関係演算のストリーム指向型並列処理における資源割り当て方式†

清木 康†† 劉 澎††† 益田隆司††

我々は，大量データを対象とする関係演算（関係データベース演算）を限られた計算機資源の中で並列に処理する方式としてストリーム指向型関係演算処理方式を提案している．本論文では，ストリーム指向型関係演算処理方式の有効性を最大限に引き出すための計算機資源割り当て方法を提案する．まず，限られたバッファ資源の中で，ストリーム指向型関係演算処理方式による問い合わせ処理の計算量を最小にする最適化方法を示す．次に，並列処理環境において，本方式の並列処理の効果を引き出すための資源割り当ての条件について述べる．最後に，実際の問い合わせ処理を行った実験の結果を示し，本論文で提案する計算機資源割り当て方法の有効性について考察する．

1. まえがき

近年，関係データベース管理システムの普及とともに，その応用分野は急速に広がってきている．また，関係データベースの演繹データベース等の知識ベースへの拡張に関する研究もさかんに行われている[3]．

このような背景の中で，データベースの応用分野は多様化し，また，扱うデータ量が増大してきており，それらに柔軟に適応できる並列処理システムの実現が重要な課題となっている．

関係データベース管理システムの基本演算である関係演算（関係データベース演算）の処理の高速化を目指して，現在までに多くの並列処理アルゴリズムおよびデータベースマシンのアーキテクチャが提案されてきた（文献 6),7) 等参照）．しかし，データベースの多様な応用分野および知識ベースに柔軟に適応するためには，関係演算の処理機能だけでなく，新しい任意の基本演算の追加を行い，それらを並列処理環境の中に容易に組み込む機能が必要となる．

我々は，個々の関係演算に内在する並列性の抽出より，むしろ，問い合わせを構成する関係演算間に存在する並列性の抽出に主眼をおき，限られた計算機資源の中で問い合わせの並列処理を実現するストリーム指向型関係演算処理方式およびその実現法を提案している[8)~10)]．また，データベースや知識ベースのような大量データを扱う様々な応用分野に柔軟に適応できる並列処理システム SMASH を提案している[10),11)]．この並列処理システムは，大量データを対象とする任意の基本演算を並列処理環境の中に容易に組み込むための機能を有し，それらの基本演算を関数型計算[1),2),15)]の枠組みの中で並列に処理する機構を実現するものであり，関数型計算を大量データ処理へ適用するという新しいアプローチに基づくシステムである．このシステムは，複数台の汎用プロセッサを高速ネットワークにより結合した環境において，要求駆動型制御による関数型計算の枠組みの中で基本演算間の並列処理を実現する[9)~11)]．

ストリーム指向型関係演算処理方式は，次の事項を目標としている．

（1） データベースの問い合わせ処理は，2次記憶から主記憶へのデータ転送，主記憶上での関係演算群の処理，ユーザへの結果の転送からなる．本方式では，演算対象データをデータのストリームとしてとらえ，そのストリームが滞ることなく，2次記憶からユーザまで伝えられる環境を実現することを目標とする．したがって，その流れを停滞させないために，並列処理を導入する．

（2） 限られたハードウェア資源の中で，大量データを扱うデータベースの基本演算を実行するための環境を実現する．プロセッサ資源，バッファ資源が限られている環境において，それらの資源を用いて問い合わせに柔軟に適応する．

並列処理システム上で本方式を実現する場合における重要な課題は，計算機資源（プロセッサ資源およびバッファ資源）の割り当てである．関係データベース

† A Resource Allocation Strategy in the Stream-oriented Parallel Processing Scheme for Relational Database Operations by YASUSHI KIYOKI (Institute of Information Sciences and Electronics, University of Tsukuba), PENG LIU (Doctoral Program in Engineering, University of Tsukuba) and Takashi Masuda (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学電子・情報工学系
††† 筑波大学工学研究科

の問い合わせ処理においては，問い合わせのコンパイル時に関係演算の組合せを決定できるので，コンパイル時に静的に資源を割り当てておけば，並列処理の効果を最大限に引き出すことができると考えられる．そこで，我々は，実行前にあらかじめ，静的にプロセッサ資源およびバッファ資源の割り当てを決める方法をとっている．

本論文では，ストリーム指向型関係演算処理方式における計算機資源の割り当て方法について述べる．限られたバッファ資源内で，問い合わせ処理の計算量を最小化するための方法，およびストリーム型の並列性を抽出するための資源割り当てについて述べる．

そして，ここで示す資源割り当て方法により，本方式の効果が引き出されることを実験結果により明らかにする．

第2章では，ストリーム指向型関係演算処理方式，および，その実現法の概要について述べる．詳細については，すでに文献[9),10)]において述べている．

第3章では，ストリーム指向型関係演算処理方式における計算機資源の割り当て方法を提案する．

第4章では，第3章で示す方法の有効性を確かめるために行った実験の結果を示す．

2. ストリーム指向型関係演算処理方式

2.1 関係演算の処理方式

関数型計算は，並列性の柔軟な抽出が容易であることが知られており，我々は，その計算機構をデータベース処理に適用することを試みている[10)]．関数型計算の駆動方式としては，逐次型評価，データ駆動型評価，要求駆動型評価がある．

要求駆動型評価には，次のような特徴がある．

(1) 関数型計算における不必要な引数の評価を排除できる．

(2) データが必要になったときに必要な個数だけそのデータの生成者に生成させるように制御することができる．

(3) 関数の引数の評価を行う際に，要求（デマンド）を伝播するためのオーバヘッドが生じる．

本方式では，関係演算をリレーションを引数とする関数計算としてとらえる．関係演算のような大量データを扱う処理では，(3)に関しては，1回のデマンドによって大きな単位のデータ生成を生成者に要求するようできるので，1回のデマンドの伝播によるオーバヘッドの影響は処理全体からみれば小さい．むしろ，(2)の特徴により有限資源下で並列処理を実現できることによる効果が顕著になるものと考えられる．

ストリーム指向型関係演算処理方式では，要求駆動型評価の枠組みの中で次のような2種類の並列性を引き出す．

(1) 関数引数の並列評価

問い合わせ処理においては，互いに独立な関係演算が並列に実行されることに対応する．**図1**の問い合わせにおいては，2つの結合演算（Join-1, Join-2）は並列に実行可能である．この並列性は，複数の引数データの消費を行う関数が，それらの引数データを生成する各関数に対して，デマンドを同時に発行することによって抽出される．

(2) ストリーム型並列性

問い合わせ処理においては，リレーションを生成する関数とそれを消費する関数が並列に実行されることに対応する．図1においては，結合演算（Join-2）と結合演算（Join-3）が並列に実行可能である．

ストリーム指向型関係演算処理方式の概要を以下に示す．

ここでは，中間結果のタプル群を生成する関数（関係演算）を生産者ノード，それらを消費する関数（関係演算）を消費者ノードとする．

1項関係演算（演算対象リレーション数が1個の関係演算：選択演算等）は，1入力バッファおよび1出力バッファを持つ．2項関係演算（演算対象リレーション数が2個の関係演算：結合演算等）は2入力バッファおよび1出力バッファを持つ．各バッファは生産者ノードと消費者ノードの2つの関数によって共有される．

ストリーム型並列処理を行う場合には，生産者ノードと消費者ノードを異なるプロセッサに配置し，それ

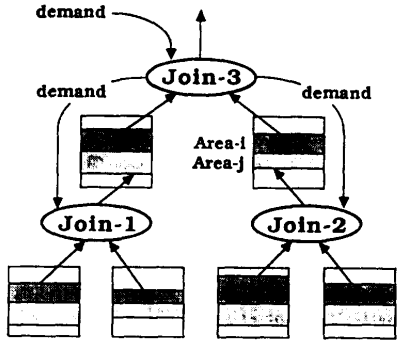


図1 ストリーム指向型関係演算処理
Fig. 1 Stream-oriented processing for relational operations.

らのノード間のバッファには，ダブルバッファリング機構を実現する必要がある．以下では，まず，ストリーム指向型関係演算処理を逐次的に実現する方式を示し，次に，並列処理方式を示す．

消費者ノードは，入力バッファ内の1ページの処理を終えると，生産者ノードへデマンドを発行する．生産者ノードはデマンドを受け取ると，出力バッファ内に演算結果の1ページを生成の完了するまで関係演算を実行し，その時点で実行を停止する．各関係演算ノードは，1デマンドに対してリレーション全体を生成せず，あらかじめ定められた1ページ分のタプル群を生成して停止する．したがって，各バッファは中間リレーション全体を格納する空間を必要としない．この方式により，大量データを扱う関係演算処理を限られたバッファ資源の中で行うことが可能となる．したがって，本方式は，問い合わせを構成する複数の関係演算を1プロセッサにより要求駆動型評価の枠組みの中で擬似並列的に逐次処理する場合にも有効である[9]．

図1のような問い合わせに対してストリーム型並列処理を行う場合には，結合演算ノード (Join-3) と結合演算ノード (Join-2) の間のバッファに，ダブルバッファリング機構が実現される．この場合，各バッファには2つの領域 (Area-i, Area-j) が用意される．各領域はそれぞれ1ページ分のタプル群を格納する容量をもつ．消費者ノードは，一方の領域（ここでは Area-i とする）に格納されているタプル群に対して関係演算を実行する前に，他方の領域 (Area-j) に次のページの格納を要求するために，生産者ノードへデマンドを先行的に発行する．これは，先行評価[1]を要求駆動型評価の枠組みの中で行うための手段である．消費者ノードは入力バッファの Area-i に格納されているタプル群に対する処理を完了すると，Area-i へ次ページの格納を要求するために生産者ノードへ再びデマンドを発行する．

そして，先出ししたデマンドによってすでに Area-j が埋められているならば，Area-j 内のタプル群に対して関係演算の実行を開始する．このとき，生産者ノードは，Area-i へ次ページを格納するために，関係演算の実行を開始する．もし，Area-j が埋められていないならば，消費者ノードはそれが埋められるまで待つ．

結果として，生産者ノードと消費者ノード間でのストリーム型並列処理が限られたバッファ資源の中で実現される．

2.2 再 計 算

2項関係演算では，アウタ・リレーションの各ページとインナ・リレーションの全ページの付き合わせを行うことにより演算が完了する．本方式では，これを限られたバッファ資源の中で行うために，アウタ・リレーションの全タプルが入力バッファに入り切らない場合には，インナ・リレーションを生成する関係演算ノードの再計算が必要となる．図1において，結合演算 (Join-3) のアウタ・リレーション用バッファにアウタ・リレーションの全タプルが入り切らない場合，すなわち，アウタ・リレーションのページ数が N (ページ，$N>1$)，アウタ・リレーション用バッファの大きさを1（ページ）とした場合には，インナ・リレーションを生成するノード (Join-2) を N 回計算する必要がある．以後，この回数を演算回数とよぶことにする．この再計算処理と結合演算 (Join-3) の処理がストリーム型並列処理により同時に実行されれば，そのオーバヘッドは並列処理の中にかくれることになる．この点については，3.2 節で詳しく述べる．

本方式では，ベース・リレーション以外に，各ノードのバッファとして1タプル分の記憶領域があれば，問い合わせ処理が可能となる．しかし，その場合には，再計算によるオーバヘッドが大きくなるので，適当なバッファ領域を各ノードのアウタ・リレーション用バッファに与えることが必要となる．

3. 資源割り当て

ストリーム指向型関係演算処理方式の実現のために，次のような2種類の並列処理環境の検討を行った．

（1） 要求駆動型評価機構をもつ専用並列処理マシン上で，関数型言語により記述した関係演算プログラムを直接に実行する．この場合，関数型言語には，引数の先行・遅延評価[1]を指示する機能，すなわち，関数間でのデマンドおよびストリームの伝達の制御を明示的に記述する機能が必要である．この実現法を前提として記述した関係演算プログラムについては文献8)に述べている．

（2） 関数として記述された各基本演算単位に，ストリーム処理および要求駆動型制御を実現する基本プリミティブ[10]を含む逐次的なコードにコンパイルし，高速ネットワーク上に複数台の汎用プロセッサを接続した環境で，各基本演算を関数としてプロセッサへ割

り当て，関数間の並列性を引き出す．デマンドおよびストリームの転送はプロセッサ間のメッセージ・パッシングにより実現される．

我々は，現在，(2)の環境の並列処理システムの構築を行っている．この環境においてストリーム指向型関係演算処理方式を実現するとき，再計算回数を軽減するために，最適なバッファ資源量の割り当てを決定する計算が次の場合に必要になる．

(1) ストリーム指向型関係演算処理方式は，1プロセッサ内で擬似並列的な問い合わせ処理を行う場合にも，限られたバッファ資源の中で各関係演算を起動でき，処理を完了することができるので有効である．このような場合，各関係演算ノードへ割り当てる最適なバッファ資源量を決定する計算が必要となる．逐次処理の環境において，限られたバッファ資源を複数の結合演算の演算対象リレーションに割り当て，I/O 回数を軽減するアルゴリズムが提案されている[5]が，そのアルゴリズムは，演算対象リレーション数が多くなるにしたがい最適解を得る可能性が減少する．本方式は，関係演算の演算回数の最小化を目的とするもので，任意の演算対象リレーション数に対し演算回数を最小にするバッファ資源配置を必ず決定できる．

(2) 本方式では，各プロセッサに，1つの問い合わせの中の複数の関係演算を割り当てることができる[9),10)]．これにより，限られたプロセッサ台数で問い合わせ処理を完了することができる．そこで，各プロセッサに割り当てられた部分問い合わせ (subquery) を構成する各関係演算ノードへ最適なバッファ資源量を割り当てるための計算が必要となる．

3.1 資源割り当て計算

本節では，限られたバッファ資源内で問い合わせ処理の計算回数を最小化するために，各関係演算ノードへの最適なバッファ資源の割り当てを行うための計算方法について述べる．

ここでは，問い合わせの実行に要するタプルの比較回数を計算回数とし，問い合わせの実行を完了するのに要する計算回数の最小値を求める計算により最適なバッファ資源割り当てを決定する．

ここでは，**図2**に示すような結合演算群からなるチェイン型問い合わせ（関係演算群が1列に並んだ形の問い合わせ）の場合を例として，資源割り当て計算方法を示す．simple query[16] の場合は常にチェイン型問い合わせとして実行可能である．一般的に，多くの問い合わせはチェイン型に変換できる．以下で提示する割り当て方式は，他の関係演算を含むチェイン型以外の問い合わせに対しても同様に適用できる．

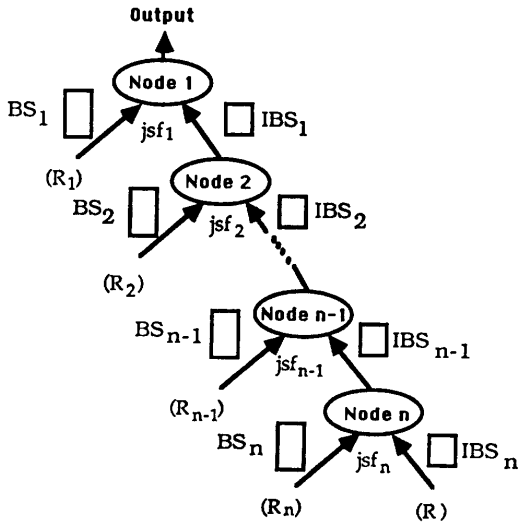


図 2 チェイン型問い合わせ
Fig. 2 A chain query.

まず，ここで用いるパラメータを次のように定義する．

(以後，Join-(i) ノードは，図の中では Node-(i) に対応するものとする．)

jsf_i：Join-(i) ノードの結合選択率 (join selectivity factor).

((Join-(i) ノードが生成する中間リレーションのタプル数)＝jsf_i＊(Join-(i) のインナ・リレーションのタプル数)＊(Join-(i) のアウタ・リレーションのタプル数)．

BS_i：Join-(i) ノードのアウタ・リレーションの1ページを格納するためのバッファのサイズ（単位はタプル数）

IBS_i：Join-(i) ノードのインナ・リレーションの1ページを格納するためのバッファのサイズ（単位はタプル数）．

n：結合演算ノード数．

R_i：Join-(i) ノードのアウタ・リレーションのタプル数．

R：Join-(n) ノードのインナ・リレーションのタプル数．

IR_i：Join-(i) ノードのインナ・リレーションのタプル数．

本方法では，結合選択率があらかじめ予測されていることを前提とした．この予測の手法については文献14) 等で提案が行われている．

ストリーム指向型関係演算処理方式では，アウタ・リレーション・ページおよびインナ・リレーション・ページ内のタプル間の比較アルゴリズムとして，nested-loop アルゴリズムあるいは sort-search アルゴリズムを対象とする．sort-search アルゴリズムでは，まず，アウタ・リレーション・ページ内のタプル群を演算対象属性上でソートし，そのページに対して，インナ・リレーション・ページ内の各タプルとバイナリ・サーチにより比較する．

ここでは，計算回数に関係のない各インナ・リレーション用バッファ（IBS_i）の割り当てはすでに終えているものとし，各ノードに割り当てるアウタ・リレーション用バッファの最適な割り当てを求める問題を考える．

各々の関係演算ノードの計算回数は，nested-loop および sort-search の場合に，それぞれ次のようになる．

nested-loop の場合：

Node 1: $R_1*(R_2*jsf_2)*\ldots*(R_n*jsf_n)*R$

Node 2: $\lceil R_1/BS_1\rceil*R_2*(R_3*jsf_3)*\ldots*(R_n*jsf_n)*R$

⋮　⋮

Node i: $\lceil R_1/BS_1\rceil*\ldots*\lceil R_{i-1}/BS_{i-1}\rceil*R_i*(R_{i+1}*jsf_{i+1})*\ldots*(R_n*jsf_n)*R$

⋮　⋮

Node $n-1$: $\lceil R_1/BS_1\rceil*\ldots*\lceil R_{n-2}/BS_{n-2}\rceil*R_{n-1}*(R_n*jsf_n)*R$

Node n: $\lceil R_1/BS_1\rceil*\ldots*\lceil R_{n-1}/BS_{n-1}\rceil*R_n*R$　(1)

sort-search の場合：

Node 1: $\lceil R_1/BS_1\rceil*(\log_2 BS_1+jsf_1*BS_1)*(R_2*jsf_2)*\ldots*(R_n*jsf_n)*R$

Node 2: $\lceil R_1/BS_1\rceil*\lceil R_2/BS_2\rceil*(\log_2 BS_2+jsf_2*BS_2)*(R_3*jsf_3)*\ldots*(R_n*jsf_n)*R$

⋮　⋮

Node i: $\lceil R_1/BS_1\rceil*\ldots*\lceil R_{i-1}/BS_{i-1}\rceil*\lceil R_i/BS_i\rceil*(\log_2 BS_i+jsf_i*BS_i)*(R_{i+1}*jsf_{i+1})*\ldots*(R_n*jsf_n)*R$

⋮　⋮

Node $n-1$: $\lceil R_1/BS_1\rceil*\ldots*\lceil R_{n-2}/BS_{n-2}\rceil*\lceil R_{n-1}/BS_{n-1}\rceil*(\log_2 BS_{n-1}+jsf_{n-1}*BS_{n-1})*(R_n*jsf_n)*R$

Node n: $\lceil R_1/BS_1\rceil*\ldots*\lceil R_{n-1}/BS_{n-1}\rceil*\lceil R_n/BS_n\rceil*(\log_2 BS_n+jsf_n*BS_n)*R$　(2)

したがって，総計算回数は，

nested-loop の場合：

$$TC=\sum_{i=0}^{n-1}\left\lceil\prod_{k=1}^{i}\left(\left\lceil\frac{R_k}{BS_k}\right\rceil\right)*R_{i+1}*\prod_{k=i+2}^{n}(jsf_k*R_k)*R\right\rceil \quad (3)$$

sort-search の場合：

$$TC=\sum_{i=1}^{n}\left\lceil\prod_{k=1}^{i-1}\left(\left\lceil\frac{R_k}{BS_k}\right\rceil\right)*\left\lceil\frac{R_i}{BS_i}\right\rceil*(\log_2 BS_i+jsf_i*BS_i)*\prod_{k=i+1}^{n}(jsf_k*R_k)*R\right\rceil \quad (4)$$

となる．

この総計算回数 TC を最小とするバッファ資源割り当てを限られたバッファ資源 BS ($=BS_1+BS_2+\ldots+BS_n$)（単位：タプル数）のもとで求める問題を解くために，すべての場合のバッファ割り当てに対して総計算回数 TC を計算し，その中から計算回数を最小とするバッファ割り当て方法を選ぶ方法が考えられる．しかし，この方法によるバッファ資源割り当てのための計算回数は，$\bar{O}(BS^n)$ となるので一般的には有効ではない．そこで，我々の方式では，以下に述べる方法により最適なバッファ資源割り当てのための計算を行う．

まず，TC を最小とするバッファ割り当てを求める問題を，次に示す TC' を最小とするバッファ割り当てを求める問題に変換する．すなわち，

nested-loop の場合：

$$TC'=\sum_{i=0}^{n-1}\left\lceil\prod_{k=1}^{i}\left(\frac{R_k}{BS_k}\right)*R_{i+1}*\prod_{k=i+2}^{n}(jsf_k*R_k)*R\right\rceil \quad (5)$$

sort-search の場合：

$$TC'=\sum_{i=1}^{n}\left\lceil\prod_{k=1}^{i-1}\left(\frac{R_k}{BS_k}\right)*\frac{R_i}{BS_i}*(\log_2 BS_i+jsf_i*BS_i)*\prod_{k=i+1}^{n}(jsf_k*R_k)*R\right\rceil \quad (6)$$

を最小とする問題を考える．

nested-loop の場合，BS_n については総計算回数と関係がないので，TC' の最小値を求める計算から BS_n は決定されない．理論的には，BS_n の最小値は 1（タップル）である．しかし，この場合，リレーション R が結合演算 Join-(n) と別のサイトにあるときはサイト間のデータ転送回数が多くなり，またリレーション

R が主記憶に入り切らない場合は I/O コストが増えることになる．したがって，それらのオーバヘッドを関係演算のストリーム型並列性に影響させないように BS_n を設定する必要がある．すなわち，BS_n は計算回数の最小化とは関係なく，物理的制約により決定されることになる．以下では，sort-search に関しては BS_1 から BS_n までのバッファ割り当てについて議論を行うが，nested-loop に関しては BS_n は 物理的制約によってすでに決定されているものとし，BS_1 から BS_{n-1} までのバッファ割り当てについて議論を行う．

ここで，$TC'(BS_1, BS_2, ..., BS_n)$ の値の最小値を限られたバッファ資源 $BS(=BS_1+BS_2+...+BS_n)$ のもとで求める問題を解くために，拡大ラグランジュ関数法[4] を用いる．この方法では，$BS_1+BS_2+...+BS_n=BS$ という制約条件のある関数 $TC'(BS_1, BS_2, ..., BS_n)$ の最小値を求める問題を，TC' の拡大ラグランジュ関数

$$ELF=TC'(BS_1, BS_2, ..., BS_n)+\beta*\left(\sum_{i=1}^{n} BS_i-BS\right)+\mu*\left(\sum_{i=1}^{n} BS_i-BS\right)^2 \quad (7)$$

の無制約条件での最小値を求める問題として扱う．バッファサイズ BS_i $(BS_i>0,\ i=1,2,...,n)$ (nested-loop では $i=1,2,...,n-1$) を変数とする関数 TC' の拡大ラグランジュ関数は微分可能であるので，勾配に関する情報を用い，この関数の値が減少する方向へ試行点を動かしながら，この関数の最小点に近づける．

この計算により，各ベース・リレーションのタプル数および jsf（結合演算選択率）の値を入力として，TC' を最小にするバッファ割り当てが定まる．

しかし，拡大ラグランジュ関数法により得られたバッファ配置は，TC' を最小とするバッファ配置であり，実際の総計算量 TC を最小とするバッファ配置ではない．

そこで，総計算量 TC の最小値に近づけるように，バッファ割り当ての修正が必要となる．

ここでは，チェイン型問い合わせに対して，修正を行うための一つの条件および一つの定理を導出し，それらに基づくバッファ割り当ての修正アルゴリズムを提示する．

まず，d_i および I_i を次のように定義する．

d_i：各関係演算ノード Node-(i) $(i=1,2,...,n)$ のアウタ・リレーション用バッファ・サイズ BS_i に依存する Node-($i+1$) の演算回数．すなわち，

$d_i=\lceil R_i/BS_i \rceil$.

($d_i=1$ のときは，Node-(i) のアウタ・リレーションがバッファ BS_i に入り切る場合に対応する．)

I_i：バッファ割り当ての修正を行った後，関係演算ノード群 Node-(j) $(j=1,2,...,n)$ のアウタ・リレーション用バッファのサイズの変化により引き起こされる関係演算ノード Node-(i) の演算回数の変化．

$I_i>0$：演算回数増加．

$I_i=0$：変化なし．

$I_i<0$：演算回数減少．

この修正問題は，I_i $(i=1, 2, ..., n)$ を求める問題となる．以後，$(I_1, I_2, ..., I_n)$ を修正方法とよぶ．

ここで，任意の修正方法に対して，各ノードの演算回数が修正する前の演算回数より多くならないための条件は次のようになる．

条件1：

$(d_1+I_1)\leq d_1$

$(d_1+I_1)*(d_2+I_2)\leq d_1*d_2$

$(d_1+I_1)*(d_2+I_2)*(d_3+I_3)\leq d_1*d_2*d_3$

$\vdots$

$(d_1+I_1)*(d_2+I_2)*...*(d_i+I_i)\leq d_1*d_2*...*d_i$

$\vdots$

$(d_1+I_1)*(d_2+I_2)*...*(d_n+I_n)\leq d_1*d_2*...*d_n$

各条件式の左辺は修正を行った後の各関係演算ノードの演算回数，条件式の右辺は修正する前の各関係演算ノードの演算回数を表している．

各条件式を満たすようなバッファ割り当ての修正を行う場合，各 I_i $(i=1,...,n)$ の決定は，I_1 から順に I_n まで行っていく．すなわち，Node-(1) のアウタ・リレーション用バッファから Node-(n) のアウタ・リレーション用バッファまで順に修正を行う．この場合，各バッファの修正は，生産者ノードのアウタ・リレーション用バッファの領域の一部を消費者ノードのアウタ・リレーションのバッファ領域へ移動することにより行う．その理由は以下のとおりである．

最初の条件式より，$I_1\leq 0$ である．これは，関係演算ノード Node-(1) の生産者側ノード群 Node-(i) $(i=2,...,n)$ のアウタ・リレーション用バッファ領域の一部を消費者側ノード Node-(1) のアウタ・リレーション用バッファに各条件式を満たす範囲内で移すことにより実現される．一般に，Node-(k) $(1\leq k\leq n)$ については，その消費者側の関係演算ノード群 Node-(i) $(i=1,...,k-1)$ のアウタ・リレーション用バッファの修

正を終えた後，生産者側の関係演算ノード群 Node-(j) ($j=k+1,\ldots,n$) のアウタ・リレーション用バッファの領域の一部を各条件式を満たす範囲内で Node-(k) のアウタ・リレーション用バッファに移すことにより，関係演算ノード Node-(k) のアウタ・リレーション用バッファの修正が実現される．

次に，あるノードのアウタ・リレーション用バッファの修正を行う場合において，その生産者側ノード群のアウタ・リレーション用バッファ領域をそのノードのアウタ・リレーション用バッファ領域へ移すための指針となる定理を示す．

定理 1：Node-(k) のアウタ・リレーション用バッファにその消費者側ノード群のアウタ・リレーション用バッファからバッファ領域を移動する場合，Node-(r) ($r>k$) と Node-(s) ($s>r>k$) のアウタ・リレーション用バッファからのバッファ領域の移動量が異なる二つの修正方法 a)，b) を考える．

a) $(I_1,\ldots,I_{k-1},I_k,I_{k+1},\ldots,I_{r-1},I_r,I_{r+1},\ldots,I_{s-1},I_s,I_{s+1},\ldots,I_n)$,

b) $(I_1,\ldots,I_{k-1},I_k',I_{k+1},\ldots,I_{r-1},I_r',I_{r+1},\ldots,I_{s-1},I_s',I_{s+1},\ldots,I_n)$.

ここで，$I_k=I_k'<0$, $I_r'>I_r\geqq 0$, $I_s>I_s'\geqq 0$ とする．

$(d_r+I_r')*(d_s+I_s')\geqq(d_r+I_r)*(d_s+I_s)$ が成り立つとき，

a) の修正方法は，b) の修正方法より演算回数を少なくする．

証明：修正方法 a) と修正方法 b) を適用した場合，ノード Node-(i) ($i=1,\ldots,r$) に関して，各ノードの演算回数は変わらない．

修正方法 a) を適用した場合，Node-(i) ($i=r+1,\ldots,s$) に関して，各ノードの演算回数は，

$$CNA_i=\prod_{j=1}^{r-1}(d_j+I_j)*(d_r+I_r)*\prod_{j=r+1}^{i-1}(d_j+I_j)$$
$$(i=r+1,\ldots,s)$$

となる．

修正方法 b) を適用した場合，Node-(i) ($i=r+1,\ldots,s$) に関して，各ノードの演算回数は，

$$CNB_i=\prod_{j=1}^{r-1}(d_j+I_j)*(d_r+I_r')*\prod_{j=r+1}^{i-1}(d_j+I_j)$$
$$(i=r+1,\cdots,s)$$

となる．

$I_r'>I_r$ であるので，$CNB_i>CNA_i$ ($i=r+1,\ldots,s$) となる．

修正方法 a) を適用した場合，Node-(i) ($i=s+1,\ldots,n$) に関して，各ノードの演算回数は，

$$CNA_i=\prod_{j=1}^{r-1}(d_j+I_j)*(d_r+I_r)*\prod_{j=r+1}^{s-1}(d_j+I_j)$$
$$*(d_s+I_s)*\prod_{j=s+1}^{i-1}(d_j+I_j)$$
$$(i=s+1,\ldots,n)$$

となる．

修正方法 b) を適用した場合，Node-(i) ($i=s+1,\ldots,n$) に関して，各ノードの演算回数は，

$$CNB_i=\prod_{j=1}^{r-1}(d_j+I_j)*(d_r+I_r')*\prod_{j=r+1}^{s-1}(d_j+I_j)$$
$$*(d_s+I_s')*\prod_{j=s+1}^{i-1}(d_j+I_j)$$
$$(i=s+1,\ldots,n)$$

となる．

$(d_r+I_r')*(d_s+I_s')\geqq(d_r+I_r)*(d_s+I_s)$ であるので，$CNB_i\geqq CNA_i$ ($i=s+1,\ldots,n$) となる．

ゆえに，Node-(i) ($i=1,\ldots,n$) に関して $CNB_i\geqq CNA_i$ となり，修正方法 a) は，修正方法 b) より演算回数を少なくする．　[証明終り]

この定理は，関係演算ノードの演算回数を減らすために，条件 1 を満たす範囲内において，Node-(n)，あるいは，なるべく Node-(n) に近い関係演算ノードのアウタ・リレーション用バッファ領域を消費者側の関係演算ノードのアウタ・リレーション用バッファ領域に移せばよいことを示している．

次に，上で述べた条件 1 および定理 1 に基づく修正アルゴリズムを示す．アルゴリズムの中で使うパラメータは，次のとおりである．

n：関係演算ノードの数．

R_i：関係演算ノード Node-(i) ($i=1,\ldots,n$) のアウタ・リレーションのタプル数．

BS_k：関係演算ノード Node-(k) ($k=1,\ldots,n$) のアウタ・リレーション用バッファのタプル数．

k：修正の対象となっている関係演算ノードおよびそのアウタ・リレーション用バッファの識別子．

t：Node-(k) のアウタ・リレーション用バッファ BS_k へバッファ領域を移動する対象となる生産者側ノード群の最上位の関係演算ノードのアウタ・リレーション用バッファの識別子 (BS_i ($i=t,\ldots,n$) の領域の一部を消費者側の関係演算ノード Node-(k) のアウタ・リレーション用バッファ BS_k に移すことになる)．

バッファ修正アルゴリズム：

ステップ 1：$k=0$.

ステップ 2：$k=k+1$ とする．

$k=n$ ならば，修正が完了する．

$k<n$ ならば，ステップ3へ移る．

ステップ3：d_k の値を $\lceil R_k/BS_k \rceil$ とする．

$d_k=1$ ならば，$BS_k \geqq R_k$ なので，(BS_k-R_k) の領域をバッファ BS_{k+1} に移し，$BS_k=R_k$ に修正し，ステップ2へ移る．

$d_k>1$ ならば，d_i $(i=k+1,\ldots,n)$ を計算し，$t=n$ とし，ステップ4へ移る．

ステップ4：Node-(k) について，$I_k=-1$ とできるかどうか調べる．すなわち，d_k を“1”減らせるかどうか (Node-$(k+1)$ の演算回数を1回減らせるかどうか) 調べる．$I_k=-1$ とした場合，定理1に基づいて条件1を満たすようなバッファ修正が存在するかどうかを I_i $(i=t,\ldots,n)$ について調べる．

そのような修正が存在する場合には，$BS_i'=R_i/(d_i+I_i)$ $(i=t,\ldots,n)$ を満たすような BS_i' を求める．そして，BS_i-BS_i' の領域をバッファ BS_k へ移し，$BS_i=BS_i'$ と修正する．そして，ステップ3へ移る．

そのような修正が存在しない場合には，ステップ5へ移る．

ステップ5：$t=t-1$ とする．

$t=k$ ならば，$BS_k'=R_k/d_k$ を計算し，BS_k-BS_k' の領域をバッファ BS_{k+1} に移し，$BS_k=BS_k'$ と修正し，ステップ2へ移る．$t>k$ ならば，ステップ4へ戻る．

次に，例として，4.1節において後述する Query1 (図4，表2) の各関係演算ノードのアウタ・リレーション用バッファに対し，ここで提示した資源割り当て計算方式によりバッファ資源を割り当てる場合を示す．

Query1 の総計算回数を求める式 TC は，3.1節の(4)式に，各演算対象のベース・リレーションのタプル数および各結合演算結合率を代入することにより得られる．3.1節の(6)式より拡大ラグランジュ関数法を用いて，限られたバッファ資源 BS $(BS_1+BS_2+BS_3=1800)$ のもとで TC' を最小とするバッファ配置を計算すると次のようになる．

$BS_1=781$, $BS_2=637$, $BS_3=382$.

次に，修正アルゴリズムを用いて，TC の最小値を求めると，最適なバッファ資源の割り当て

$BS_1=1024$, $BS_2=512$, $BS_3=264$

が得られる．

3.2 並列性抽出のための資源割り当て

ここでは，異なるプロセッサに配置された関係演算ノード間でストリーム型の並列性を抽出するためのバッファ資源割り当ての条件を示す．図3に示すような

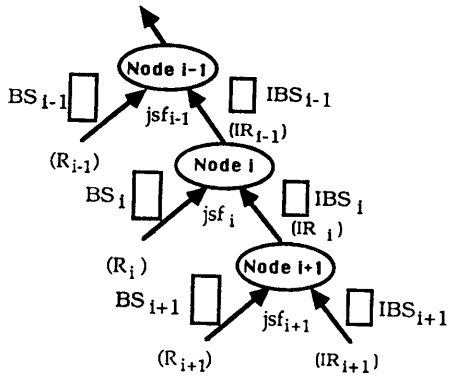


図3 チェイン型問い合わせ
Fig. 3 A chain query.

結合演算ノード (Join-(i), Join-$(i+1)$) からなる問い合わせの一部（チェイン型問い合わせ）を考える．この場合，Join-(i) ノードが2つの入力バッファ (BS_i, IBS_i) 内に格納されている2ページに対して結合演算を行っている間に，Join-$(i+1)$ ノードが Join-(i) のインナ・リレーションの次ページの生成を完了できれば，Join-(i) ノードにおいて実行の中断は生じない．すなわち，Join-(i) ノードにおける演算実行の中断は，Join-$(i+1)$ によって生成される次ページの生成が遅れることによって引き起こされる．

ここでは，Join-(i) ノードの実行を中断させないための条件を設定する．

2ページ間のタプル群の比較アルゴリズムとして nested-loop を用いた場合にストリーム型の並列性を引き出すための条件は，すでに文献8)で示したように，次のとおりである．

$BS_i \geqq 1/jsf_{i+1}$.

比較アルゴリズムとして，sort-search を用いた場合の条件は次のようになる．ここでは，Join-(i) および Join-$(i+1)$ ノードの各々のアウタ・リレーション・ページ内のタプルは，すでに演算対象属性上でソートされているものとする．

Join-(i), Join-$(i+1)$ ノードでは各々，結合演算属性上でソートされたアウタ・リレーションのページに対してバイナリサーチ・アルゴリズムによって結合演算を行う．

Join-(i) ノードに対して，すでにソートされたアウタ・リレーションのページとインナ・リレーション・ページ内のタプル群との比較回数は次のとおりである．

$$IBS_i*(\log_2 BS_i+jsf_i*BS_i). \qquad (8)$$

Join-(i) ノードのインナ・リレーションの1ページ

(IBS_i 個のタプル数) を生成するための Join-(i+1) ノードにおける比較回数は次のようになる．(ここで，$1/jsf_{i+1}$ は，nested-loop アルゴリズムの場合に，1 タプルを生成するのに必要な平均比較回数である．)

$$IBS_i * (\log_2 BS_{i+1} + jsf_{i+1} * BS_{i+1}) / (jsf_{i+1} * BS_{i+1}). \quad (9)$$

ここで，演算結果のタプル群の出力バッファへの書き込み時間を無視できるとすると，Join-(i) ノードが中断なく結合演算を行うための条件は次のようになる．

(8)≧(9)，すなわち，

$$\log_2 BS_i + jsf_1 * BS_i \geq (\log_2 BS_{i+1} + jsf_{i+1} * BS_{i+1}) / (jsf_{i+1} * BS_{i+1}). \quad (10)$$

異なるプロセッサに配置された 2 ノード (生産者ノードと消費者ノード) 間で(10)式の条件が成立する場合に，ストリーム型の並列性が最大限に抽出されることになる．

3.3 実際の資源割り当て

実際の処理系で，問い合わせを 1 台のプロセッサにより処理する場合には，3.1 節で示したバッファ資源割り当て計算により各関係演算ノードへのバッファ割り当てを決める．一つの問い合わせを複数台のプロセッサにより処理する場合には，まず，問い合わせを構成する関係演算群およびプロセッサ台数，各プロセッサの内部メモリ容量，ベース・リレーションの存在する場所等から，各関係演算ノードを実行するプロセッサを決定する．この場合，3.1 節において述べたように，1 プロセッサに複数の関係演算ノードを割り当てることができる．ここでは，各プロセッサへの関係演算ノードの配置はすでに終えられているものとし，各プロセッサに割り当てられた関係演算ノード群 (サブ・キュアリ) に対して，3.1 節および 3.2 節の条件に基づき，実際のバッファ資源割り当てを考える．また，簡単のために，各プロセッサのパイプライン・ノード (チェイン型問い合わせにおいてインナ・リレーションを生成するノード) において，タプル間の比較アルゴリズムとして，nested-loop を用いる場合を考える．図 3 のように，パイプライン・ノード Node-(i−1), Node-(i), Node-(i+1) が異なる 3 台のプロセッサに割り当てられた場合，各パイプライン・ノードのアウタ・リレーション用バッファ量は，3.2 節で示した nested-loop における並列性抽出のための条件を満たすか否か，および，アウタ・リレーション全体を格納できるか否か(再計算が必要としないか否か)により 4 通りに分類できる．さらに，3 ノード (Node-(i−1), Node-(i), Node-(i+1)) 間のストリーム型並列性を検討するために，その並列性に関連するアウタ・リレーション用バッファの量 BS_{i-1} および BS_i を組み合わせて場合分けすると，問い合わせ処理の状態は**表 1** に示すような 16 通りに分類される．表 1 において，下線が引かれているノードは，ストリーム型並列処理においてボトルネックとなる (2 ノードに下線が引かれている場合は，どちらか一方がボトルネックとなる) ノードであることを示す．表 1 より，各パイプライン・ノード間で並列性抽出のための条件を満たすか，あるいは，アウタ・リレーション全体を格納するか，どちらか一方を満たすようにバッファ資源を配置することが必要であることがわかる．どちらも満足しない場合 (表 1 の case 1, 2, 3, 4, 5, 9, 13 の場合) には，再計算を行うノードがボトルネックになり，本方式の有効性が活かされない．このことから，異なるプロセッサに配置された各パイプライン・ノードのアウタ・リレーション用バッファには，3.2 節で述べた並列性を抽出する条件を満たすようにバッファ資源を割り当てるか，あるいは，アウタ・リレーション全体を格納するバッファ資源を割り当てることにより，本方式の効果が引き出せることがわかる．また，各プロセッサ内のサブ・キュアリの各関係演算ノードに関しては，3.1 節で示したバッファ資源割り当て計算により，各ノードへのバッファ割り当てを決定する．

4. 実　験

ストリーム指向型関係演算処理方式の性能を評価する環境を実現するために，Sun-2 ワークステーション[13] (プロセッサ：MC68010, OS：UNIX4.2BSD)上にストリーム指向型関係演算処理系を開発した[9),10)]．この処理系は，2 章で述べた処理方式により，実際に問い合わせ処理を行い，また，各関係演算の実行時間を実測する環境を備えている．また，複数台のプロセッサによる並列処理の効果を評価するために，この関係演算処理系上に並列処理の評価環境を実現した．並列処理の評価環境では，プロセッサ間のデータ転送 (ストリームデータおよびデマンドの送出) 時間だけは計算式によって求め，関係演算の処理時間は上述の処理系を用いて測定した実測値を用いる．データ転送の速度については，プロセッサ間データ転送速度を 16 msec/2k-byte packet に設定した[10),12)]．問い合わせの実行時間は，Sun-2 ワークステーションによって

表 1 関係演算ノード間の計算回数の比較と並列性

Table 1 Comparisons of the computation times among relational operation nodes and parallelism.

BS_i Size \ BS_{i-1} Size	$BS_{i-1}<1/jsf_i$, $BS_{i-1}<R_{i-1}$	$1/jsf_i \leq BS_{i-1}<R_{i-1}$	$R_{i-1}=BS_{i-1}<1/jsf_i$	$R_{i-1}=BS_{i-1}, 1/jsf_i \leq BS_{i-1}$
$BS_i<1/jsf_{i+1}$, $BS_i<R_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 1) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $>R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_{i-1}/BS_{i-1})*(R_i/BS_i)$ $*R_{i+1}*IR_{i+1}$ $>(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 2) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $\leq R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_{i-1}/BS_{i-1})*(R_i/BS_i)$ $*R_{i+1}*IR_{i+1}$ $>(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 3) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $>R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_i/BS_i)*R_{i+1}*IR_{i+1}$ $>R_i*jsf_{i+1}*R_{i+1}*IR_{i+1}$ $=R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 4) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $\leq R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_i/BS_i)*R_{i+1}*IR_{i+1}$ $>R_i*jsf_{i+1}*R_{i+1}*IR_{i+1}$ $=R_i*IR_i$
$1/jsf_{i+1} \leq BS_i<R_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 5) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $>R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_{i-1}/BS_{i-1})*(R_i/BS_i)$ $*R_{i+1}*IR_{i+1}$ $\leq(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 6) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $\leq R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_{i-1}/BS_{i-1})*(R_i/BS_i)$ $*R_{i+1}*IR_{i+1}$ $\leq(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 7) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $>R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_i/BS_i)*R_{i+1}*IR_{i+1}$ $\leq R_i*jsf_{i+1}*R_{i+1}*IR_{i+1}$ $=R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 8) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $\leq R_{i-1}*IR_{i-1}$ Node $i+1$: $(R_i/BS_i)*R_{i+1}*IR_{i+1}$ $\leq R_i*jsf_{i+1}*R_{i+1}*IR_{i+1}$ $=R_i*IR_i$
$R_i=BS_i<1/jsf_{i+1}$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 9) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $>R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: (R_{i-1}/BS_{i-1}) $*R_{i+1}*IR_{i+1}$ $>(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 10) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $\leq R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: (R_{i-1}/BS_{i-1}) $*R_{i+1}*IR_{i+1}$ $>(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 11) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $>R_{i-1}*IR_{i-1}$ Node $i+1$: $R_{i+1}*IR_{i+1}$ $>R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 12) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $\leq R_{i-1}*IR_{i-1}$ Node $i+1$: $R_{i+1}*IR_{i+1}$ $>R_i*IR_i$
$R_i=BS_i, 1/jsf_{i+1} \leq BS_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 13) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $>R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: (R_{i-1}/BS_{i-1}) $*R_{i+1}*IR_{i+1}$ $\leq(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 14) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $\leq R_{i-1}*jsf_i*R_i*IR_i$ $=R_{i-1}*IR_{i-1}$ Node $i+1$: (R_{i-1}/BS_{i-1}) $*R_{i+1}*IR_{i+1}$ $\leq(R_{i-1}/BS_{i-1})*R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 15) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $>R_{i-1}*IR_{i-1}$ Node $i+1$: $R_{i+1}*IR_{i+1}$ $\leq R_i*IR_i$	Node $i-1$: $R_{i-1}*IR_{i-1}$ (case 16) Node i: $(R_{i-1}/BS_{i-1})*R_i*IR_i$ $=R_i*IR_i$ $\leq R_{i-1}*IR_{i-1}$ Node $i+1$: $R_{i+1}*IR_{i+1}$ $\leq R_i+IR_i$

——: Bottleneck Node

モニタされる実際の CPU 時間およびディスクアクセス処理等のシステムコール処理の実行時間の実測値である．また，バッファ資源割り当て計算に要する処理時間については，Sun-2 ワークステーション上に浮動小数点演算用補助プロセッサを付加した環境でバッファ資源割り当て計算を行うプログラムの実行時間を実測した．

4.1 問い合わせ

関係演算処理系を評価する場合に生じる問題は，評価用の問い合わせの選択である．ここでは，複数の結合演算からなるチェイン型問い合わせの処理時間を評価する．結合演算は関係演算の中で最も重要でかつ長い処理時間を要する演算として知られている．問い合わせの中には，選択演算や射影演算が含まれている場合がより一般的であるが，ここでは結合演算間でのストリーム指向型処理方式の効果を明確にするために，結合演算だけからなる問い合わせを対象とする．関係データベースの問い合わせ処理では，一般に選択演算，射影演算は各々，結合演算群の前および後に行われるので，ここで示す問い合わせ処理の評価環境は結合演算間の処理時間の測定に関して一般的であると考えられる．ここでは，3つの結合演算からなる4種類の問い合わせを用いる．それらの問い合わせは図 4 に示すような構造を持つ．

ベース・リレーションのタプル数，結合演算結合率は，各問い合わせにおいて**表 2** のように設定した．

Query 1 は，中間リレーションがベース・リレーションと同じサイズとなる場合である．

Query 2 は，Join-2 (Node-2) がベース・リレーションの2倍のタプル数 (2048 タプル) をもつ中間リレーションを生成し，その中間リレーションの大きさを保って最後のノードまで受け渡される場合である．

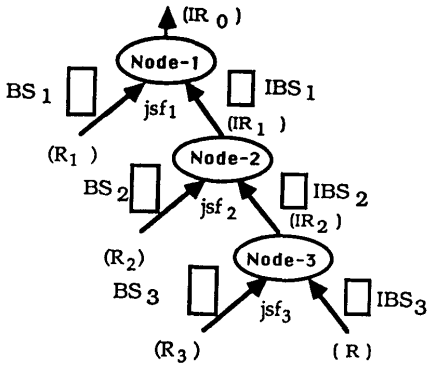


図 4 評価用問い合わせの構造
Fig. 4 A structure of evaluated queries.

表 2 評価用問い合わせにおけるベース・リレーション・サイズおよび結合演算結合率
Table 2 Base-relation sizes and join-selectivity-factors in each evaluated queries.

		Query 1	Query 2	Query 3	Query 4
R_1, R_2, R_3, R		1024	1024	1024	1024
Node 3	jsf_3	1/1024	1/512	1/512	1/2048
	IR_3	1024	1024	1024	1024
Node 2	jsf_2	1/1024	1/1024	1/512	1/2048
	IR_2	1024	2048	2048	512
Node 1	jsf_1	1/1024	1/1024	1/512	1/2048
	IR_1	1024	2048	4096	256
Output relation	IR_0	1024	2048	8192	128

($R_1 \sim R_3$, $IR_0 \sim IR_3$, R: the number of tuples)

Query 3 は，各結合演算ノード (Node-1, Node-2, Node-3) が各々の演算対象のインナ・リレーションのタプル数の2倍のタプル数を生成する場合である．

Query 4 は，各結合演算ノードが各々の演算対象のインナ・リレーションのタプル数の 1/2 倍のタプル数を生成する場合である．

ストリーム指向型関係演算処理方式では，結合演算結合率およびバッファサイズが変わらなければ，通信トラフィックの頻度，1回のデータ転送量，およびページ間の演算時間は変化しない．すなわち，それらは，ベース・リレーションのタプル数には依存しない．したがって，ベース・リレーションのタプル数の変化による問い合わせ処理時間の変化に関する実験結果の詳細については，ここでは提示しない．

ベース・リレーションのタプル数は，各問い合わせにおいて各々同じ値 (1024 タプル) に設定する．各ベース・リレーションのタプル長は 64 バイトとし，各タプルは結合演算対象属性として2つの整数型属性 (各4バイト) を含む．それらの整数型属性内の値は一様分布の乱数を用いた．

結合演算においてインナ・リレーションの1ページとアウタ・リレーションの1ページを比較するアルゴリズムとしては，sort-search アルゴリズムを用いる．

4.2 実験結果

実験結果を表3に示す Query 1, Query 2, Query 3, Query4 について，case1 は，全結合演算ノードのアウタ・リレーション用バッファの総和 (タプル数：BS) が $BS=1800$ (タプル) のとき，最適化を行わず，各結合演算ノードのアウタ・リレーション用バッファに

均等にバッファ領域を割り当てた場合である．

Query 1, Query 2, Query 3, Query 4 において，case 2 は，3.1 節で示したバッファ資源割り当て計算（拡大ラグランジュ関数法によりバッファ割り当ての値を求め，修正アルゴリズムによってその値を修正する計算）の結果にしたがって割り当てた場合である．本実験で行った問い合わせ処理では，関係演算ノードが少なく，また，バッファ・サイズがリレーション・サイズと比べて比較的大きい場合なので，バッファ資源の割り当ての結果は単純なものとなった．しかし，複雑な問い合わせの場合，あるいは，バッファ資源（*BS*）が小さい場合には，必ずしもこのような単純な結果とはならない．

各問い合わせの case2 について，バッファ資源割り当て計算を行う処理時間（*BAC*）を**表 3** に示した．この処理時間は関係演算の並列処理時間と比較して，1/97～1/68 であり，また，逐次処理の場合の処理時間と比較して，1/250～1/130 である．したがって，この計算のためのオーバヘッドはほとんど無視できると考えられる．

Query 1-4 において，case 3 は，各アウタ・リレーションがすべて，アウタ・リレーション用バッファに入り切る場合である．また，Query 1, Query 2, Query 3 では，case3 は，3.2 節で示したパイプライン性を抽出するための条件も満たしている．

また，Query 1 における case 3-case 7 は，各アウタ・リレーションがすべて，アウタ・リレーション用バッファに入り切る場合である．ただし，各 case 間では，インナ・リレーション・ページのグラニュラリティが異なっている．インナ・リレーション・ページのグラニュラリティは，プロセッサ間でのストリームデータの転送回数，デマンドの発行回数に影響を与える．

各 case の逐次実行（*SP*）および並列実行（*PP*）の場合の実行時間および逐次実行時間と並列実行時間の比を表 3 に示す．逐次実行の場合は，各結合演算が 2.1 節で述べた方式に従って，1 台のプロセッサにより擬似並列に実行される．また，並列実行の場合は，

表 3 実験結果
Table 3 Experimental results.

	Query 1							Query 2			Query 3			Query 4		
	case 1	case 2	case 3	case 4	case 5	case 6	case 7	case 1	case 2	case 3	case 1	case 2	case 3	case 1	case 2	case 3
BS	1800	1800	3072	3072	3072	3072	3072	1800	1800	3072	1800	1800	3072	1800	1800	3072
BS_1	600	1024 (781)	1024	1024	1024	1024	1024	600	1024 (845)	1024	600	1024 (909)	1024	600	1024 (706)	1024
BS_2	600	512 (637)	1024	1024	1024	1024	1024	600	512 (638)	1024	600	512 (591)	1024	600	512 (649)	1024
BS_3	600	264 (382)	1024	1024	1024	1024	1024	600	264 (317)	1024	600	264 (300)	1024	600	264 (445)	1024
IBS_1	100	100	100	512	64	16	8	100	100	100	100	100	100	100	100	100
IBS_2	100	100	100	512	64	16	8	100	100	100	100	100	100	100	100	100
IBS_3	100	100	100	512	64	16	8	100	100	100	100	100	100	100	100	100
SP (sec)	82.7	52.3	22.2	28.4	22.6	36.1	62.6	105.6	65.5	38.3	141.9	99.9	63.1	68.5	46.7	17.0
PP (sec)	47.5	27.3	8.0	15.4	8.5	14.1	25.8	56.1	29.7	14.5	61.6	38.8	31.0	43.2	27.5	7.6
SP/*PP*	1.74	1.92	2.78	1.84	2.66	2.56	2.43	1.88	2.21	2.64	2.30	2.57	2.04	1.59	1.70	2.24
BAC (sec)	—	0.4	—	—	—	—	—	—	0.4	—	—	0.4	—	—	0.3	—

$BS = \sum_{i=1}^{3} BS_i$
$BS_1 \sim BS_3$: Outer-Relation Buffer
$IBS_1 \sim IBS_3$: Inner-Relation Buffer
( ): Before Revising
SP: Execution Time in Serical Processing Case
PP: Execution Time in Parallel Processing Case
BAC: Computation Time of Optimization for Buffer Allocation

各関係演算のアウタ・リレーションの各ページのアクセス，および各結合演算の実行は異なる3台のプロセッサにより実行され，また，Node-3 のインナ・リレーション R の各ページのアクセスは，他の1台のプロセッサにより実行される．

Query1 の case1 と case2 の比較において，3.1 節で述べた拡大ラグランジュ関数法に基づく資源割り当て計算およびその修正アルゴリズムによる最適化の効果が大きいことがわかる．

case2 と case3 の並列実行の場合の比較において，パイプライン性の条件が満たされ，かつ再計算のない case3 の場合，本方式の効果が最大限に発揮されることがわかる．この場合，並列処理の効果により，逐次実行時間と並列実行時間の比は 2.78 となり，高い並列性が得られることがわかる．

Query 1 の case 3-case 7 の比較により，インナ・リレーション・ページのグラニュラリティの変化と処理時間の関係がわかる．インナ・リレーション・ページのグラニュラリティが case 6, 7 のように小さい場合，デマンド転送回数の増加によりオーバヘッドが増大することを示している．また，case4 のようにグラニュラリティが大きくなると，ストリーム型の並列性が低くなっていくことにより，並列処理の効果が減り，処理時間が長くなることがわかる．case3 あるいは case 5 ではグラニュラリティの設定が適切である．一般に，インナ・リレーション・ページのグラニュラリティの設定については，プロセッサ間通信速度やネットワークのトラフィック量によって適切な値は異なる．

Query2 の case1 と case2 との比較において，拡大ラグランジュ関数法およびその修正アルゴリズムを用いた最適化の効果により，逐次処理の場合の処理時間が 65.5/105.6 に短縮され，その効果が大きいことがわかる．すなわち，限られたバッファ資源の中で問い合わせを行う case2 において，最適化により処理時間が大きく短縮されることがわかる．case2 は，アウタ・リレーションがバッファに入り切っていない場合（再計算がある場合）であり，case3 は，アウタ・リレーションがバッファに入り切っており，またパイプライン性抽出のための条件も満たしている場合である．Query1 と同様に，Query2 においても case3 が最も良い処理効率を示している．

Query3 のcase1 と case2 の比較においても，最適化の効果が大きいことがわかる．case 2 と case 3 の比較において，パイプライン性の条件が満たされ，かつ，再計算のない case3 において，本方式の効果が最大限に発揮されることがわかる．逐次実行の場合には，case2 と case3 の処理時間は 99.9 : 63.1 であり，case2 の処理時間がかなり長い．しかし，並列実行の場合には，case2 と case3 の処理時間は 38.8 : 31.8 であり，その差は大きくない．これは，case2 の場合の資源割り当てが，3.2 節で示した並列性の抽出のた

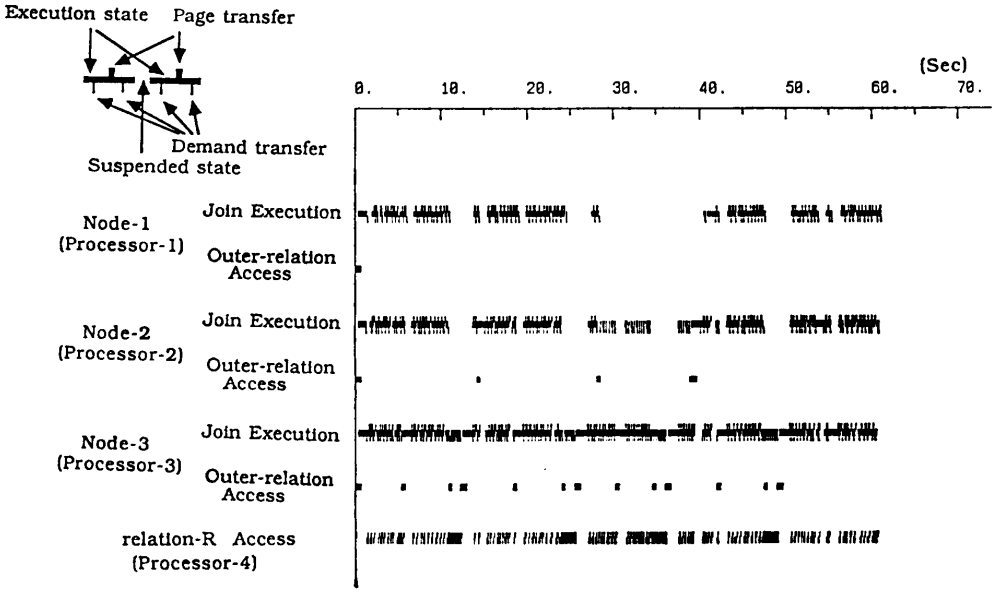


図 5 タイムチャート (Query 3, case 1)
Fig. 5 Time chart (Query 3, case 1).

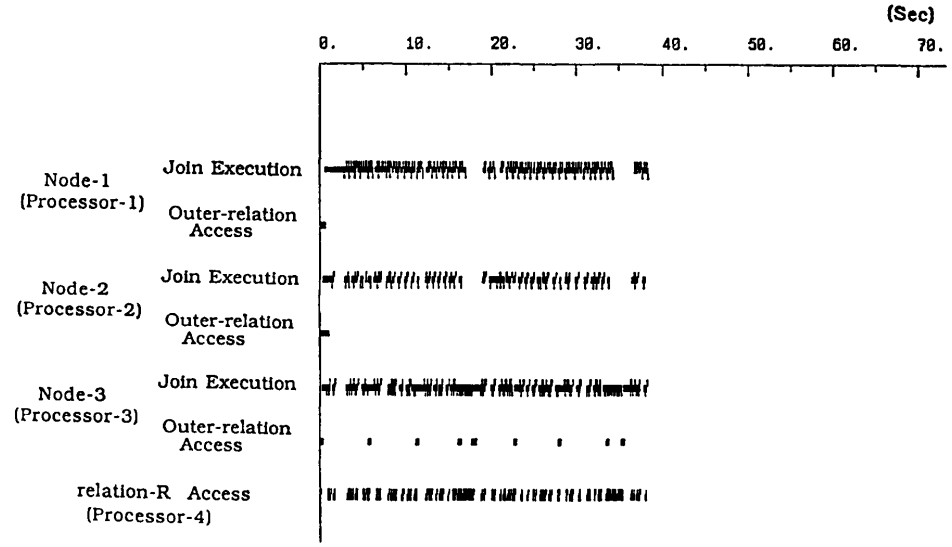


図 6 タイムチャート (Query 3, case 2)
Fig. 6 Time chart (Query 3, case 2).

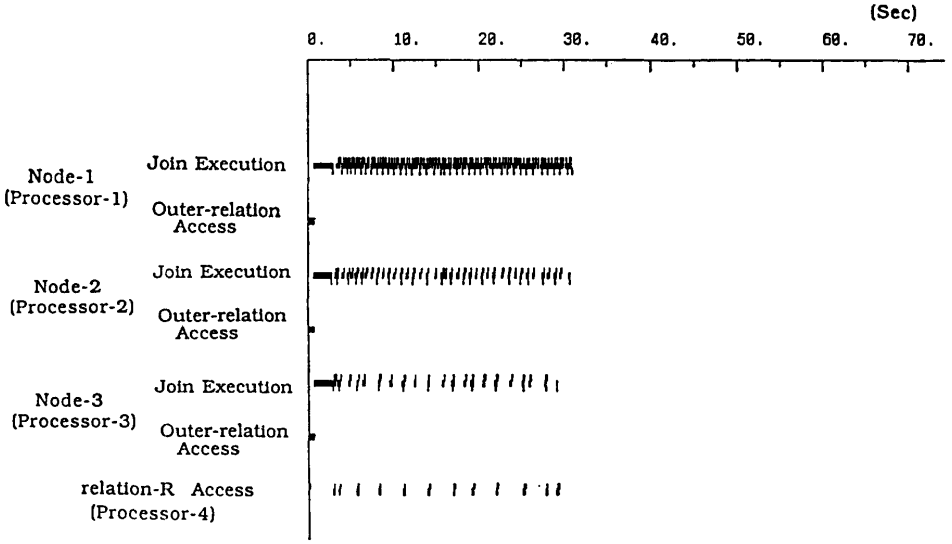


図 7 タイムチャート (Query 3, case 3)
Fig. 7 Time chart (Query 3, case 3).

めの条件を完全に満たしてはいないが，それに近い値を実現しており，生産者ノード (Node-2, Node-3) の再計算のオーバヘッドがパイプライン処理の中にかくれるからである．この場合のように，再計算回数が最小化され，かつ，パイプライン性の抽出の条件に近い値が設定できると，限られた資源の中でストリーム型の並列処理を行う本方式の有効性が顕著になることがわかる．Query3 の各 case における並列実行の様子を示すタイムチャートを各々，図5，図6，図7に示す．case1 では，Node-2, Node-3 の再計算のオーバヘッドが大きいことがわかる．case2 では，最適化の効果により Node-2, Node-3 の演算回数が軽減され，かつ，並列処理の効果がよく引き出されていることがわかる．case3 では，再計算がなく，処理時間が最も短い．

Query4 は，生産者ノード (Node-3) の負荷が重い問い合わせである．したがって，他の問い合わせと比較して，生産者ノードの再計算によるオーバヘッドが全体の処理時間に与える影響が大きい．case1 と case 2 の比較において，最適化の効果が顕著である．case2 と case3 の比較において，再計算のない case3 の場合の処理時間の改善率が他の問い合わせと比べて最も

大きい.

以上のように，各問い合わせにおいて，アウタ・リレーションが入り切ることにより再計算がなくなるcase3の場合は，最も処理時間が短縮される．しかし，限られたバッファ資源の中では，3.1節で示した資源割り当て計算により最適な資源割り当てを行うcase2において，最適化を行わないcase1に比べ，処理時間の短縮が顕著となることがわかる．また，各結合演算ノードが異なるプロセッサへ配置される並列処理環境では，3.3節で述べたように，パイプライン性抽出の条件を満たし，かつ再計算がないcase3が最も処理効率が良いことがわかる．すなわち，並列処理環境では，パイプライン・ノード群のアウタ・リレーション用バッファには，並列性抽出の条件を満たすようにバッファ資源を配置するか，あるいは，再計算をなくすためにアウタ・リレーション全体を格納するバッファ資源を配置することにより，ストリーム指向型関係演算処理方式の効果を最大限に引き出せることがわかる.

5. むすび

本論文では，ストリーム指向型関係演算処理方式の有効性を最大限に引き出すための計算機資源割り当て方法を提案した．この方式では，(1)再計算回数の軽減，(2)並列性の抽出，が重要な課題であり，(1)については，拡大ラグランジュ関数法を適用し，さらに修正アルゴリズムにより最適なバッファ資源割り当てを行う方法を示した．また，(2)については，ストリーム型の並列性を抽出のための資源割り当ての条件を示し，並列処理環境における資源配置の指針を明らかにした．本稿では，結合演算を対象として検討を行った．他の演算が含まれるような問い合わせについても並列性抽出の条件を，3.2節で示したものと同様に設定できる.

また，ストリーム指向型関係演算処理方式を実現する関係演算処理系を用いて，実際に問い合わせ処理の実験を行い，本論文で提案した資源割り当て方法の有効性を明らかにした.

ストリーム指向型関係演算処理方式では，問い合わせを構成する各関係演算は，少量の演算対象タプルがそろった時点で起動可能となるので，まず，少量のバッファ領域を各関係演算に配置して各関係演算を起動し，そこで，結合演算結合率を予測する手法を現在検討している．この手法により，実用的な予測が可能となると考えている.

現在，複数台のワークステーションを高速ネットワークにより結合したハードウェア環境の上に並列処理システムSMASHの構築を進め，また，関係データベース演算の処理系を実現している．さらに，今後，この並列処理システム上に演繹データベース等，大量データを対象とする分野の処理系を実現していく予定である.

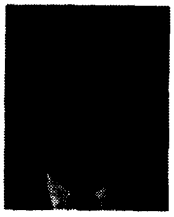
謝辞 本研究にあたり，有意義な討論をしていただいた筑波大学工学研究科加藤和彦氏に感謝いたします.

参考文献

1) Amamiya, M. and Hasegawa, R.: Dataflow Computing and Eager and Lazy Evaluations, *New Generation Computing*, Vol. 2, No. 2, pp. 105-129 (1984).
2) Friedman, D. P. and Wise, D. S.: Aspects of Applicative Programming for Parallel Processing, *IEEE Trans. Comput.*, Vol. C-27, pp. 289-296 (Apr. 1978).
3) Gallaire, H., Minker, J. and Nicolas, J. M.: Logic and Databases: a Deductive Approach, *ACM Comput. Surv.*, Vol. 16, No. 2, pp. 153-185 (June. 1984).
4) Hestenes, M. R.: Multiplier and Gradient Methods, *J. Optimization Theory and Applications*, Vol. 4, pp. 303-320 (1969).
5) Kim, W.: A New Way to Compute the Product and Join of Relations, *Proc. of the ACM-SIGMOD Conf.*, pp. 179-187 (1980).
6) 喜連川，伏見：データベースマシン，情報処理，Vol. 28, No. 1, pp. 56-67 (1987).
7) Kiyoki, Y., Isoda, M., Kojima, K., Tanaka, K., Minematsu, A. and Aiso, H.: Performance Analysis for Parallel Processing Schemes of Relational Operations and a Relational Database Machine Architecture with Optimal Scheme Selection Mechanism, *Proc. 3rd Int. Conf. Distributed Computing Systems*, pp. 196-203 (Oct. 1982).
8) 清木，長谷川，雨宮：先行・遅延評価機構を用いた関係演算処理方式，情報処理学会論文誌，Vol. 26, No. 4, pp. 685-695 (1985).
9) 清木，加藤，益田：要求駆動型制御による関係演算パイプライン処理方式の実現法，情報処理学会アドバンストデータベースシンポジウム予稿集，pp. 51-58 (1985).
10) Kiyoki, Y., Kato, K. and Masuda, T.: A Relational Database Machine Based on Functional Programming Concepts, *Proc. ACM-IEEE Computer Society Fall Joint Computer*

Conf., pp. 969-978 (Nov. 1986).
11) Kiyoki, Y., Kato, K., Yamaguchi, N. and Masuda, T.: A Stream-Oriented Approach to Parallel Processing for Deductive Databases, *Proc. 5th Int. Workshop on Database Machines*, pp. 102-115 (1987).
12) Lu, H. and Carey, M. J.: Some Experimental Results on Distributed Join Algorithms in a Local Network, *Proc. 11th Int. Conf. on VLDB*, pp. 292-304 (1985).
13) Programmers Reference Manual for the Sun Work-station, Sun Micro Systems, Inc. (1982).
14) Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G.: Access Path Selection in a Relational Database System, *Proc. of the ACM-SIGMOD Conf.*, pp. 23-34 (1979).
15) Treleaven, P. C., Brownbridge, D. R. and Hopkins, R. P.: Data-driven and Demand-driven Computer Architecture, *ACM Comput. Surv.*, Vol. 14, No. 1, pp. 93-144 (Mar. 1982).
16) Yu, C. T. and Chang, C. C.: Distributed Query Processing, *ACM Comput. Surv.*, Vol. 16, No. 4, pp. 399-433 (Dec. 1984).

(昭和62年4月27日受付)
(昭和62年9月 9 日採録)

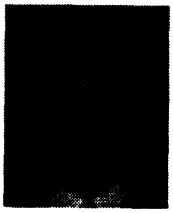


清木 康 (正会員)

昭和31年生. 昭和53年慶応義塾大学工学部電気工学科卒業. 昭和58年慶応義塾大学大学院工学研究科博士課程修了. 工学博士. 同年, 日本電信電話公社武蔵野電気通信研究所入所. 昭和59年より筑波大学電子・情報工学系に勤務, 現在同学系講師. データベースシステム, 計算機アーキテクチャ, 関数型プログラミングの研究に従事. 電子情報通信学会, ACM 各会員.

劉 澎 (正会員)

1961年生. 1982年中国南京工科大学電子計算機科学系卒業. 1986年筑波大学大学院修士課程理工学研究科修了. 現在同大学院博士課程工学研究科に在学中. データベースシステム, 並列処理, 関数型プログラミング言語に興味をもつ. ACM, IEEE 各会員.



益田 隆司 (正会員)

昭和14年生. 昭和38年東京大学工学部応用物理学科卒業. 昭和40年同大学院修士課程修了. 同年(株)日立製作所入社. 中央研究所, システム開発研究所に勤務. 昭和52年筑波大学に移り, 現在, 筑波大学電子・情報工学系教授. 工学博士. オペレーティング・システムの研究開発を経て, その方式論, 計算機システムの性能評価, データベース管理システムの設計技法の研究に従事.