ネットワーク構造の特徴を生かした利用者インタフェースについて

秋口 忠三　　鈴木 淳之
（静岡大学 電子科学研究科）

1. はじめに

データベース管理システム（DBMS）のアーキテクチャとして，データベースを概念レベル，外部レベルおよび内部レベルの三つのレベルに分けて記述する，いわゆる三層スキーマ構成は高い評価を得ている（1）．

本論文では，概念スキーマとしてCODASYLスキーマ（2）を用いた場合の終端利用者インタフェース（EUI）（3）の実現法について，新しい方式を提案する．EUIは概念スキーマに基づいて外部スキーマを書くためのデータ記述言語と外部スキーマで記述された外部データベースに対してデータ処理要求を表現するための問い合せ言語から構成される．

外部データベースを利用者のデータ要求に適合した構造で与えることができれば，データ処理要求は直接的に表現可能となる．我々は，特定の要求をもった利用者にとって扱かい易いデータベースビューはトリー構造である，という認識に基づいて終端利用者向きデータモデルとしてトリー構造を基本にしたCONTEXTモデルを提案する．そしてCODASYLスキーマ上でCONTEXTを定義するための言語とCONTEXT上で問い合せ要求を記述するための問い合せ言語を与える．CONTEXTということばは，このデータモデルが利用者のデータ処理要求を表現するための適切な場（context）を設定できる，という意味合いをこめて用いている．

本研究の動機は次の三つの問題点の認識に基づいている．

1）現実世界には関係モデル（4）で表現しにくい性質をもったデータが存在する．このようなデータは情報保存セットをもつネットワークモデルで簡潔に表現できることがタい．（これについては2－4節で述べる．）

2）関係モデルでは問い合せを非手続き的に表現するために終端利用者用の問い合せ言語が数タく提案されている（例えばSEQUEL2（5），QBE（6））が，ネットワークモデルでは十分な記述能力をもった問い合せ言語はみられない．

3）ネットワークモデル上で関係モデルインタフェースを実現する方法が提案されている（7）が，この方法ではネットワークモデルで自然に表現されていたデータ間の関係を関係スキーマ上で切り離し問い合せを表現する際に再度結合するという不自然さがあり，ネットワークモデルの特徴を生かせない．

以上よりネットワーク構造の特徴を生かした終端利用者インタフェースの必要性が浮び上がってくる．

第2章では概念モデルとしての関係モデルとネットワークモデルの比較を行ない，第3章では終端利用者用データモデルCONTEXTについて，第4章ではCONTEXT用問い合せ言語QLCについて述べる．

2. 関係モデルとネットワークモデル

データベースで扱かうデータは，きわめて大量であり，個々のデータの異種性よりも共通性に注目した処理が基本であるから，フォーマット化して管理するこ

とが必然的である．レコード構造はこのようなフォーマット化されたデータを表現するのに最適なデータ構造である（8）．現実のDBMSで使用しているデータモデルとしては，関係モデル，ネットワークモデルおよび階層モデルが代表的である（9）が，これらは共にレコード構造を基本構成要素として使用している．

データベースを構成するもう一つの基本要素はCODASYLセットで代表されるレコード間の1対多関係である．本章ではデータベースの基本構成要素であるレコードとセットの定義を与え，関係モデルとネットワークモデルの概念スキーマ表現能力を比較する．

本論文ではCODASYLスキーマ（ネットワークスキーマ）の論理データ記述の部分だけに興味がある．要点を明確にするために二つの仮定を設けた．

1）レコードはデータ集団（data aggregate）を含まない．

2）多重メンバセット（multimember set）を含まない．

2-1．レコードとレコード型

レコードはレコード記述に従って構成された一定系列のデータ項目値の集まりである．データ項目はデータの最小単位である．レコード記述は一意なレコード名といくつかのデータ項目記述を与え一つのレコード型を定める．各々のデータ項目記述はデータ項目名とデータ型を与える．

データベース中のレコードは一つかつただ一つのレコード型に属し，一意な内部識別子をもつ．各時点において一つのレコード型に属するレコードの集合は一意に定まる．

レコード名R，データ項目名I_1，I_2，…，I_mをもつレコード型を

$$R(I_1, I_2, \cdots, I_m)$$

で表わす．レコード型Rに属するレコードを

$$r_i(v_1, v_2, \cdots, v_m)$$

で表わす．ここでr_iはレコードの内部識別子，v_jはデータ項目I_jのデータ項目値である．以後レコードをその内部識別子で代表させる．

レコード型Rに属するレコード集合を

$$Rec(R) = \{ r_i \mid 1 \leq i \leq n \}$$

で表わす．ここで，nはデータベース中でレコード型Rに属するレコードの個数である．

2-2．セットとセット型

セット記述は，一意なセット名と一つのオーナレコード型，一つのメンバレコード型を指定し，一つのセット型を定める．一つのセットはオーナレコード型の一つのレコードとメンバレコード型の任意個のレコードの集まりである．オーナレコード型に属する各々のレコードが一つのセットを作る．メンバレコード型のレコードは一つのセット型の中で高々一つのセットに属し得る．だからセット型はメンバレコード集合からオーナレコード集合への部分関数を表わしている．

セット名S，オーナレコード型R，メンバレコード型Mのセット型を

$$S : R \rightarrow\!\!\!\!\!\rightarrow M$$

で表わす．

2－3．情報保有型セットと非情報保有型セット（10）

各レコード型に対して主キーを定めることができる。主キーはそのレコード型のいくつかのデータ項目から成る。特定の主キーの値をもつレコードは高々一つしか存在しない。

セット型 S：R →→ M について，オーナレコード型Rが主キーをもちしかもメンバレコード型MがRの主キー項目をすべてもっているとき，Sは非情報保有型セット（non-information bearing set）であるという。非情報保有型セットでないセットを情報保有型セット（information bearing set）という。Sが非情報保有型セットならばSのセット集合をデータベースから取り除いてもSが表わしていたレコード間の論理的関係を保存できる。つまりメンバレコードの中にオーナレコードの主キーが含まれているのでその データ項目の値の一致をとればよい。

2－4．関係モデルとネットワークモデルのレコード間関係の表現方法の比較

主キーをもつレコード型のレコード集合は関係モデルの関係と等価である。関係モデルと本論文で用いた用語の対応を次に示す。

レコード型 —— 関係スキーム	レコード —— タプル
レコード名 —— 関係名	データ項目 —— 属性
レコード集合 —— 関係	データ型 —— 定義域

関係スキーマは主キーをもつレコード型の集まりである。関係データベースは関係スキーマで定義されたレコード型のレコード集合の集まりである。

ネットワークスキーマはいくつかのレコード型（必ずしも主キーをもっていなくても良い）といくつかのセット型の集まりである。ネットワークデータベースは，ネットワークスキーマで定義されたレコード型，セット型のそれぞれのレコード集合，セット集合の集まりである。

両データモデルのレコード間関係を比較するために，ある会社のデータベースを考えよう。（図2－1）

DEPT (DNO, DNAME, LOC)
EMP (ENO, DNO, ENAME, JOB, SAL, MGR)
SCH-CAR (ENO, DNO, DEG, SCHOOL)
PART (PNO, PNAME)
SUPPLIER (SNO, SNAME, LOC)
DPS (DNO, PNO, SNO, QTY)

Fig.2-1 (a) Relational schema.

DEPT (DNO, DNAME, LOC)
EMP (ENO, ENAME, JOB, SAL, MGR)
SCH-CAR (DEG, SCHOOL)
PART (PNO, PNAME)
SUPPLIER (SNO, SNAME, LOC)
DPS (QTY)

D-E : DEPT ->> EMP
E-S : EMP ->> SCH-CAR
D-DPS : DEPT ->> DPS
P-DPS : PART ->> DPS
S-DPS : SUPPLIER ->> DPS

Fig.2-1 (b) Network schema. (Simplified CODASYL schema.)

Fig.2-1 Database schema of some manufacture.
(Primary key is underlined.)

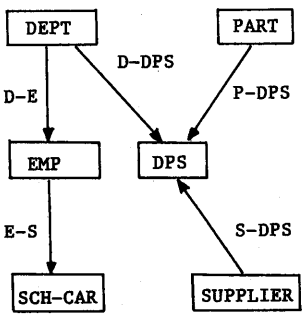


Fig.2-1 (c) Data structured diagram.

関係モデルでは，現実世界で頻繁に現われる階層構造の表現が不自然である．図2-1．a）の例で，ある部門（DEPT）に所属する従業員（EMP）の学歴（SCHCAR）を表わす階層構造を考えてみよう．（この例では従業員番号（ENO）が一つの部門の中だけでしか一意性をもたないとした．）学歴レコードの主キーを作るために上のレベルのレコード型（EMPとDEPT）の主キーを繰り返さなければならず，冗長データを導入し，情報構造が不明確になる．これは情報保存型セットの必要性が主張される根拠であり（10），関係モデルやる正規形批判の原点でもある（11）．

一方ネットワークモデルでは，多対多関係あるいは n項関係（n>2）の表現が不自然である．図2-2．b）の例で，部門が部品供給者（SUPPLIER）から部品（PART）を購入しているという3項関係を表現するために，リンクレコードと呼ばれるレコード型DPSと三つのセット型D-DPS，P-DPS，S-DPSを定義しなければならない．これらは全部が一つにまとまってはじめて意味をもちえる．個々のレコード型，セット型のもつ意味は希薄である．

結局，すべてのセットを非情報保有型とすることも，すべてを情報保有型セットとすることも適当ではないであろう．ここに情報保有型セットをもつネットワークモデルの存在意義とリンクレコードによる表現の不自然さを克服する必要性が痛感されるのである．

3．終端利用者向きデータモデル — CONTEXTモデル

本章では終端利用者向きデータモデルとしてCONTEXTモデルを提案し，CODASYLスキーマ上でCONTEXTを定義するための言語機能を与える．

3－1．終端利用者にとって望ましいデータモデルとは．

終端利用者は一定の処理要求に基づいてデータベースを見る．この場合，一定の規則で結びついたデータ群の中の特定のデータに着目し，そのデータを思考の中心に据えて他のデータはこのデータとの関連において捉える．すなわちデータ処理要求に基づいて一定の要求表現の場（context）を設定する．

このような要求表現の場を与えるためのデータモデルは次の条件を満たすべきであろう．

1）利用者にとって興味のあるデータ間の論理的関係を直接表現できる．

2）特定のデータに注目して他のデータはそれとの関連で据えることができる．

3）データ処理要求を簡潔に表現できる．

我々は，以上の条件に情報保有型セットをもつネットワーク構造の特徴を生かせる，という要求を考慮して，終端利用者用データモデルはレコードをノードとするトリー構造を基本にすべきである，という結論を得た．

トリー構造モデルは関係モデルと比較した場合，次の利点がある．

レコード間の基本的な関係は構造的に表現されるので，問い合せを記述する際にデータ値の一致条件によるレコードの結合（Symbolic join）を行なう必要がほとんどなくなる．

ネットワークモデルと比較した場合，次の利点がある．

二つのレコード間には一つの関係しか存在しないので，問い合せを記述する際にどの経路をとるのかを指定する必要がない．

3－2．アクセス関数

本節では，レコード間の論理的関係を表現するための道具としてアクセス関数（12）の概念を導入する．アクセス関数はネットワークデータベースの中を巡航する（13）ための論理的なアクセス経路を与えてくれる．

［アクセス関数の定義］

アクセス関数は二つのレコード型の間で宣言される．一方を定義レコード型他方を値域レコード型と呼ぶ．両者は同じレコード型であっても良い．アクセス関数は定義域レコード型の各々のレコードを値域レコード型のレコード群（0個以上）に対応づける．アクセス関数は定義域レコード型から値域レコード型へ向うアークとして表わすことができる．

3－2－1．ネットワーク（CODASYL）スキーマ上でのアクセス関数の記述

アクセス関数は基本アクセス関数と合成アクセス関数に分類できる．基本アクセス関数はインプリメンテーションの方法によって，順セット，逆セットおよび記号結合の3種類に分類できる．順セットはセットのオーナレコードからメンバレコードへ向うアクセス関数，逆セットはセットのメンバレコードからオーナレコードへ向うアクセス関数，記号結合は二つのレコード型のデータ項目の一致条件によってレコードを関連づけるアクセス関数である．合成アクセス関数は基本アクセス関数を通常の合成関数の構成規則で組合せることによって宣言できる．

3－3．階層レコード型

あるレコード型に属するレコード集合あるいはその特定の部分集合をあるデータ項目の値の同一性に基づいて類別し，各々のデータ項目値の同値類に対して集計的な処理を施すことが頻繁に行なわれる．たとえば，OS開発部に所属する従業員に対して各職種ごとに平均給与を計算したい，というような要求がこの代表的な例である．

このような要求に対して適当なデータ構造を与えるために，階層レコード型の概念を導入する．レコード型 $R(I_1, I_2, \cdots, I_n)$ をデータ項目 I_i について階層化した2段の階層レコード型は，I_i をデータ項目としてもつ仮想的レコード型 $R(I_i)$ を上位レコードとし，レコード型 R を下位レコードとしてもつ二階層のレコード型であり，次のように宣言される．

$$R(I_i) / X \Longrightarrow R$$

ここで X は仮想レコード型 $R(I_i)$ の参照名でロール名と呼ぶ．

同様にして多段の階層レコード型を定義することもできる．m段（$m<n$）の階層レコード型は次のように宣言される．

$$R(I_{i_1}) / X_1 \Longrightarrow \cdots \Longrightarrow R(I_{i_{m-1}}) / X_{m-1} \Longrightarrow R$$

$R(I_{i_1})$ を階層レコード型の始点レコード型，R を基本レコード型と呼ぶ．

仮想レコード型は問い合せ言語の中で基本レコード型と同様に扱われる．以後仮想レコード型と基本レコード型を含めて単にレコード型と呼ぶ．

階層をもたないレコード型は階層レコード型の特別の場合（$m=1$）であると考えられる．

3－4．ＣＯＮＴＥＸＴデータモデル．

ＣＯＮＴＥＸＴモデルはトリー形アクセスパスグラフ（Ｔ－ＡＰＧ）の各ノードにレコード選択式を付け加えて構成される．

［Ｔ－ＡＰＧの定義］

1．階層レコード型はＴ－ＡＰＧである．この始点レコード型がＴ－ＡＰＧのルートとなる．

2．ＴをＴ－ＡＰＧ，ＢをＴの中の一つの基本レコード型ノード，ＦをＢのレコード型を定義域レコード型とするアクセス関数，Ｆの値域レコードをＲ，Ｒを基本レコード型とする階層レコード型をＨ，Ｈの始点レコード型をＸとする．このときＦによってＸをＢに接続して得られる全体はＴ－ＡＰＧである．Ｔがレコード型Ｒを基本レコード型とするノードをもっていたら両方のノードを区別するために少なくともどちらか一方にロール名をつけなければならない．

3．この操作を有限回適用して得られるものだけがＴ－ＡＰＧである．

Ｔ－ＡＰＧを通して見たデータベースは，概念的にはルートノードのレコード型に属するレコードをルートとするレコードのトリーの集団である．ネットワークデータベースはレコードをノードとするネットワーク構造をもつが，Ｔ－ＡＰＧを通すとネットワークをノードの重複をもたせてトリー状に展開した形で見える．

レコード選択式は，Ｔ－ＡＰＧの各ノードに対してそのレコード型に属するレコードの中でそのＣＯＮＴＥＸＴから見ることのできるレコードの条件を与える．あるノードに付けられた選択式はそのレコード型のデータ項目とその祖先ノードのレコードのデータ項目および定数項より作られる論理式である．まず最初にＣＯＮＴＥＸＴのルートノードに対して対応する選択式が適用される．選択条件を満足しないレコードおよびその子レコードは取り除かれる．条件を満足するレコードの各子レコードに対して同様の選択が繰り返される．

3－5．ネットワーク（ＣＯＤＡＳＹＬ）スキーマ上でのＣＯＮＴＥＸＴの記述

図3－1はアクセス関数とＣＯＮＴＥＸＴの宣言の例を示す．アクセス関数の宣言ではアクセス関数名，定義域レコード型，値域レコード型およびインプリメ

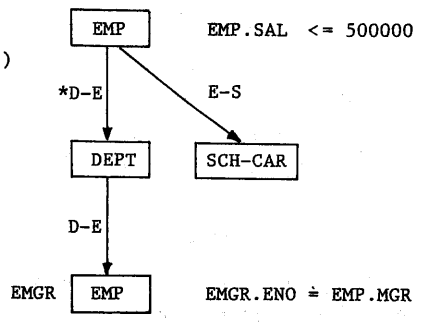
```
(F1)  ACCESS FUNC SUPPLY : SUPPLIER => PART : S-DPS.*P-DPS
(F2)  ACCESS FUNC EQLOCSUP : DEPT => SUPPLIER : ( LOC = LOC )
(F3)  ACCESS FUNC QUICK-SUPPLIED-PART : DEPT => PART
                                      : EQLOCSUP.SUPPLY

(C1)  CONTEXT POOR-EMP-SPEC
          : EMP == E-S ==> SCH-CAR
          , EMP == *D-E ==> DEPT == D-E ==> EMP/EMGR
            WHERE EMP.SAL <= 50000, EMGR.ENO = EMP.MGR

(C2)  CONTEXT DEPT-JOB-EMP
          : DEPT == D-E ==> EMP(JOB)/EJOB ==> EMP

(C3)  CONTEXT DEPT-SUP-PART
          : DEPT == D-DPS.*S-DPS ==> UNIQUE SUPPLIER
                 == @S-DPS.*P-DPS ==> PART
```

Fig.3-1 Some examples of access function and CONTEXT declaration.



EMGR is role name.

Fig.3-2 CONTEXT diagram of POOR-EMP-SPEC.

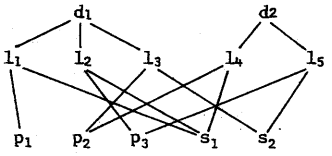
ソテーションの仕様を与える．(F1) の SUPPLY はレコード型 SUPPLIER から PART へ向うアクセス関数で，順セット S-DPS と逆セット *P-DPS の合成アクセス関数である．

(F2) の EQLOCSUP は記号結合アクセス関数の例である．EQLOCSUP は各々の DEPT レコードに対して同じ LOC（所在地）値をもつ SUPPLIER レコード群を対応づける．

(F3) の QUICK-SUPPLIED-PART は既に宣言された アクセス関数の合成アクセス関数である．

CONTEXT の宣言では CONTEXT 名を与え T-APG，レコード選択式を記述する．(C1) の POOR_EMP_SPEC は給与が5万円以下の従業員に関する詳細情報を得るための CONTEXT である．CONTEXT を図式表現すると直感的理解が得やすい．POOR_EMP_SPEC を図式表現した CONTEXT ダイアグラムを図3-2に示す．

(C2) の DEPT_JOB_EMP は3-3節で述べた階層レコード型を用いた CONTEXT の例である．



Rec(DEPT) = {d1, d2}
Rec(PART) = {p1, p2, p3}
Rec(SUPPLIER) = {s1, s2}
Rec(DPS) = {l1, l2, l3, l4, l5}

Fig.3-3 (a) A part of network database.

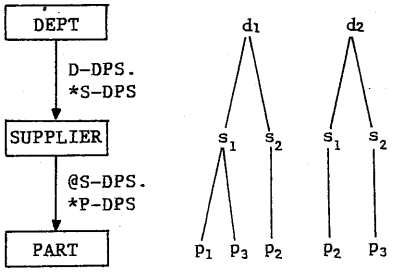


Fig.3-3 (b) Database view through DEPT-SUP-PART.

Fig.3-3 CONTEXT using backtracking.

n 項関係（n > 2）を表わすために導入された リレクレコードが作るネットワーク構造をトリー構造に展開するためには，後もどり処理（backtracking）が必要である．(C3) はその例である．DEPT_SUP_PART は各部門がどの供給者からどの部品を購入しているかを表わす CONTEXT である．図3-3にネットワークデータベースの一部とそれを DEPT_SUP_PART を通して見たときのデータベースビューを与える．アットマーク @ が後もどり処理の指定を意味する．レコード型 SUPPLIER の前の UNIQUE は，同じ DEPT レコードの下で同じ SUPPLIER レコードの重複を取り除く指定である．

4. CONTEXT モデル用問い合せ言語：QLC

本章では CONTEXT の上で問い合せ要求を記述するための言語 QLC (Query Language for Context model) について概要を述べる．CONTEXT の宣言を含めた QLC の構文は付録で与えた．

4-1 QLC の基本機能

QLC はキーワードによって CONTEXT，処理単位および抽出すべき情報の種類を指定する．QLC の基本構成を図4-1に示す．

FROM 句で指定した CONTEXT (C) の中で興味の中心であるレコード型 (R) を FOR EACH 句で指定する．C から見ることのできるレコードトリー群の中の R に属するレコードの各々について，このレコードをルートとす

るサブトリーと（もしあったら）すべての祖先レコードが一つの処理単位となる。FOR EACH句の代わりにFOR SYSTEMを書くとこのレコードトリー群全体が一つの処理単位となる。

複数のCONTEXTを用いる問い合せでは，CONTEXT間で各々の処理単位が満たすべき条件をHAVING句で指定する。一般に各々のCONTEXTに対して処理単位は複数個存在するのでHAVING句の条件式に対して量限定（quantification）を行なう必要がある。FOR SOME句は指定した処理単位の少なくとも一つがCONTEXT間条件を満足していることを要求する。

レコード選択式は各レコードの内容および祖先レコードの内容に基づく制約であった。レコード限定式（qualification）は子レコード群の内容に基づく制約を与える。一般に同一レコード型の子レコードは複数個存在しうるので量限定による制約が必要である。量限定をトリー構造に基づいて表現することにより，制約を受ける対象が自明となり，複雑な問い合せが簡潔に分り易く表現できる。これがCONTEXTモデルおよびQLCの最大の特長といえるであろう。

4－2．QLCによる問い合せ表現の例．

図4-2に図2-1b）のネットワークスキーマに対する問い合せ表現の例を示す．

Q1．部門番号が25か47のいずれかの部門に所属する給与が10万円以上の従業員の従業員番号，職種，および所属部門名を求む．

Q2．各部門における従業員の最高，最低の給与について会社全体での平均を求む。

Q3．事務員が二人以上いる部門について，部門番号とその部門の従業員全体の平均給与を求む．

Q4．上司より多くの給与を得ている従業員の従業員番号と所属部門番号およびその上司の従業員番号を求む．

Q5．部門番号が50の部門が使用している部品をすべて供給している供給者のすべての属性を求む．

Q6．取得したすべての種類の学位を同一の学校から受けている従業員の名前を求む。

Q7．すべての部品供給者から2種類類以上の部品を購入している部門の部門番号と部門名を求む．

Q8．各部門について職種別に従業員の平均給与を求む．

Q2．では局所処理単位の指定を行なっている．従業員の給与の最大値，最小値は各々の部門について求められる．即ちこの値は部門の属性と考えることができる．AVGはこの部門属性に対して作用する．

Q5はこつのCONTEXTを用いた問い合せの例である．FOR SOME UNIQUE DEPTは条件を満たす部門がただ一つだけ存在していることを要求する．GET句でレコード名を指定するとそのすべてのデータ項目値が取り出される．．

Q6の問い合せ表現は次のように読むことができる．「各々の従業員に対して「ある出身校があって，そこで取得した学位の集合はその従業員が取得した全学位の集合に等しい」ならばその従業員の名前を求む．」内がレコード限定式の解釈である．要求をきわめて直接的に表現できることが理解できるであろう．

```
GET     target-list
FROM    context-list
FOR EACH  rec-id  ( qualification )
FOR SOME  rec-id  ( qualification )
HAVING  inter-context-condition
```

Fig.4-1 Main frame of a typical query expression by QLC.

Q7とQ8で使用しているCONTEXTは3-5節のC3C2でそれぞれ宣言したものである．これら二つの問い合せがきわめて簡潔に表現できるのはCONTEXT記述能力の豊かさによるところが大きい．

```
(Q1)  GET   ENO, JOB, DNAME
      FROM  DEPT == D-E ==> EMP
        WHERE  DNO=25 OR DNO=47, SAL>=100000
      FOR EACH  EMP

(Q2)  GET   AVG(MAX(EMP.SAL FOR EACH DEPT))
           ,AVG(MIN(EMP.SAL FOR EACH DEPT))
      FROM  DEPT == D-E ==> EMP
      FOR  SYSTEM

(Q3)  GET   DNO, AVG(SAL)
      FROM  DEPT == D-E ==> EMP
      FOR EACH  DEPT(COUNT(EMP : JOB='CLERK')>=2)

(Q4)  GET   DNO, EMP.ENO, EMGR.ENO
      FROM  EMP == *D-E ==> DEPT == D-E ==> EMP/EMGR
        WHERE  EMGR.ENO=EMP.MGR
      FOR EACH  EMP(SOME EMGR(EMGR.SAL<EMP.SAL))

(Q5)  GET   SUPPLIER
      FROM  SUPPLIER == SUPPLY ==> PART/PS
           ;DEPT == D-DPS.*P-DPS ==> PART/PD
      FOR EACH  SUPPLIER
      FOR SOME UNIQUE DEPT(DNO=50)
      HAVING  SET(PS)>=SET(PD)

(Q6)  GET   ENAME
      FROM  EMP == E-S ==> SCH-CAR(SCHOOL)/SCX
                       ==> SCH-CAR/DX
           ,EMP == E-S ==> SCH-CAR/DY
      FOR EACH  EMP(SOME SCX(SET(DX.DEG)=SET(DY.DEG)))

(Q7)  GET   DNO, DNAME
      FROM  DEPT-SUP-PART
      FOR EACH  DEPT(ALL SUPPLIER(COUNT(PART)>=2))

(Q8)  GET   DNO, JOB, AVG(SAL)
      FROM  DEPT-JOB-EMP
      FOR EACH  EJOB
```

Fig.4-2 Some examples of query expression by QLC.

5. まとめ

本論文ではネットワークデータベースへの端末利用者インタフェースについて述べてきた。

端末利用者向きデータモデルとしてCONTEXTモデルを提案した。CONTEXTモデルはトリー構造を基本とし，利用者のデータ要求に適したデータベースビューを与える．

次にCONTEXTをネットワークスキーマ（単純化したCODASYLスキーマ）の上で定義するための言語機能を与えた。CONTEXTモデルはネットワーク構造の特徴を十分生かしていることを示した．

最後にCONTEXT上で問い合せ要求を表現するための問い合せ言語QLCの概要を与えた。本論文ではQLCの問い合せ機能だけについて述べた。更新機能については現在検討を加えているところである．

CONTEXTモデルは利用者のレベルに応じて適切な利用者インタフェースを実現できる．レベルの高い利用者は自らCONTEXTを作成し自分の要求に適したデータベースビューを形成できる．レベルの低い利用者は自分のデータ要求をデータベース管理者に伝えCONTEXT作成を肩代りしてもらうことになろう．適切なCONTEXTが与えられればその上で要求を表現することは容易である．

本論文で提案したCONTEXTモデルに基づく利用者インタフェースはネットワークモデルに限らずレコード構造を基本にした他のどのデータモデルの上でも実現可能であることは明らかであろう．

[REFERENCES]

1) The ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems, Information Systems, Vol.3,No.3, pp.173-191, (1978)

2) Report of the CODASYL Data Description Language Committee, Information Systems, Vol.3, No.4, pp.247-320, (1978)

3) A Progress Report on the Activities of the CODASYL End User Facility Task Group, June 1975, ACM FDT, Vol.8, No.1, (1976)

4) E.F. Codd :A Relational Model of Data for Large Shared Data Banks, CACM, Vol.13, No.6, pp.377-387, (1970)

5) D.D. Chamberlin, et al. :SEQUEL2 : A Unified Approach to Data Description, Manipulation, and Control, IBM J.Res.Develop., pp.560-575, (1976)

6) M.M. Zloof :Query-by-Example : a Data Base Language, IBM Syst.J., Vol.16, No.4, pp.324-343, (1977)

7) C. Zaniolo :Design of Relational Views over Network Schemas, Proc. 1979 ACM SIGMOD, PP.179-190, (1979)

8) W.Kent :Limitations of Record-Based Information Models, ACM TODS, Vol.4,No.1, pp.107-131, (1979)

9) C.J.Date :An Introduction to Database Systems, 2nd ed., Addison-Wesley, (1977)

10) A.Metaxides :"Information bearing" and "non-information bearing" sets, Data Base Description, North-Holland, pp.363-368, (1975)

11) 植村俊亮： データベースの関係モデル第3正規形批判，情報処理，Vol.18, No.1, pp.58-62, (1977)

12) J.R. Abrial :Data Semantics, Data Base Management, North-Holland, PP.1-60, (1974)

13) C.W. Bachman :The Programmer as Navigator, CACM, Vol.16, No.11, pp.653-658, (1973)

APPENDIX - I SYNTAX OF QLC.

```
statement ::= env-spec  request { request } END
env-spec ::= SCHEMA  schema-name
             ACCESS CONTROL KEY  key-value
             rec-entry { rec-entry } { set-entry }
            { af-declaration } {context-declaration }
rec-entry ::= RECORD  schema-rec-name [ => renamed-rec-name ]
              ( item-entry )
item-entry ::= item-decl { item-decl} | ALL ITEM | NO ITEM
item-decl ::= schema-item-name [ => renamed-item-name ]
set-entry ::= SET  schema-set-name [ => renamed-set-name ]
request ::= query | update

af-declaration ::= ACCESS FUNCTION  af-name
                   : rec-name => rec-name: af-def
af-def ::= ( item-join { , item-join } ) | composite-af
item-join ::= item-name = item-name
composite-af ::= primitive-af  {. primitive-af}
primitive-af ::= [ inverse ] set-name | af-name
inverse ::= *

context-declaration ::= CONTEXT  context-name : context-spec
context-spec ::= hier-rec  {arc-rec }
     {, rec-id  arc-rec { arc-rec}} [ WHERE  rec-sel-exp ]
arc-rec ::= arc  hier-rec
hier-rec ::= [ UNIQUE ] rec-name  hier-type
hier-type ::= [ / role-name ]
             |( cluster-list ) / role-name ==> hier-rec
cluster-list ::= item-name  {, item-name }
arc ::= == access-path { . access-path } ==>
access-path ::= [ backtrack ] primitive-af
back-track ::= @

query ::= GET  [ ( unsigned-int ) ]  target-list
        FROM  context  { ; context }
       {FOR  proc-unit  [ ( qualification ) ]}
      [ HAVING  inter-context-cond ]
target-list ::= target  {, target }
context ::= context-name  [ WHERE  rec-sel-exp ]
           |context-spec
proc-unit ::= SYSTEM | EACH  rec-id | ALL  rec-id
            |SOME  [ UNIQUE ]  rec-id
qualification ::= logical-exp
inter-context-cond ::= logical-exp
rec-sel-exp ::= logical-exp  { , logical-exp }
target ::= arith-exp | logical-exp | rec-id | RESULT

logical-exp ::= logical-term  {OR  logical-term}
logical-term ::= logical-factor  {AND  logical-factor}
logical-factor ::= [ NOT ]  logical-primary
logical-primary ::= ( logical-exp ) | predicate
predicate ::= item-predicate | agg-predicate
item-predicate ::= item-id  rel-op  arith-exp
agg-predicate ::= agg-func-primary  rel-op  arith-exp
                 |quantifier  {quantifier} ( logical-exp )
                 |set-exp  set-rel-op  set-exp
quantifier ::= quant-symbol  bound-var
quant-symbol ::= SOME | ALL
bound-var ::= rec-id | item-id  /  item-var
agg-func-primary ::= agg-func  (  rec-attr  [ : logical-exp ]
                                 [ FOR EACH  rec-id ]  )
                   |COUNT ( [ UNIQUE ]  rec-id  [ : logical-exp ]
                              [ FOR EACH  rec-id ] )

arith-exp ::= arith-term  {add-op  arith-term}
arith-term ::= arith-factor  {mult-op  arith-factor}
arith-factor ::= [ add-op ]  arith-primary
arith-primary ::= constant | rec-attr | (  arith-exp  )

set-exp ::= set-term  {add-op  set-term}
set-term ::= set-primary  { *  set-primary}
set-primary ::= ( set-exp ) | set-const
               |SET ( elem-type [ : logical-exp ] )
elem-type ::= rec-id | item-id { , item-id}
set-const ::= NULL | (* lit-tuple { , lit-tuple } *)
lit-tuple ::= constant  |<constant { , constant}>

rec-attr ::= item-id | agg-func-primary
item-id ::= [ rec-id . ]  item-name | item-var
rec-id ::= rec-name | role-name
agg-func ::= MAX | MIN | SUM | AVG | COUNT
rel-op ::= = | /= | < | > | <= | >=
set-rel-op ::= rel-op
add-op ::= + | -
mult-op ::= * | /

constant ::= NULL | USER | DATE | TIME
           |quoted-string | number
quoted-string ::= ' char { char} '
number ::= unsigned-int  [ . unsigned-int ]
          [ E [ add-op ]  unsigned-int ]
unsigned-int ::= digit { digit }
```