANSFORM
  SELF.FoundWord := EXISTS(PROJECT(BadWordDS,FindWord(LEFT,L.firstname))(Found=TRUE));
  SELF := L;
END;

result := PROJECT(SearchDS,CheckWords(LEFT));

OUTPUT(result(FoundWord=TRUE));
OUTPUT(result(FoundWord=FALSE));
```

Esse código é um PROJECT simples de cada registro que você deseja procurar. O resultado será um conjunto de registros contendo o campo ID de registro, o campo de busca de primeiro nome e um campo de sinalizador BOOLEAN FoundWord, indicando se alguma palavra obscena foi localizada.

A busca em si é feita por um PROJECT aninhado do campo a ser procurado em relação ao DATASET de palavras obscenas. Usar a função EXISTS para detectar se algum registro foi retornado desse PROJECT, o campo Found retornado é TRUE, define o valor de campo de sinalizador FoundWord.

A função Std.Str.Find apenas detecta a presença de quaisquer palavras obscenas em qualquer lugar na cadeia de busca O OUTPUT do conjunto de registros onde o FoundWord é TRUE e permite o pós-processamento para avaliar se o registro vale a pena ser mantido ou se pode ser descartado (provavelmente exigindo intervenção humana).

O código acima é um exemplo específico dessa técnica, mas seria muito mais útil ter uma MACRO que realizasse essa tarefa, algo como (também contido no arquivo BadWordSearch.ECL):

```
MAC_FindBadWords(BadWordSet,InFile,IDfld,SeekFld,ResAttr,MatchType=1) := MACRO
  #UNIQUENAME(BadWordDS)
  %BadWordDS% := DATASET(BadWordSet,{STRING word{MAXLENGTH(50)}});

  #UNIQUENAME(outrec)
  %outrec% := RECORD
    InFile.IDfld;
    InFile.SeekFld;
    BOOLEAN FoundWord := FALSE;
    UNSIGNED2 FoundPos := 0;
  END;

  #UNIQUENAME(ChkTbl)
  %ChkTbl% := TABLE(InFile,%outrec%);

  #UNIQUENAME(FindWord)
  {BOOLEAN Found,UNSIGNED2 FoundPos}
    %FindWord%(%BadWordDS% L,INTEGER C,STRING inword) := TRANSFORM
  #IF(MatchType=1) //"contains" search
    SELF.Found := Std.Str.Find(inword,TRIM(L.word),1) > 0;
  #END
  #IF(MatchType=2) //"exact match" search
    SELF.Found := inword = L.word;
  #END
  #IF(MatchType=3) //"starts with" search
    SELF.Found := Std.Str.Find(inword,TRIM(L.word),1) = 1;
  #END
    SELF.FoundPos := IF(SELF.FOUND=TRUE,C,0);
  END;
  #UNIQUENAME(CheckWords)
  %outrec% %CheckWords%(%ChkTbl% L) := TRANSFORM
    WordDS := PROJECT(%BadWordDS%,%FindWord%(LEFT,COUNTER,L.SeekFld));
    SELF.FoundWord := EXISTS(WordDS(Found=TRUE));
    SELF.FoundPos := WordDS(Found=TRUE)[1].FoundPos;
    SELF := L;
  END;
  ResAttr := PROJECT(%ChkTbl%,%CheckWords%(LEFT));
ENDMACRO;
```

Essa MACRO faz um pouco mais do que o exemplo anterior. Ela começa passando pelo:

* Conjunto de palavras a ser localizado

* Arquivo para busca

* Campo de identificador único para o registro de busca

* Campo para pesquisa

* Nome de atributo do conjunto de registro resultante

* Tipo de correspondência a ser realizada (com padrão para 1)

Especificar o conjunto de palavras para busca permite que a MACRO opere em relação a qualquer conjunto determinado de strings. Especificar o nome de atributo de resultado permite um pós-processamento simplificado dos dados.

Essa MACRO vai além do exemplo anterior no parâmetro MatchType, que permite que a MACRO use a função Template Language #IF para gerar três tipos diferentes de buscas a partir de uma mesma base de códigos: uma busca com "contém" (o padrão), uma correspondência exata e uma busca de "começa com".

Ela também possui uma estrutura de RECORD de saída expandida que inclui um campo FoundPos para incluir o ponteiro à primeira entrada no conjunto especificado combinado. Isso possibilita que o pós-processamento detecte

correspondências de posição no conjunto para que "pares combinados" de palavras possam ser detectados, como neste exemplo (também contido no arquivo BadWordSearch.ECL):

```
SetCartoonFirstNames := ['GEORGE','FRED', 'FREDDY'];
SetCartoonLastNames := ['JETSON','FLINTSTONE','KRUEGER'];

MAC_FindBadWords(SetCartoonFirstNames,SearchDS,ID,firstname,Res1,2)
MAC_FindBadWords(SetCartoonLastNames,SearchDS,ID,lastname,Res2,2)

Cartoons := JOIN(Res1(FoundWord=TRUE),
                 Res2(FoundWord=TRUE),
                 LEFT.ID=RIGHT.ID AND LEFT.FoundPos=RIGHT.FoundPos);

MAC_FindBadWords(SetBadWords,SearchDS,ID,firstname,Res3,3)
MAC_FindBadWords(SetBadWords,SearchDS,ID,lastname,Res4)
SetBadGuys := SET(Cartoons,ID) +
              SET(Res3(FoundWord=TRUE),ID) +
              SET(Res4(FoundWord=TRUE),ID);

GoodGuys := SearchDS(ID NOT IN SetBadGuys);
BadGuys := SearchDS(ID IN SetBadGuys);
OUTPUT(BadGuys,NAMED('BadGuys'));
OUTPUT(GoodGuys,NAMED('GoodGuys'));
```

Observe que a posição dos nomes de personagens de desenhos animados em seus conjuntos individuais define um único nome de personagem para procurar em múltiplas passagens. Acionar a MACRO duas vezes e procurar pelo primeiro e último nomes separadamente permite o pós-processamento dos resultados com um JOIN interno mais simples onde o mesmo registro foi localizado em cada um e, o mais importante, onde os valores de posição das correspondências são os mesmos. Isso evita que "GEORGE KRUEGER" seja marcado indevidamente como um nome de personagem de desenho animado.

Amostras aleatórias simples

Há um conceito estatístico denominado "Amostra aleatória simples" – diferente de simplesmente usar a função RANDOM() – no qual uma amostra estatisticamente aleatória de registros é gerada a partir de qualquer dataset. O algoritmo implementado no exemplo de código a seguir foi fornecido por um cliente.

Esse código é implementado como uma MACRO para permitir que múltiplas amostras sejam produzidas na mesma tarefa (contida no arquivo SimpleRandomSamples.ECL):

```
SimpleRandomSample(InFile,UID_Field,SampleSize,Result) := MACRO
  //build a table of the UIDs
  #UNIQUENAME(Layout_Plus_RecID)
  %Layout_Plus_RecID% := RECORD
     UNSIGNED8 RecID := 0;
     InFile.UID_Field;
  END;
  #UNIQUENAME(InTbl)
  %InTbl% := TABLE(InFile,%Layout_Plus_RecID%);

  //then assign unique record IDs to the table entries
  #UNIQUENAME(IDRecs)
  %Layout_Plus_RecID% %IDRecs%(%Layout_Plus_RecID% L, INTEGER C) :=
     TRANSFORM
     SELF.RecID := C;
     SELF := L;
  END;
  #UNIQUENAME(UID_Recs)
  %UID_Recs% := PROJECT(%InTbl%,%IDRecs%(LEFT,COUNTER));

   //discover the number of records
   #UNIQUENAME(WholeSet)
   %WholeSet% := COUNT(InFile) : GLOBAL;

   //then generate the unique record IDs to include in the sample
   #UNIQUENAME(BlankSet)
   %BlankSet% := DATASET([{0}],{UNSIGNED8 seq});
   #UNIQUENAME(SelectEm)
   TYPEOF(%BlankSet%) %SelectEm%(%BlankSet% L, INTEGER c) := TRANSFORM
     SELF.seq := ROUNDUP(%WholeSet% * (((RANDOM()%100000)+1)/100000));
   END;
   #UNIQUENAME(selected)
   %selected% := NORMALIZE( %BlankSet%, SampleSize,
                           %SelectEm%(LEFT, COUNTER));

  //then filter the original dataset by the selected UIDs
  #UNIQUENAME(SetSelectedRecs)
%SetSelectedRecs% := SET(%UID_Recs%(RecID IN SET(%selected%,seq)),
                         UID_Field);
  result := infile(UID_Field IN %SetSelectedRecs% );
ENDMACRO;
```

Essa MACRO usa quatro parâmetros:

* O nome do arquivo para amostra * O nome do campo de identificador único nesse arquivo * O tamanho da amostra a ser gerada * O nome do atributo para o resultado (para que possa ser pós-processado)

O algoritmo em si é bastante simples. Primeiro criamos uma TABLE de campos de identificadores numerados de forma única. Em seguida, usamos NORMALIZE para produzir um conjunto de registros dos registros candidatos. O candidato escolhido toda vez que a função TRANSFORM é acionada é determinado ao gerar um valor "aleatório" entre zero e um, usando uma divisão de módulo por cem mil no retorno da função RANDOM() e depois multiplicando o resultado pelo número de registros para amostra, arredondando para o próximo número inteiro maior. Isso determina

a posição do identificador de campo para uso. Depois que o conjunto de posições na TABLE é determinado, ele é usado para definir o SET de campos únicos para uso no resultado final.

Esse algoritmo foi projetado para produzir uma amostra "com substituição" para que seja possível ter um número inferior de registros retornados em comparação ao tamanho de amostra solicitado. Para produzir exatamente o tamanho de amostra necessário (isto é, uma amostra "sem substituição"), é possível solicitar um tamanho de amostra maior (por exemplo, 10% superior) e depois usar a função CHOOSEN para localizar apenas o número real de registros necessário, como neste exemplo (também contido no arquivo SimpleRandomSamples.ECL).

```
SomeFile := DATASET([{'A1'},{'B1'},{'C1'},{'D1'},{'E1'},
                     {'F1'},{'G1'},{'H1'},{'I1'},{'J1'},
                     {'K1'},{'L1'},{'M1'},{'N1'},{'O1'},
                     {'P1'},{'Q1'},{'R1'},{'S1'},{'T1'},
                     {'U1'},{'V1'},{'W1'},{'X1'},{'Y1'},
                     {'A2'},{'B2'},{'C2'},{'D2'},{'E2'},
                     {'F2'},{'G2'},{'H2'},{'I2'},{'J2'},
                     {'K2'},{'L2'},{'M2'},{'N2'},{'O2'},
                     {'P2'},{'Q2'},{'R2'},{'S2'},{'T2'},
                     {'U2'},{'V2'},{'W2'},{'X2'},{'Y2'},
                     {'A3'},{'B3'},{'C3'},{'D3'},{'E3'},
                     {'F3'},{'G3'},{'H3'},{'I3'},{'J3'},
                     {'K3'},{'L3'},{'M3'},{'N3'},{'O3'},
                     {'P3'},{'Q3'},{'R3'},{'S3'},{'T3'},
                     {'U3'},{'V3'},{'W3'},{'X3'},{'Y3'},
                     {'A4'},{'B4'},{'C4'},{'D4'},{'E4'},
                     {'F4'},{'G4'},{'H4'},{'I4'},{'J4'},
                     {'K4'},{'L4'},{'M4'},{'N4'},{'O4'},
                     {'P4'},{'Q4'},{'R4'},{'S4'},{'T4'},
                     {'U4'},{'V4'},{'W4'},{'X4'},{'Y4'}
                     ],{STRING2 Letter});

ds := DISTRIBUTE(SomeFile,HASH(letter[2]));
SimpleRandomSample(ds,Letter,6, res1) //ask for 6
SimpleRandomSample(ds,Letter,6, res2)
SimpleRandomSample(ds,Letter,6, res3)

OUTPUT(CHOOSEN(res1,5)); //actually need 5
OUTPUT(CHOOSEN(res3,5));
```

Hex String para Decimal String

Recebi um e-mail solicitando a sugestão de uma forma de converter uma cadeia com valores hexadecimais para uma cadeia com o decimal equivalente àquele valor. O problema era que esse código precisava ser executado no Roxie e a função biblioteca do plugin StringLib.String2Data não estava disponível para uso em consultas Roxie naquele momento. Dessa forma, era necessária uma solução completamente em ECL.

Essa função de exemplo (contida no arquivo Hex2Decimal.ECL) oferece essa funcionalidade, enquanto também demonstra o uso prático dos números inteiros BIG ENDIAN e de transferência de tipo.

```
HexStr2Decimal(STRING HexIn) := FUNCTION

  //type re-definitions to make code more readable below
  BE1 := BIG_ENDIAN UNSIGNED1;
  BE2 := BIG_ENDIAN UNSIGNED2;
  BE3 := BIG_ENDIAN UNSIGNED3;
  BE4 := BIG_ENDIAN UNSIGNED4;
  BE5 := BIG_ENDIAN UNSIGNED5;
  BE6 := BIG_ENDIAN UNSIGNED6;
  BE7 := BIG_ENDIAN UNSIGNED7;
  BE8 := BIG_ENDIAN UNSIGNED8;

  TrimHex := TRIM(HexIn,ALL);
  HexLen := LENGTH(TrimHex);
  UseHex := IF(HexLen % 2 = 1,'0','') + TrimHex;

  //a sub-function to translate two hex chars to a packed hex format
  STRING1 Str2Data(STRING2 Hex) := FUNCTION
    UNSIGNED1 N1 :=
        CASE( Hex[1],
            '0'=>00x,'1'=>10x,'2'=>20x,'3'=>30x,
            '4'=>40x,'5'=>50x,'6'=>60x,'7'=>70x,
            '8'=>80x,'9'=>90x,'A'=>0A0x,'B'=>0B0x,
            'C'=>0C0x,'D'=>0D0x,'E'=>0E0x,'F'=>0F0x,00x);
     UNSIGNED1 N2 :=
        CASE( Hex[2],
            '0'=>00x,'1'=>01x,'2'=>02x,'3'=>03x,
            '4'=>04x,'5'=>05x,'6'=>06x,'7'=>07x,
            '8'=>08x,'9'=>09x,'A'=>0Ax,'B'=>0Bx,
            'C'=>0Cx,'D'=>0Dx,'E'=>0Ex,'F'=>0Fx,00x);
     RETURN (>STRING1<)(N1 | N2);
  END;

  UseHexLen := LENGTH(TRIM(UseHex));
  InHex2 := Str2Data(UseHex[1..2]);
  InHex4 := InHex2 + Str2Data(UseHex[3..4]);
  InHex6 := InHex4 + Str2Data(UseHex[5..6]);
  InHex8 := InHex6 + Str2Data(UseHex[7..8]);
  InHex10 := InHex8 + Str2Data(UseHex[9..10]);;
  InHex12 := InHex10 + Str2Data(UseHex[11..12]);
  InHex14 := InHex12 + Str2Data(UseHex[13..14]);
  InHex16 := InHex14 + Str2Data(UseHex[15..16]);
  RETURN CASE(UseHexLen,
          2 => (STRING)(>BE1<)InHex2,
          4 => (STRING)(>BE2<)InHex4,
          6 => (STRING)(>BE3<)InHex6,
          8 => (STRING)(>BE4<)InHex8,
          10 => (STRING)(>BE5<)InHex10,
          12 => (STRING)(>BE6<)InHex12,
          14 => (STRING)(>BE7<)InHex14,
          16 => (STRING)(>BE8<)InHex16,
          'ERROR');
```

```
END;
```

Essa HexStr2Decimal FUNCTION usa um parâmetro de STRING de comprimento variável contendo o valor hexadecimal para avaliação. Ela começa redefinindo os oito tamanhos possíveis de números inteiros BIG ENDIAN não assinados. Essa redefinição é apenas para fins estéticos – para melhorar a legibilidade do código subsequente.

Os próximos três atributos detectam se um número par ou ímpar de caracteres hexadecimais foi especificado. Se um número ímpar for especificado, então um caractere "0" será anexado ao valor especificado para garantir que os valores hexadecimais sejam colocados nos nibbles corretos.

A Str2Data FUNCTION usa um parâmetro da STRING de dois caracteres e converte cada caractere para o valor hexadecimal apropriado para cada nibble da STRING resultante de um caractere. O primeiro caractere define o primeiro nibble, enquanto o segundo define o segundo nibble. Esses dois valores passam pela operação OR juntos (usando o operador bitwise |), e o resultado passa por uma transferência de tipo para uma cadeia de um caractere usando a sintaxe abreviada (>STRING1<) para que o padrão de bits permaneça inalterado. O resultado RETURN desta FUNCTION é uma STRING1 porque cada parte sucessiva de dois caracteres do parâmetro de entrada da FUNCTION HexStr2Decimal passará pela FUNCTION Str2Data e será concatenado com todos os resultados anteriores.

O atributo UseHexLen determina o tamanho adequado do número inteiro BIGENDIAN a ser usado na conversão de hexadecimal em decimal, enquanto os atributos InHex2 até o InHex16 definem o valor hexadecimal final acondicionado para avaliação. A função CASE usa esse UseHexLen para determinar qual atributo InHex deve ser utilizado para o número de bytes do valor hexadecimal especificado. Apenas os números pares de caracteres hexadecimais são permitidos (o que significa que o acionamento da função precisaria adicionar um zero à esquerda para quaisquer valores hexadecimais ímpares para conversão) e o número máximo de caracteres permitido é dezesseis (representando um valor hexadecimal acondicionado de oito bytes para conversão).

Em todo os casos, o resultado do atributo InHex passa por uma transferência de tipo para o número inteiro BIG ENDIAN. A conversão de tipo padrão para STRING realiza a conversão do valor real de hexadecimal para decimal.

Os seguintes acionamentos retornam os resultados indicados:

```
OUTPUT(HexStr2Decimal('0101'));               // 257
OUTPUT(HexStr2Decimal('FF'));                 // 255
OUTPUT(HexStr2Decimal('FFFF'));               // 65535
OUTPUT(HexStr2Decimal('FFFFFF'));             // 16777215
OUTPUT(HexStr2Decimal('FFFFFFFF'));           // 4294967295
OUTPUT(HexStr2Decimal('FFFFFFFFFF'));         // 1099511627775
OUTPUT(HexStr2Decimal('FFFFFFFFFFFF'));       // 281474976710655
OUTPUT(HexStr2Decimal('FFFFFFFFFFFFFF'));     // 72057594037927935
OUTPUT(HexStr2Decimal('FFFFFFFFFFFFFFFF'));   // 18446744073709551615
OUTPUT(HexStr2Decimal('FFFFFFFFFFFFFFFFFF')); // ERROR
```

Resolução de Layout de arquivo no tempo de compilação

Ao ler um arquivo em disco no ECL, o layout do arquivo é especificado no código ECL. Isso permite que o código seja compilado para acessar os dados de modo muito eficiente, mas pode causar problemas se o arquivo em disco estiver, na verdade, usando um layout diferente.

Em especial, isso pode ser um desafio para o processo de controle de versão caso você tenha consultas ECL que estejam sendo alteradas para adicionar funcionalidades, mas que precisam ser aplicadas sem modificar os arquivos de dados cujo layout esteja sendo mudado em um cronograma diferente.

Uma solução parcial para esse dilema foi criada no Roxie para indexar arquivos – a capacidade de converter o tempo de execução dos campos no arquivo de índice físico para os campos especificados no índice. No entanto, essa solução tem um grande potencial de sobrecarga e não está disponível para arquivos simples ou no Thor.

Um novo recurso, adicionado à versão 6.4.0 do HPCC Systems, possibilita que a resolução do arquivo seja realizada no tempo de compilação, proporcionando as seguintes vantagens:

- As mudanças de código podem ser isoladas das mudanças de layout de arquivo – você só precisa declarar os campos que deseja de fato usar a partir de um arquivo de dados.
- As incompatibilidades de layout de arquivo podem ser detectadas com antecedência.
- O compilador pode usar informações sobre os tamanhos do arquivo para orientar as decisões de otimização de código.

Há duas construções de linguagem associadas a esse recurso:

- Usando um atributo LOOKUP em declarações DATASET ou INDEX.
- Uso do atributo LOOKUP em uma função RECORDOF

Usando o LOOKUP em um DATASET

A adição do atributo LOOKUP a uma declaração DATASET indica que o layout de arquivo deve ser consultado no tempo de compilação:

```
myrecord := RECORD
  STRING field1;
  STRING field2;
END;

f := DATASET('myfilename', myrecord, FLAT);
  // This will fail at runtime if file layout does not match myrecord
f := DATASET('myfilename', myrecord, FLAT, LOOKUP);
  // This will automatically project from the actual to the requested layout
```

Se supormos que o layout real do arquivo em disco seja:

```
myactualrecord := RECORD
  STRING field1;
  STRING field2;
  STRING field3;
END;
```

Então o efeito do atributo LOOKUP será como se seu código fosse:

```
actualfile := DATASET('myfilename', myactualrecord, FLAT);
f := PROJECT(actualfile, TRANSFORM(myrecord, SELF := LEFT; SELF := []));
```

Os campos que estão presentes em ambas as estruturas de registro são atribuídos entre eles; os campos que estão presentes apenas na versão de disco são descartados e os campos que estão presentes apenas na versão ECL recebem seu valor padrão (um aviso será emitido neste último caso).

Há também uma diretiva de compilador que pode ser usada para especificar a conversão para todos os arquivos:

```
#OPTION('translateDFSlayouts',TRUE);
```

O atributo LOOKUP aceita um parâmetro (TRUE ou FALSE) para permitir o controle mais fácil de onde e quando você deseja que a conversão ocorra. Qualquer expressão booleana que possa ser avaliada no tempo de compilação pode ser fornecida.

Ao usar #OPTION para *translateDFSlayouts*, você pode querer usar LOOKUP(FALSE) para substituir o padrão em alguns datasets específicos.

Uso do atributo LOOKUP em uma função RECORDOF

O uso do atributo LOOKUP em uma função RECORDOF é útil quando os campos estavam presentes no original e depois foram descartados, ou quando você quer gravar em um arquivo que corresponda ao layout de um arquivo existente (mas você não sabe qual é o layout).

O atributo LOOKUP na função RECORDOF usa um nome de arquivo em vez de um dataset. O resultado é expandido no tempo de compilação para o layout de registro armazenado nos metadados do arquivo nomeado. Há várias formas desse constructo:

```
RECORDOF('myfile', LOOKUP);
RECORDOF('myfile', defaultstructure, LOOKUP);
RECORDOF('myfile', defaultstructure, LOOKUP, OPT);
```

Também é possível especificar um DATASET como o primeiro parâmetro em vez de um nome de arquivo (uma conveniência sintática), e o nome de arquivo especificado no dataset será usado para a consulta.

A *defaultstructure* é útil para situações nas quais as informações de layout de arquivo podem não estar disponíveis (por exemplo, ao realizar uma verificação de sintaxe localmente ou ao criar um arquivo). Também é útil quando o arquivo que está sendo verificado possa não existir - é aí que o OPT deve ser usado.

O compilador verifica se a estrutura de registro real obtida da consulta de sistema de arquivo distribuído contém todos os campos especificados e se os tipos são compatíveis.

Por exemplo, para ler um arquivo cuja estrutura é desconhecida, além de conter um campo de ID e criar um arquivo de resultado contendo todos os registros que corresponderam a um valor fornecido, você poderia escrever:

```
myfile := DATASET('myinputfile', RECORDOF('myinputfile', { STRING id },
                                                          LOOKUP), FLAT);
filtered := myfile(id='123');
OUTPUT(filtered,,'myfilteredfile');
```

Detalhes Adicionais

- A sintaxe foi projetada para que não seja necessário realizar a resolução de arquivos para verificação de sintaxe ou criação de arquivos. Isso é importante para que o modo de repositório local possa funcionar.

- A resolução de arquivos externos funciona da mesma maneira – basta usar a sintaxe de nome de arquivo para a resolução de nome de arquivos externos.
- Você também pode usar o atributo LOOKUP nas declarações INDEX, assim como também DATASET.
- Ao usar a forma RECORDOF e fornecer um layout padrão, pode ser necessário usar a forma => da sintaxe de layout de registro para especificar ambos campos de conteúdo e com chave no mesmo registro.
- Arquivos que foram distribuídos aos nós em vez de criados por tarefas ECL podem não ter informações de registro (metadados) disponíveis no sistema de arquivo distribuído.
- Há alguns novos parâmetros ao eclcc que podem ser usados se você quiser usar essa função para compilações locais:

-dfs=ip	Usa IP do Dali especificado para resolução de nome de arquivo.
-scope=prefix	Usa prefixo de escopo especificado na resolução de nome de arquivo.
-user=id	Usa nome do usuário especificado na resolução do nome de arquivo.
password=xxx	Usa senha especificada na resolução de nome de arquivo (deixe em branco para solicitar)

Linguagem Embarcadas e Armazenamento de Dados

Assinatura do Código, Linguagens Embarcadas e Segurança

As versões da plataforma HPCC Systems® anteriores a 6.0.0 sempre concederam algum controle em relação às operações permitidas no código ECL. Isso era feito (entre outros motivos) como forma de garantir que operações como PIPE ou C++ incorporado não possam ser usadas para contornar os controles de acesso sobre os arquivos através da leitura direta pelo sistema operacional.

A versão 6.0.0 (e outras mais recentes) possui dois recursos que oferecem mais flexibilidade sobre o controle dessas operações.

- Agora é possível limitar quais funções SERVICE são acionadas no tempo de compilação usando o atributo FOLD. Normalmente, por motivos de segurança, FOLD só deve ser acionado em módulos assinados.
- É possível configurar os direitos de acesso (que controlam a capacidade de usar PIPE, C++ incorporado, ou o uso restrito de um SERVICE) como dependentes do código que está sendo assinado. Isso significa que podemos fornecer um código assinado na Biblioteca padrão ECL que faça uso desses recursos sem a necessidade de abri-lo a ninguém para acionar qualquer coisa.

Parâmetros de Configuração ECLCC

No Gerenciador de Configurações, o componente ECLCC Server possui uma guia denominada **Options**. Esta guia permite inserir pares de valores de nome nas permissões para executar vários tipos de código ou plugins incorporados.

Name

--allow	Permite a opção especificada
--deny	Nega a opção especificada.
--allowsigned	Permite a opção especificada se o código foi assinado e se há a presença de uma chave.

Observação: Partes da Biblioteca padrão podem não funcionar se o uso do C++ e das definições externas for negado. De um modo geral **allowsigned** é preferencial.

Cluster

Especifica o cluster para o qual esta regra se aplica. Se o cluster for deixado em branco, a restrição se aplicará a todos os clusters no ambiente.

Value

cpp	Permitir/Negar C++ e outras linguagens incorporadas. Para linguagens diferentes de C++ e Cassandra, também deve ser instalado um plugin opcional.
pipe:	Permitir/Negar o uso de aplicações externas através do comando PIPE.
extern:	Permitir/Negar uma função externa (SERVICE)
datafile:	(Válido apenas para --allowedsigned). Isso especifica que o acesso aos dados é permitido apenas se o código tiver sido assinado e se houver uma chave.

Assinatura do Código

A assinatura do código é semelhante a forma com que os e-mails podem ser assinados, de forma a provar sua identidade e que o código não foi adulterado, usando o pacote padrão gpg.

Um arquivo assinado conterá uma assinatura anexa com um hash criptográfico do conteúdo do arquivo e a chave privada do assinante. Qualquer pessoa que tiver a chave pública do assinante poderá verificar a validade da assinatura e que o conteúdo não foi alterado.

Assinamos as definições SERVICE fornecidas pelos plugins padrão ECL e incluímos a chave pública na instalação da plataforma HPCC. O código que tenta usar as definições de serviço assinadas continuará funcionando como antes, porém o código que tentar acionar funções arbitrárias da biblioteca usando definições SERVICE fornecidas pelo usuário resultará em erros de compilação se o código não estiver assinado e se a configuração externa (veja acima) estiver definida para “deny” ou “allowsigned”.

Administradores do sistema podem instalar chaves adicionais na máquina do ECLCC Server. Por isso, se você quiser usar suas próprias definições de serviço, elas podem ser assinadas usando uma chave que tenha sido instalada da seguinte forma:

```
gpg --output <signed-ecl> --default-key <key-id> --clearsign <ecl-file-to-sign>
```

Com este método, uma pessoa de confiança pode assinar o código para indicar que seu uso é aceitável por pessoas não confiáveis sem permitir que essas pessoas executem um código arbitrário.