大規模並列関係データベース処理における

高速化技法に関する研究

中　野　美　由　紀

もくじ

図一覧

表一覧

第 1 章

序論

1.1 研究の背景

二十一世紀に入り、情報化社会の発展はますます加速し、インターネットの急速な普及に伴い、ワールド・ワイド・ウェブ等に見られるように、個人が世界に向けて情報発信が可能となった時代を迎えている。それに伴い、様々な人類の活動の記録は、電子的データとして日々蓄積され、その容量は爆発的に増大している。データベースシステムは計算機の出現により始まった情報化社会の発展に伴い、銀行のオンライン処理、航空機、電車などの座席予約システム、製造業、流通業の在庫管理システム、企業内人事管理システムなど、ビジネスにおけるあらゆる側面において必要不可欠なシステムとして利用されるようになった。さらに、インターネット上を多量の情報が流通する現在、電子商取引、各種のウェブサービス、また、ウェブ上にある多種多様な大容量コンテンツ管理およびその情報解析等において、データベースシステムは必要不可欠な基盤技術と認識され、上位のアプリケーションと下位のオペレーティング・システムの間を介在するミドルウェアと呼ばれている。

人間社会のデジタル化を背景に近年のデータベース技術の進歩には目覚しいものがある。特に、関係データベースは高いデータの独立性、簡易なユーザインタフェース、強固な論理的基盤などのすぐれた特徴を有し、現在、データベーシステムの中心となっている。しかしながら、これらの利点を実現するためシステムに課せられた負荷は巨大なものになり、その高速化、高性能化が大きな課題とされてきた。実際、関係データモデルが Codd によって提案されて以来商用化に到るまでには１０年以上の歳月を要し、その後も現時点に至るまで、恒常的に性能の高速化、高機能化が求められている。しかしながら、携帯電話機を用いたメール通信の普及、ウェブ上における情報の発信、収集の状況を見るに、社会における情報のデジタル化、デジタルデバイスへの依存度は急激に高まりつつある。情報化社会の基盤となる関係データベースシステムはこのような急激な社会の変化に迅速に対応することが望まれている。

一方、最近の IC 技術の飛躍的な進歩は計算機システムの急速な進歩を推す原動力となっている。特に，関係データベース・システムの分野では大規模なデータを高速に処理することが急務となっており，この問題を解決するために様々なアーキテクチャの提案，システムの実現が行われているが大きな成功を治めているとは言い難い。技術の進歩に伴い、共有メモリ計算機、分散メモリ計算機、分散共有メモリ計算機などアーキテクチャの異なる並列計算機が提案され、現在では１０００台規模を超えるクラスタシステムが大規模データセンタ、ウェブ情報の検索エンジンなどで広く利用されている。また、ピア・ツー・ピア（Peer to Peer）システムなど、ネットワークを介した個々のパソコン、ワークステーションの計算資源を他のユーザなどにも供することで、広く分散処理を目指したシステム・アーキテクチャも実際に運用されつつある。結果、オラクルの Oracle10i、 IBM の DB2、マイクロソフトの SQL Server など広く用いられる商用関係データベースシステムも、急増するデータ量、計算機技術の進歩によるアーキテクチャの進化に伴い、常に処理の高速化、性能向上を求められ、一年ごとにシステムが改版されていくのが実情である。

以上に述べたように、関係データベースシステムは、社会基盤の不可欠な要素の一つとして、多様に変化する利用者側の高性能、高機能への要求に答えると共に、新たに現れる異なるアーキテクチャ

上への効率のよい実装が常に望まれてきた。しかしながら、関係データベースシステムの並列処理技法にはいまだ多くの課題が残っており、スケーラブルなシステム拡張を容易とする高性能、高機能並列データベースシステムの研究が急務となっている。

1.2 研究の目的と概要

本研究は、現在の情報化社会における社会基盤 (インフラストラクチャ) の一つである関係データベースシステムの性能向上を目的として、大規模データを扱う関係データベースシステムの並列処理方式を新たに提案し、アルゴリズムの開発、実装および評価を行うものである。関係データベースシステムはそれ自体が非常に大きく複雑なシステムであるため、並列処理化もシステムの一部だけでは効果はでず、また、そのアプローチも問合せ処理階層における並列化、実行時のストレージアクセスおよび入出力バッファ管理機構における並列化と様々である。本研究では、異なる並列計算機環境上での関係データベースの並列処理化について考察を行い、ストレージの入出力制御方式、データベースシステム内のバッファ管理方式、関係データベース演算処理の並列化方式、関係データベース問合せ処理方式の観点から、高性能化、高機能化技法の開発、評価を行っている。

ストレージのアクセス手法およびシステム上のバッファ管理機構は関係データベース処理性能を向上させる上で重要な要素である。汎用 OS 上で実現されている単純な入出力処理機能を用い、マルチスレッドによるアクセスを行っていては、関係データベース上の大規模データを効率良く処理することはできない。そこで、関係データベースシステム上で必要な入出力機構、フィルタリング機構を抽出し、従来の OS とは異なるモジュール分割を行うことで関係データベースシステムに適合した新たなシステム・アーキテクチャ「機能ディスクシステム」の提案を行う。「機能ディスクシステム」試作機の開発では、本研究で提案した関係データベースシステムに適合した入出力ドライバ、入出力ライブラリおよび共有メモリ管理機構ライブラリを既存の OS9 をベースとして新たに構築、試作機上に実装した。さらに、このシステム上において、 QUEL をベースとする関係データベース問合せシステムおよび「機能ディスクシステム」のハッシュ機構を利用した並列問合せ処理システムを構築し、データベースベンチマーク (Wisconsin Benchmark) の性能測定を行う。この結果、従来の関係データベースシステムと比較し、百倍程度の高速化を達成したのみならず、 Wisconsin 大学の Gamma プロジェクトの結果と比較し、計算機システム構成としては小さいながらも、システム・アーキテクチャとして必要な実装を行うことで、数十倍の性能向上が得られることを示す。

共有メモリ計算機は、近年では数百 GB におよぶ共有メモリを搭載したサーバが出現するに至っている。共有メモリ計算機では、いったん主記憶上にデータがロードされれば、並列処理は容易に実現できる。しかしながら、これらの大容量の主記憶を利用する場合、必要なデータをストレージからロードするコストは非常に重く、バッファ管理および入出力制御は単純な方式では性能向上が得られない。本研究では、ハッシュ関数を用いた並列問合せ処理の実装をめざし、ハッシュ結合演算処理の実装方式について、入出力バッファの柔軟な切り分けと共有メモリを有効利用について検討を行う。実際に、商用の共有メモリ計算機 (Sequent) 上にて、並列処理効果のハッシュ関数を用いた並列問合

せ処理を実装し、その結果から提案する方式が高い並列処理効果を得ることができることを示す。また、ストレージからのデータロードと主記憶上での演算処理を重ね合わせて処理することにより、実装した方式がストレージの入出力転送幅を十分に利用可能であることを示す。

分散メモリ計算機は共有メモリ計算機と比較し、データの爆発的増加にも容易に拡可能であり、また、高速なバスを必要としないため、コストパフォーマンスも良い。一方、それぞれのノードが個々に処理を行うため、処理の同期、負荷分散などを考慮しなくてはならない。分散メモリ計算機環境における並列結合演算を対象とした多重結合演算処理方式の最適化方式は、主記憶、ネットワーク転送の利用などを個々に取り扱ったものはあるが、システム全体における計算機資源の消費度という観点からは検討されていない。分散メモリ計算機では、あるノードで過剰な処理負荷がいったん生じると、それがボトルネックとなり、全体の処理性能が低下する。そこで、分散共有メモリ計算機における主要な資源としてネットワーク転送、 CPU 処理、入出力転送について、並列結合演算処理に係わるコストを検討する。この三つのコストを用いて、資源消費が均衡するような部分木を多重結合演算が独立したグループ単位で生成し、候補木の集合とし、それらの部分木の組み合わせを基に dynamic programming 法を用いて最小の処理コストとなる最終実行木を生成する方式を提案する。シミュレータを生成し、提案する最適化技法により生成される実行木のコストが従来の資源消費について考慮しない従来の手法と比較し、生成されるプランの品質にも、プランを探索するための空間の大きさも、十分によい結果が得られることを示す。

次に、分散メモリ計算機のシステム拡張性などの利点および共有メモリ計算機の実装の容易性を併せ持つ分散共有メモリ計算機上において、分散共有メモリのアクセス特性に合わせてデータベースシステムのアクセス局所性を考慮した並列結合演算処理方式を提案する。分散共有メモリのアクセス特性に合わせてデータベースシステムのアクセス局所性を考慮した並列結合演算処理方式を提案し、商用分散共有メモリ計算機上に実装し、キャッシュすることにより生じるデータ参照局所性を利用することにより、一般にデータ局所性がほとんどないと言われる大規模意思決定支援システム上での問合せ処理などに関し、提案する方式が有効であることを示す。さらに、分散共有メモリ計算機のキャッシュコヒーレンシ機構に関するメモリアクセスコスト等について、実機上において詳細なデータをとり、これに基づき、シミュレーションを作成し、提案する方式が、キャッシュサイズが相対的に小さくなるようなノード台数が大幅に拡張された場合にも有効であることを確認する。また、あらかじめキャッシュ上の参照される頻度の高いデータを複製することで、データのアクセスに偏りが生じた場合にも、負荷分散が可能であることを示す。

本研究では並列関係データベース演算として、最も処理負荷の高い結合演算を対象として議論を進めている。結合演算の並列化アルゴリズムとしては、 Grace ハッシュアルゴリズムを代表とするハッシュに基づく結合演算処理の性能がよい。しかしながら、電話帳の名前の分布などでも知られているように実世界のデータは偏っており、データの偏りによっては、均等に分割できる適当なハッシュ関数がない場合も多い。そこで、ハッシュ関数を用いた並列処理技法に従来のネストループ処理方式を組み合わせ、 GN Hash 方式を提案する。一般にネストループ方式は、内側のリレーションを複数回読み出すため処理コストは高いが、その処理コストはデータの偏りには依存しない。一方、ハッ

シュ結合演算では、ハッシュ関数適用後のそれぞれのクラスタのデータ分布が一様であるという保証はなく、データの偏りが高い場合にはクラスタの再分割を繰り返し行う必要がある。また、データ分布が一様であったとしても、外側のリレーションが主記憶の数倍程度の大きさであれば、ネストループ方式の処理コストがハッシュ結合演算の処理コストよりも小さくなる。そこで、データの偏りによりクラスタの再分割が起こった場合には、ネストループ方式をそのクラスタに適応することで、繰返されるクラスタの再分割を防ぐことで、負荷の偏りにもロバストとなる。ハッシュ結合演算処理およびネストループ処理の詳細なコスト式を導出し、シミュレーションによる詳細な解析を行う。その結果、他の結合演算処理方式と比較し、 GN ハッシュ方式がデータの偏りが非常に高い場合にも性能劣化が少ないことを確認する。

1.3 本論文の構成

本論文は以下の章からなっている。まず、序論として本章があり、続いて第２章にて従来の並列関係データベース処理の研究について、システム構築の観点および並列計算機のアーキテクチャの観点から簡単に概観する。第３章では、関係データベース処理機能をディスク部分に取り込んだ機能ディスクシステムの研究およびその性能評価について述べ、第４章では共有メモリ計算機における並列ハッシュ結合演算処理方式の研究について述べる。第５章では、分散メモリ計算機における並列多重結合演算処理の最適化技法の研究について述べる。第６章では、分散共有メモリ計算機における関係データベース処理に適合したバッファ管理の研究について、ハッシュ結合演算処理を基に述べる。第７章は GN ハッシュ結合演算処理として、並列処理環境に適合したアルゴリズムをデータのサイズに合わせ動的に選択する処理方式の研究について述べる。第８章にて、本研究で得られた成果と今後に残された課題についてまとめる。

第 2 章

並列関係データベース処理と従来の研究

2.1 概観

近年、関係データベースシステムが商用データベースの中心となり、パソコンからワークステーション、メインフレームまで、広く実装されるようになっている。しかし、年々増大するデータの容量に対し、いっそうの性能向上が要求されており、現在の関係データベースシステムでは並列処理効果によるシステムの性能向上が研究の中心といえる。実際、商用並列計算機 Sequent の上で実装されている Oracle など、すでに商用化されている関係データベースシステムもある。しかし、 Oracle の実装方式では、 Inter-Query の並列化として複数の問合せあるいは、データベースが提供しているいくつかのプロセスの並列処理などが、各プロセッサにより、行なわれているが、その手法は単純に機能別にプロセッサにプロセスを割り振ったに留まり、十分な並列処理効果を得ているとは言い難い。[30] で述べられているように、並列関係データベースシステムにおいては、選択する並列アーキテクチャにより、異なるシステム資源の組み合わせを考慮した実装を行なわねばならない。ここでは、並列関係データベースシステムの問合せ処理方式の環境として共有メモリ環境 (Shared-Everything (SE) Environment)、共有ディスク環境 (Shared-Disk （SD） Environment)、分散メモリ Shared-Nothing (SN) Environment) の３つの異なる並列アーキテクチャ上でいかなる並列処理方式が提案されているかについて簡単にまとめる。さらに、並列関係データベースにおける並列処理化を、演算処理のレベルから以下の三レベル、すなわち、 (1) 単一の演算内での並列処理化 (Intra-Operation Parallelism)、複数の同一演算間での並列処理化 (Inter-Operation Parallelism)、問合せ内、問合せ間の並列処理化 (Intra-Query Parallelism , Inter-Query Parallelism) に分類し、並列処理における問合せ最適化、負荷分散のための手法について簡単まとめる。また、実装されている並列データベースシステムの事例を挙げ、具体的なシステムの上で並列処理方式について、まとめる。

2.2 並列関係データベースシステム

並列関係データベースシステムの提案は並列計算機アーキテクチャの研究とともに始まり、すでに２０年以上経過している。 1970 年代、 Slotonik の Logic par Head から始まり Gamma, Bubba, Teradata, Super Database Computer (SDC) まで、関係データベースシステム専用アーキテクチャだけでもさまざまなものが提案されている。しかし、現状のテクノロジーの発展とそれに起因する高性能、安価なワークステーション、高速なメモリ、高速ネットワーク、ディスクの低価格化、小型化など、またそれを背景に発表される商用並列計算機の登場を理由に、並列関係データベースシステムのアーキテクチャも専用高機能ハードウエアを関係データベースに特化して構築するよりは、既存の並列計算機の上での実装、あるいは特殊なハードウエアを用いないアーキテクチャ構成に移行しつつあると言える。

この章では、 [30, 118, 34] を参考に関係データベースシステムのアーキテクチャ的側面からの特性について、検討を行なう。並列処理システムでは、理想的にはそのシステム資源を増やした場合に、負荷の高い単一処理の処理時間のリニアな速度向上（いわゆるスケーラビリティ）、または処理要求が増大した場合にもその応答時間が一定である（スループットの増大）ことが、期待される。近

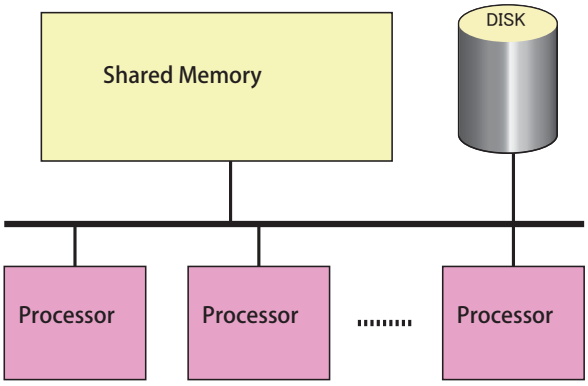

図 2.1: 共有メモリ環境

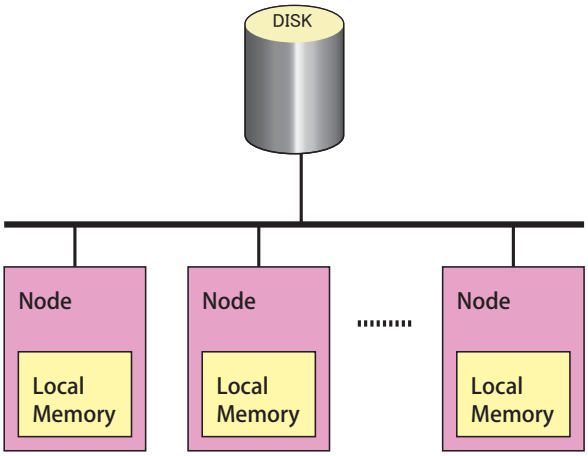

図 2.2: 共有ディスク環境

年の並列データベースシステムはその規模の拡大のしやすさ（スケーラビリティ）及びネットワーク技術の発達などにより分散メモリ環境を中心に研究が行なわれているが、システムの実装の容易さなどを考慮すると、商用並列データベースなどは共有メモリ環境を中心として実装されている。以下、三つの並列環境における並列データベースシステムの実装方式について考察する。

2.2.1 共有メモリ環境

共有メモリ環境にけるシステム構成を図 2.1に示す。このシステム構成では高速なバスにより全ての資源が密結合されており、メモリ、ディスクなどの資源をどのプロセッサからもアクセス可能である。しかし、バス幅などの技術的な限界から、並列性を十分に期待できる並列度はたかだかプロセッサ数十台程度と考えられる。つまり、 SE アーキテクチャではバス幅を越えて資源を増やしても、それぞれのプロセッサからのアクセス要求がぶつかってしまい、性能は劣化することになる。また、現在使用されている高速なマイクロプロセッサの性能に追随できる高速バス自体、高価なものとなってしまい、コストパフォーマンスの面からは以下で述べる分散メモリ環境の基に構築されたデータベースシステムの方が良いと考えらる。

これらのアーキテクチャ的観点からはいくつかのマイナス面があげられる反面、メモリがいずれのプロセッサからもアクセスすることができるため、プロセッサ間の同期、通信を実現することはたやすく、また処理負荷のバランスをとることも容易である。さらに、基本的にはメモリ、ディスクともに全てトランスペアレントに見えているため、従来の single CPU 上で開発された逐次アルゴリズムを容易に並列化することが可能である。

一方、データベース処理においては、大量のデータを主記憶バッファにロードし、複数のプロセッサから頻繁にアクセスするため、プロセッサ、メモリ、ディスク間のバスの転送幅がネックとなる。

2.2.2 共有ディスク環境

共有ディスク環境におけるシステム構成を図 2.2に示す。こシステム構成では、各プロセッサごとにメモリは分散しているが、ディスクへのアクセスは専用 I/O バスを通して、全てのプロセッサがアクセスすることが可能である。ここでは、共有されているがのディスクであるため、 SSE アーキテクチャほどはバスの高速性は望まれないが、データ転送幅自体はやはり、大きいことが望ましい。また、並列度という点では、メモリが分散しているため、プロセッサを増やした場合に性能向上が期待できるが、やはり入出力性能に関しては SE アーキテクチャと同じ問題を抱えており、並列度はたかだか数十台程度と考えられる。

プロセッサ間通信、同期、メモリ間の整合性などを考慮すると SN アーキテクチャと同じ問題が出てくるため、並列アルゴリズムの実装に関しては、容易ではない。しかし、処理負荷のバランスについては、最も問題となる入出力アクセスが共有されているため、 SN アーキテクチャと比較すると簡単に解決できると考えられる。

一方、各ノードからディスクへの書き込みの同期、複数のノードからの大量の入出力アクセスなどに関し、入出力処理インテンシブなデータベース処理では、ディスクのデータ転送速度が処理のボトルネックとなると考えられる。

2.2.3 分散メモリ環境

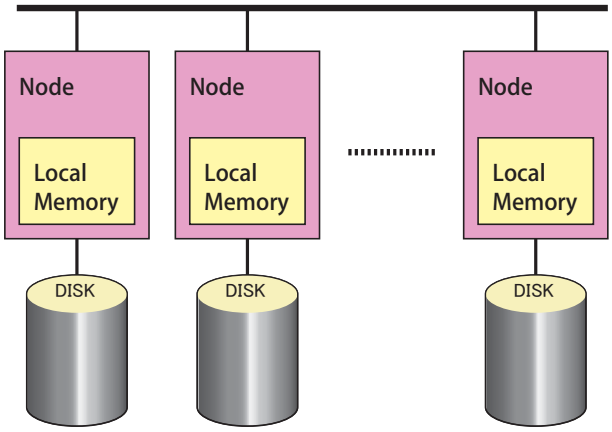

図 2.3: 分散メモリ環境

分散メモリ環境におけるシステム構成を図 2.3に示す。このシステム構成では、ネットワークを通じて複数のプロセッサモジュールが結合されており、各プロセッサモジュールごとにメモリ、ディスクなどの資源が分散している。従って、現在のネットワーク実装技術を考慮すると、この形態が最もスケーラビリティがあるといえる。一方、それぞれのプロセッサモジュール毎で並列に処理を行なう場合、プロセッサ間の同期、メモリ内容の整合性、通信、どれをとってもネットワークを介して行なうためのオーバヘッドが全体の性能に影響を与えるまたプロセッサモジュール間のデータ転送もネットワークを通じて行なうため、ネットワーク性能が全体の性能にとって重要な要素となる。

近年は大規模データセンター等も数百台から千台規模のブレードサーバに大容量ディスクアレイを接続した構成が多く採用され、オラクルの Oracle10i、 IBM の DB 2 など多くの商用システムが分散メモリ環境に対応している。

分散共有メモリ環境では、すべての資源が分散しているため従来からの逐次アルゴリズムを効率よく並列化するのは容易ではない。また、負荷のバランスに関してもそれぞれのプロセッサごとに独立して処理を行なっているため、局所的に負荷の高いモジュールがあった場合、負荷の平坦化を行なうは難しく、新たな並列アルゴリズムを考慮する必要がある。

2.3 関係データベースシステムにおける並列化

2.3.1 並列処理形態

関係データベースでは、データの一様性とオペレーションの一様性から容易に並列処理化を行なうことができる。データベースの並列処理形態は、 [30] によると、従来の並列処理と同じく、データの流れからパイプライン処理とパーティション並列処理の二つに分類される。図 2.4にこの二つの処理方式のデータフローを示す。例えば、パイプライン処理では、複数の結合演算処理を行う場合に、まず外側のリレーションをディスクから読み出し (図中で Scan)、その結果を主記憶上でソートしている間に、並列に次のソースリレーションの読み出しを行う。一方、パーティション並列処理では、あらかじめ複数のディスク等に分散配置されたリレーションを並列に読み出し、それぞれのノードの主記憶上でソート処理を施し、主ノード上にて並列にソートされた結果をマージする。

2.3.2 問合せ処理の並列化

関係データベース問合せ処理は、通常、それぞれの問合せに対してそれが発せられた順に、アクセス要求の整合性を確認したあと、複数の問合せが一般には並行に処理される。ここでは、これらのトランザクション処理とその並行処理、ロック機構に関しては触れない。以下では個々の問合せ処理の並列化について検討を行なう。

単一の問合せ処理は、まず、複数の関係データベース演算にパージング、それらの演算の処理順を問合せが効率的に処理できるように、スケジューリングの最適化がなされ、実行順序を定めたスケジューリング木が構成される。実行用のスケジューリング木では、それぞれのリーフは個々の選択演算や、射影演算、ソート処理などを表し、それらのリーフが交わるノードが結合演算を現す。決定されたスケジューリングに従って、各々のリーフから Bottom-Up に処理が行なわれる。従って、データベース問合せ処理システムの中での並列化は、その処理レベル毎に演算（選択、結合、射影などの個々の演算）内並列処理（Intra-Operation Parallelism)、演算間並列処理（Inter-Operation Parallelism）、問合せ間並列処理（Inter-Query Parallelism）の 3 レベルが考えられる。

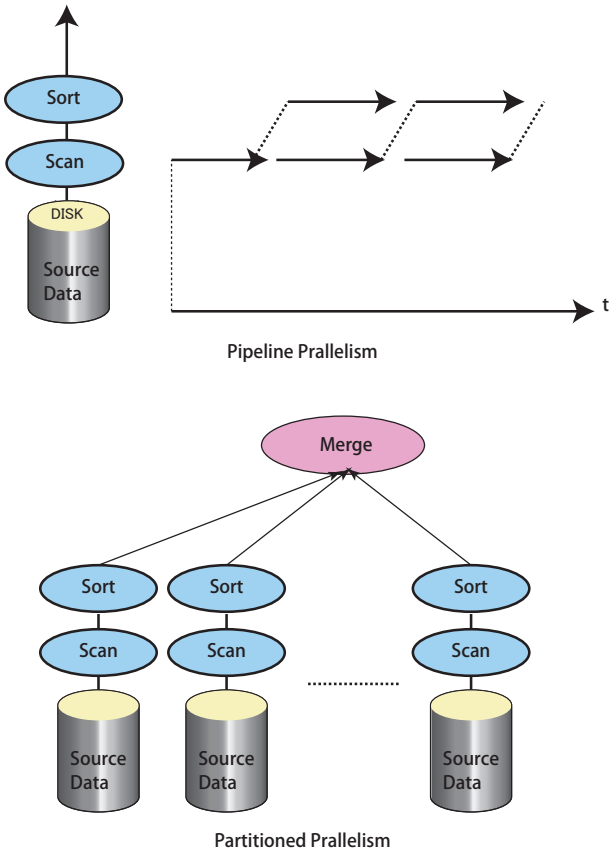

図 2.4: 並列処理の形態

2.4 単一関係データベース演算内における並列処理

関係データベースのオペレーション内並列処理化の研究は、主として演算の中でも負荷が重い演算、結合演算、ソート処理を中心として行なわれてきた。選択演算などの対象オペランドが一つしかない演算では、各プロセッサ間での同期が簡単なため、並列化もスムーズに行なわれる。しかし、ソート処理では、例えば個々のプロセッサで並列にソートして得られた結果を更にマージしなければならない。そのため、個々のソート列を再度走査するマージ処理がボトルネックとなり、単純な並列処理化では性能を向上することは難しい。また、結合演算では二つのオペランドをマッチングする必要があり、並列化自体は簡単にできるが、大幅な性能向上を目指すためには、メモリの有効利用と不要な入出力アクセスの削減、さらには、並列化ができず性能の劣化の原因となる部分を減らすなどの工夫が必要である。

結合演算の並列処理化については、専用データベースマシンの研究の進展と共に、ネストループ方式、ソートマージ方式、ハッシュに基づく処理方式等、いくつかの手法が提案され、それぞれの並列化が考案された。

ここでは、近年その性能の高さから並列処理化の研究の中心となっているハッシュに基づく結合

演算方式の並列化について述べる。ハッシュに基づく結合演算方式は [61, 12, 25] などで提案された当初から並列データベース環境での議論がなされており、その並列化については共有メモリ環境から分散メモリ環境まで、前述の 3 つの並列環境上での処理方式が考案され、ハッシュに基づく結合方式の優位性が示されている。特に、分散メモリ環境上での並列化についてはメモリの利用方法、データ (リレーション) のノードへの分散方法などについては多く検討され、さらにデータの偏りがある場合についての検討が行われている。

ここでは、[96] で述べられているハッシュに基づく並列結合演算の処理方式を簡単に説明すると共に、以下の節における結合演算処理方式の説明で用いられる言葉について説明する。

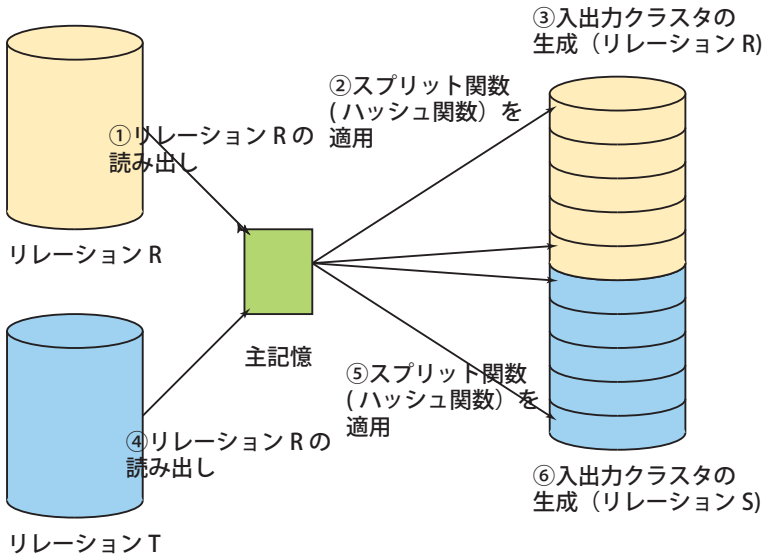

図 2.5: ハッシュ結合演算並列処理の流れ ースプリット処理ー

二つのリレーション R、S に対し、結合属性 joinkey に関する等結合演算を想定する。対応する SQL 文を以下に示す。

```
SELECT * FROM R,S WHERE R.joinkey = S.joinkey
          AND R.a1 < Constant1 AND S.b1 > Constant2....
```

where 節第一条件は二つのリレーションの結合を表し、第 2 条件以降は R,S 各々に対する選択条件とする。主記憶上でタプルの突き合わせのためにハッシュ分割されるクラスタをバケット、分割に用いられる関数をハッシュ関数と呼び、二次記憶に書き戻されるクラスタを入出力クラスタ、分割に用いられる関数をスプリット関数と呼ぶ。

・スプリット処理 (図 2.5参照)

リレーション R をディスクから読み出し、適当なスプリット関数を選び、それぞれのクラスタが主記憶に格納できるような大きさになるよう分割し、クラスタ毎にディスクに書き戻し、入出力クラスタを生成する。リレーション S も同じスプリット関数を用いて分割し、ディスクに書き戻す。

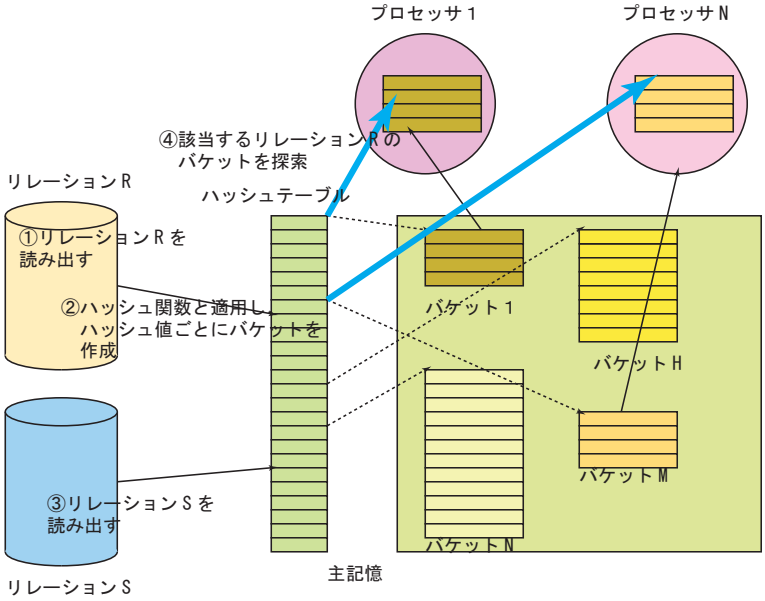

図 2.6: ハッシュ結合演算並列処理の流れ －ハッシュ処理とプローブ処理－

・ハッシュ処理 (図 2.6参照)

主記憶にロード可能なリレーション R あるいはスプリット処理を行った後の入出力クラスタをディスクから読み込み (１)、結合演算におけるタプル突き合わせ処理を行うために、主記憶上に、リレーション R の結合属性にハッシュ関数を適応し (２)、ハッシュテーブルを生成する

・プローブ処理 (図 2.6参照)

主記憶上に、リレーション S を読みだし (３)、その結合属性にハッシュ関数を適応し、当該ハッシュテーブルを走査し (４)、実際に結合演算におけるタプル突き合わせ処理を行う。

2.5 多重結合演算の並列処理とそのスケジューリング最適化

並列処理技法の目標は、複雑は問い合わせに対してその並列効果（スケーラビリティ）をどこまで得られるように並列化を行なえるかであり、これは実装するアーキテクチャの種類と処理中のデータの分布などの動的な負荷の変動により、静的なシステム資源とデータベース情報から得られる最適化のみでは十分な並列性を引き出すことは難しいと考えられる。

ここでは、複雑な問い合わせの中でも、最近、並列化とその最適化の研究成果が多く報告されているマルチジョインをとりあげる。まず、単一 CPU におけるマルチジョインの最適化として、 System R のグループによって提案された方式について簡単に説明を行なう。従来、マルチジョインに関してはこのリニアツリーの構成法以降、単一 CPU の環境においては、静的ななデータベース情報を用いて選択率などを推定して得られるコスト式の提案、及び結合演算を行なう際のリレーションの順序などについての提案はあったものの、本質的には [99] で提案されているものと同じであった。しか

し、 Wisconsin 大学 Gamma グループによる分散メモリ環境における複雑な問合せの処理として、多重結合演算をそれぞれ Left-Deep 木, Right-Deep 木, Bushy 木の形式に従って処理した場合の性能比較（具体的な実装はは LD と RD のみ）の報告 [98] がなされて以来、並列環境におけるマルチジョインの研究が活発に行なわれるようになった。本節では、多重結合演算のスケジューリングを木の形状ごとに簡単に説明し、前節の三つの並列処理環境毎に具体的にどのような手法が提案されているか述べる。

2.5.1 多重結合演算のスケジューリング

多重結合演算の処理では、複数の結合演算を行なう順を示すために木構造を用いている。木構造のリーフは、ソースリレーションを表し、各ノードはその子供のノードからの結果またはリーフで示されるリレーションとの結合演算を表す。並列ハッシュ結合演算の多重処理を行なう場合、効率よくメモリを使い、ディスクのアクセスを少なくすることがスケジューリング木を生成する目標となる。[98] では、生成されるスケジューリング木の枝の形態により Left-Deep 木, Right-Deep 木と Bushy 木の 3 タイプに分けている。

(1) Left-Deep 木

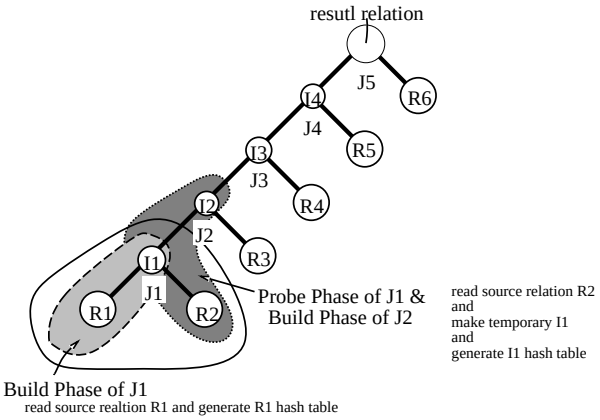

図 2.7: Left-Deep 木

Left-Deep（LD）木は、枝分かれがなく左に傾いた形態をしており、単一の結合演算処理を順次最下層のリーフから行なっていく。[99] で提案された方式（daynamic programming 方式と呼ばれる）では、この形の木が生成される。図 2.7にリレーションが 6 個あった場合の LD 木を示す。この場合、最も下の結合演算 J1 から処理が行なわれる。まず最初の結合演算 J1 を実行するために、 R1 をディスクから読み出し、ビルドフェーズが行なわれる。次に、プローブフェーズとして R2 がディスクから読み出され、生成された R1 ハッシュテーブルを検索して突き合わせ処理を行なう。このとき、生成される中間結果 I1 は, メモリに空きがあれば次の結合演算 J2 のビルドフェーズとして各プロセッサにスプリット関数を適応した後、転送する。メモリに空きがない場合には、いったん生成さ

れたプロセッサのディスクに書き戻される。以上の処理が、順次結合演算 J2,J3,J4,J5 と繰り返されて、演算結果を得る。

LD は主記憶上にその時点で行なわれている結合演算のハッシュテーブルがロードされ、さらに主記憶に空きがあれば次の結合演算のためにプローブ後の中間結果がハッシュテーブルとしてロードされる。従って、中間結果が小さい場合には主記憶に空きがでる。一方、最下層の結合演算（図 2.7では J1）を除いて中間結果を次の結合演算のビルドフェーズで用いるため、中間結果のサイズを正しく予測できない場合には不要な入出力が増大する。また、結合演算間の並列処理はプローブフェーズと次のビルドフェーズがオーバラップして処理されるだけである。

(2) Right-Deep 木

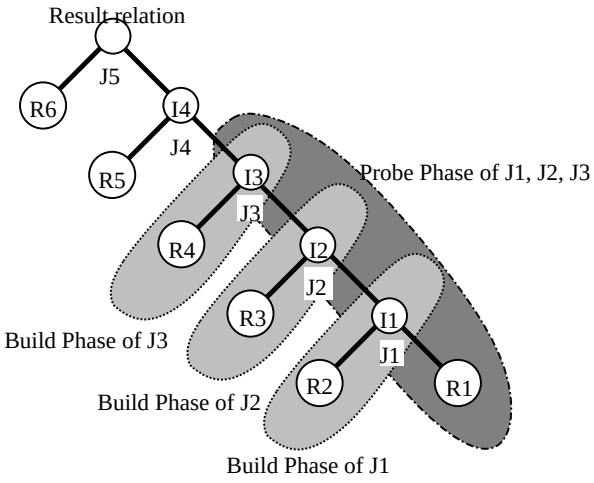

図 2.8: Right-Deep 木

Right-Deep（RD）木も Ledt-Deep 木と同じく枝分かれのない形態であるが、傾きが異なり右である。ハッシュ結合演算では、左のリーフあるいはノードの結果をハッシュテーブル生成のためのビルドフェーズで読み出し、右のリーフあるいはノードの結果をプローブ処理に用いるため、左右非対象な処理となっており、 RD 木と LD 木では処理の流れが異なる。図 2.8にリレーションが 6 個あった場合の RD 木を示す。 RD 木でも最下層の結合演算から処理されるが、まず、左リーフのソースリレーションのサイズからメモリに入るだけの結合演算を並列に処理することを決定する。図 2.8では $(R2 + R3 + R4) < M$ とすると、連続した J1, J2, J3 の結合演算のビルドフェーズをまず実行する。この 3 つのビルドフェーズが終了すると、 J1 のプローブフェーズが始まり、順次 J2,J3 のプローブフェーズがパイプライン処理される。つまり、 J1 で生成される処理結果は LD と異なり J2 のプローブフェーズで使用されるため、 J1 のプローブフェーズが終了するのを待つ必要はなく、生成された時点で次のプローブフェーズ処理のために該当プロセッサに転送される。 J3 のプローブ処理後は一旦中間結果をディスクに書き戻し（あるいは余裕があれば主記憶にロードしておいてもよい）、つぎの結合演算群の処理を行なう。

このように、 RD では主記憶にハッシュテーブルとして展開可能なソースリレーションをもつ結合演算を複数並列処理することとなる。従って、 LD と比較して結合演算間の並列性を引き出しているといえる。また、主記憶にロードするのは、ソースリレーションであるため、その大きさが静的にわかっており、 LD よりも主記憶の利用効率を上げることが可能である。一方、主記憶をほとんどハッシュテーブルとして利用するため、中間結果はディスクに書き戻さねばならない。 [98] では実際に多重結合演算処理を並列計算機上に実装し、複数のリレーションがメモリに収まる場合には LD よりも RD の性能がよいという実験結果を示している。

(3) Bushy 木

前述の二つの木と異なり、 Bushy 木は図 2.9に示すようにいくつかの枝から構成されている。従って、どの分枝からいかなる順序で処理を行なうかにより、その性能は大きくことなる。

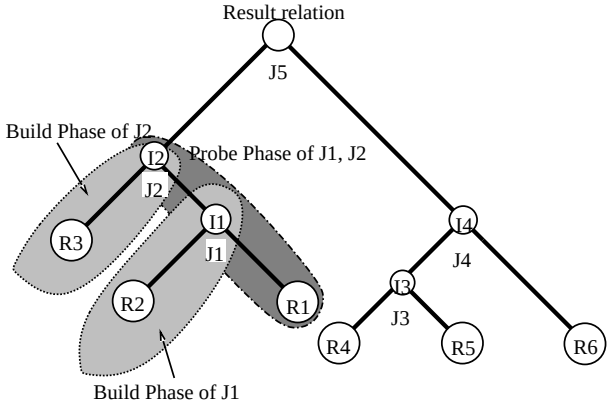

図 2.9: Bushy 木

図 2.9では、左側の枝の J2,J1 の結合演算の処理を RD と同様に R2,R3 のハッシュテーブルを生成した後、二つのプローブフェーズをパイプライン処理する。同時に J2 の結果とは全く依存してないため、右側の枝の J3 を J1,J2 の結合演算と並列に処理することができる。このように、 RD では一つの枝上で連続した結合演算間の並列性に着目して処理がおこなわれていたが、 Bushy ではまったく別の枝に連なる結合演算群を並列に実行することが可能であり、並列性を引き出す上では最も自由度の高い形態をしている。

しかし、 [98] で指摘されているようにことなるソースリレーションへの同時アクセスはディスクからのリレーションの読み出し時間のオーバヘッドとなり、並列に処理できる結合演算の数は増えるが、処理性能は向上しない。また、それぞれ独立して処理を行なうことにより、結合演算群の間での同期をとるためのオーバヘッドが必要となってしまう。

2.5.2 単一 CPU における多重結合演算スケジューリング

まず、従来の単一 CPU の環境でのマルチジョインの最適化については、すでに SYSTEM R の研究において、[99] がリニアツリーの生成方式（Left-Deep 木の形式）について、報告している。ここでは、リレーションサイズ、インデクスなどのデータベースが保持している静的情報を元に、コスト式を用いて最小のコストで処理できるようなツリー構成の探索方法を示している。この方式は基本的に可能性のあるパスを全て探索するものであり、3 ウェイ程度の結合演算であればその探索領域は小さいが、探索パスは N! 通りあるため、単純な方法では負荷が思い。従って、結合演算においては、探索領域を減らすための簡単な heuristics も提案されている。これも、結合属性が同じ場合の結合演算が複数あれば、それぞれ毎に結合順序を置換してパスを探索するだけのものであり、与えられた結合演算によっては、探索パスは減らないこともある。

探索パスを減らすための、 heuristic な方法は結合属性について着目することで、 T1 * T2 * T3 と 3 つのリレーションの結合演算をおこなう場合、 T1 * T2 と T2 * T3 の結合属性が異なる場合、 T1 * T3 * T2 と T3 * T1 * T2 等のパスについては、コスト比較をする場合の探索パスから除いておくというものである。これら、結合属性を含まない結合結果は、全カーディナリティの積となるため、問合せの処理コストが高くなるからである。

single CPU の環境においては、ここで提案されたコスト計算と同様のことが行われ、それぞれのオペレーションを順次処理するようにスケジューリングが行なわれてきた。しかしながら、並列処理においては実装する環境及び、これから述べるように、複数の計算機資源、特にプロセッサ処理と入出力処理などをオーバラップする可能であることを考慮すると、最適逐次処理プランを単純に並列化するだけでは最適スケジューリングとはいえない場合も多い。

2.5.3 共有メモリ環境におけるスケジューリング

(1) シンガポール大学の Bushy 木スケジューリング

Singapore 大学の Lu ら [75] は、 Bushy 木を用いた場合の多重結合演算スケジューリングの最適化方式、 Greedy Parallel Multiway Join Method (GP) を提案している。ここでは、 Bushy 木の形状をベースとし、各段ごとの処理では、必ずそのステップに属する全ての結合演算処理を同期することで、必ず処理が完結していることとし、 RD 木などのパイプライン処理による複数結合演算にまたがる非同期なものは考えない。提案された最適化方式はグリーディに候補を探索する。

この方式では、各ステップごとに同時に処理を行なう結合演算の数とその時の最後まで処理を行った場合の最小コストを計算しながら、各ステップごとの結合演算の組合せを求める。従って、全ての場合について計算しようとすると、計算量が非常に大きくなるため、計算量を減らすための経験則として、以下の二つの方式が提案されている。

- GP based on Total cost (GP:T)
  図 2.10にあるステップのコスト比較図を示す。求めるステップ毎に、並列に処理を行なうことのできるリレーションのペアが選ばれると、それぞれの次の段のコスト計算を行なう。図中で

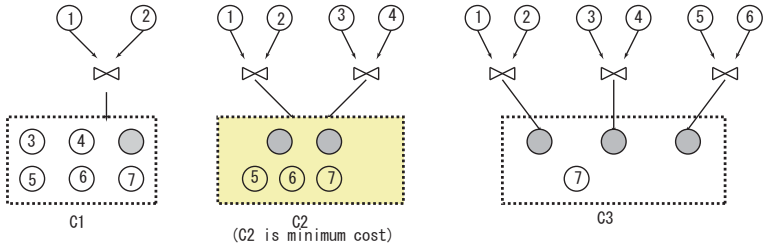

図 2.10: Greedy Parallel Multiway Join Method

は７つのリレーションから二つのリレーションが選ばれると、次段のリレーション候補は、選ばれた R1,R2 のリレーションの中間結果とそれ以外のリレーションとなる。ソースリレーションが７つある場合、結合演算するペアは最大三組まで選ぶこととなる。それぞれの組合せごとに、次段の結合コストを計算し、最小のコストなるものを次のステップのベースリレーションとして選ぶ。図では、二つの結合演算を一段目で選んだ場合のコストを最小としている。結果、二つの中間結果リレーションと三つのソースリレーションの組合せから次段の処理が行われる。ステップごとに、それぞれの組のコスト計算を行なう必要があるため、並列に処理するペアの組合せごとの全てのコスト計算を求めねばならないので、次に示す GP:P と比較し、計算量が多くなる。

- GP based on Partial cost (GP:P)
  この手法では、まず結合演算をする必要のあるリレーションペアの全数に対し、 minimum なものを求め、その結果と残りのリレーションを結合したときのコストを加えたものが、一つペアを増やした時の minimum コスト式を比較し、より小さなものを選ぶ。

また、これらの heuristics の中で使われている relation-pair の生成には以下のような方式が使われている。まず、なるべく共通の結合属性を持っているペアの中から何らかの heuristics(ここでは、４つ提案されている）で中間結果が最小になりそうなものを順に選択し、最後にこの heuristics で選ばれなかったものをペアにして生成する。

二つの GP:T と GP:P の性能比較がコスト式を用いたシミュレーションによって行われ、計算量が多いだけ、 GP:T の性能が良い。リレーション数が８つまでは、 optimize なスケジューリングと比較しても９０パーセント程度の問い合わせが童貞度の性能で処理できると報告されている。一方、GP:P が GP:T の性能を上回るのは与えられた問い合わせのほぼ１パーセント程度であり、５０パーセント程度が GP:T と同じ程度の性能がでるスケジューリングを生成できると報告している。

ここでは、 Bushy 木は同期がとれるようなバランスしたもののみを並列化の対象としており、オペレーション間処理における並列度は上がるものの、 Schneider らが提案している RD によるパイプライン効果が得られない。さらに、 RD 処理と比較して、同期をとるためのオーバヘッドなどを考慮に入れた場合の比較がなされていない。また、実際に並列効果を最大限に活かすためのプロセッサス

ケジューリングに関してもほとんど、触れられていない。中間結果はいったん書き戻すとしているため、メモリ利用効率についても検討されていないといえる。

(2) Zig-Zag 木 (DBS 3における多重結合演算処理方式)

INRIA で開発されている並列データベースシステム（DBS3）で提案されている多重結合演算処理方式 [130, 131, 4] である。最終的にできあがるスケジューリング木の形状から、 Zig-Zag 木と呼ばれ、シミュレーティドアニーリング手法によって最終解が求められる。 DBS3 は共有メモリ型計算機上に構築された並列データベースシステムであり、並列化は比較的容易である。

基本的には RD を適当な部分で切り離すことで、いくつかのセグメントに分割して、その切り離されたセグメントごとにハッシュテーブルの生成に用いるものであり、基本的発想は segmented RD と変わらない。しかし、この方式では中間結果をメモリにおいているため、 segmented RD よりも全体のコストを下げることが可能である。しかし、 tree の変形そのものは、そのセグメントの中間結果を常に次のセグメントの最初のハッシュテーブルとして用いらるため、 segmented RD より制限が加えられている。

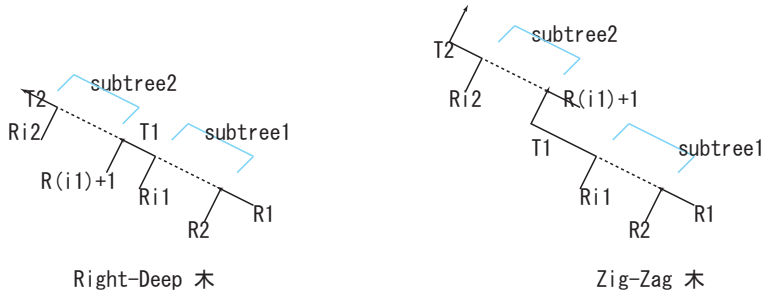

図 2.11: Zig-Zag 木の形状

図 2.11に Sliced RD 木と Zig-Zag 木の形式を示す。左側に示したものは、 RD 木を主記憶に入る大きさでそのまま切り離し、セグメント化している。これらのセグメントでは、最後に出来た中間結果をディスクに書き戻し、次段の右ソースリレーションとして利用する。これに対し、 Zig-Zag 木では、ディスクに中間結果を書き戻さず、次セグメントの左ソースリレーションとして用い、ハッシュテーブルを主記憶上に作成する。 Zig-Zag 木は、メモリに中間結果をなるべくのせることで、不要な入出力コストを下げることが出来、その性能はよいが、他の論文と比べると実際のコスト式などが不明瞭で、中間結果がわかっている場合には確かに有効に実装できるが、データの偏りなどを考える上では、中間結果が最も小さくなるような場合を選んでセグメントを切らないと、その性能はあがらないと思われる。また、このようなメモリの有効利用を最大限考慮するような木の生成方法についてはほとんど述べられていない。また、この論文では Optimal RD をセグメントに分割するとしか記述しておらず、 Optimal sequential plan を考えずに木の生成を行なう segmented RD とメモリを組み合わせた方が性能が良くなる可能性は高い。

[131] では、 Sliced RD 木 (Scheider らの Static RD 木) と Zig-Zag 木の性能比較をパラメタを変えて行い、 Zig-Zag 木の有効性を示している。上述の Zig-Zag 木の方式からわかる通り、メモリが一つのソースリレーションと中間結果のハッシュテーブルが展開できるだけの容量があれば、ディスクの入出力コストが経るため、 Zig-Zag 木の方が性能が良くなる。

さらに、新しい報告 [71] では query plan とその最適化のために検索する空間の大きさ、およびそれを減らすための手法について検討している。つまり、最適プランを得るためには、基本的に非常に大きな空間を検索しなくてはならないため最適化コストが大きなものになってしまう。一方、検索する空間を何らかの heuristics などを導入することである程度小さくすれば、最適化コストも小さくなるが、得られるプランが最適なものになるかどうかは、保証されない。ここでは、 Toured Simulated Annealing 方式を提案することで、検索空間を狭めることで最適化コストを小さくしながら、ほぼ最適プランを得る方式を提案している。

まず、従来の方式における検索空間は、 [71] によれば、これから、 LD 木、 RD 木などのリニアな木の形状に対し、 bushy tree の検索空間はリレーションが１０個を越えると極端に大きくなると報告している。このスケジューリング木の候補空間から最適な木を探索する方式として、基本的に一つの tree にソースリレーションまたは中間結果を合わせて次第に成長させていく方式と、例えば逐次最適化方式から得られる一つの最適解候補の木から順次、リレーションの順番、セグメントの切り型を改良していく転移方式の二つの対象的な方式が提案されている。

また、 DBS3 では、検索方式として、以下の５つの方式をシミュレートしている。

- Greedy
  これは、最も検索空間が小さくてすむ。 tree 探索空間の最も深いところから順に expand していく方式であり、探索レベルごとに不要な空間を捨てていくため、探索中に保持しなくてはならない空間は非常に小さい。 DBS3 では、最初に最もサイズの小さいリレーションを選び、あとはコストが小さくなるようにリレーションを選んでいく。

- Dynamic Programming 方式
  通常の逐次環境における最適化方式とほぼ同じである。この方式では、横並びの探索を行なうタイプのものであり、コストを小さくする可能性のある全ての木の空間を検索するため、ほんのわずかなリレーションで非常に大きな探索空間となってしまい、現実的ではない。

- randomized
  この方式では、最初に Greedy 方式で候補の木を生成し、それを基にランダムに転移 (リレーションの移動、リニアな木の分割、部分木の入れ替え) を行ない、最小のコストを得られるように木の変形を行なう。

- Interactive 方式この方式では、最初に Greedy 方式で候補の木を生成し、それを基に下から順番にリレーションの入れ換えを行い、最小コストが得られるように木の変形を行う。

- Simuulated Annealing 方式シード木のリレーションの入れ換え手法として、 Simulated Aneealing 手法を応用している。計算時間は Interactive 方式や randomized と比較すると大きいが最適な解が得られる可能性が高い。

(3) 最適化コスト式の検討 (Michigan 大学)

ミシガン大学の Silvastava ら [107, 88] は、従来から提案されている問合せプラン生成方式では、まだそのコスト式が不明確であり、また並列処理環境で最適化を行なう場合にその検索範囲が逐次処理の時に比較して非常に大きくなるが、その検索範囲と得られる問合せプランの性能とのトレードオフが明確になっていないと指摘し、多重結合演算を取り上げて、並列処理を導入した場合のコスト式と問合せプラン木の生成方式について提案、検討を行なっている。

また、最適化問い合わせ処理プランを得るために、 Exhaustive な方式と heuristics を採り入れた方式を提案している。以下に二つの方式の処理の流れを示す。また、シミュレーションにより、 heuristics を用いた方式でもほぼ全件検索を行なう場合と同じ性能が得られるという結果を得ている。

前提としている環境が SE であるため、コスト式自体は単純なものであり、このまま、分散メモリ環境には適用できない。また、 heuristics における木の生成自体に具体的な例がついておらず、実際木のノードとソースリレーションをどのような構成でつないでいくのか、 Lu らの結果と比較されておらず、今一つ不明確である。確かに、小さな検索範囲で最適なものが得られるような結果が出ているが、この方式自体の不明確さもあって、データの分布の偏りなどに対する動的な処理などを付加するのは難しいと考えられる。

2.5.4 共有ディスク環境

(1) Segmented Right-Deep 木

IBM の Young のグループ [19] では、以下に述べている [98] の研究結果を元にさらに性能を向上するための結合演算のスケジューリングとして、 Right-Deep 木をいくつかの並列処理単位（Segment) とした場合に、 optimal な RD よりも性能向上が可能な Segmented Right-Deep 木の提案をしている。

Segmented RD の基本方針は、メモリのサイズから、 RD をいくつの segment に分割するかを決定し、各 segment 毎にリレーションを選んで最も全体のコストが小さくなるような木を構成することである。従って、最初の segment の個数はメモリサイズとソースリレーションから推定されるデータサイズから決定される。図 2.12に生成された segment RD とその処理の流れを示す。また、segment 内のそれぞれのリレーションに対するハッシュテーブルの生成を stage と呼んでいる。つまり、図にあるように全部で 9 台のプロセッサがる環境で 3 つの結合演算を含む segment を処理する場合、各 stage 毎に最も効率よくなるようにプロセッサが割り当てられ、ビルドフェーズが並列に行なわれた後、プローブフェーズがパイプライン方式で行なわれることになる。

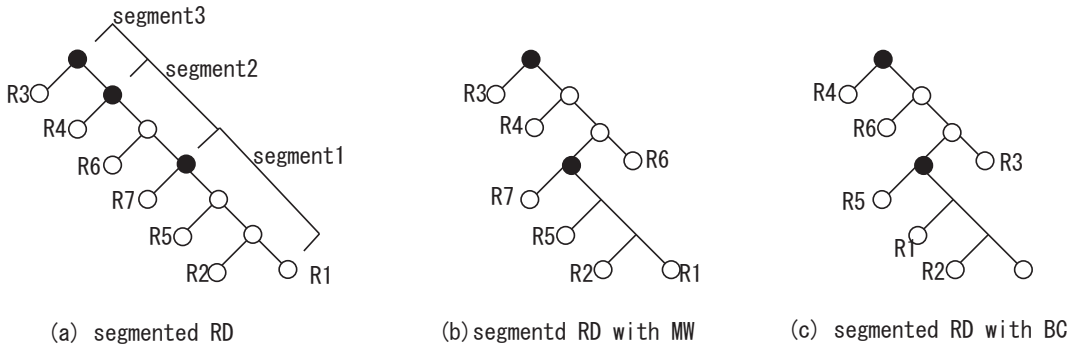

図 2.12: segmented RD 木の形状

これから、わかるように個々の segment では、各プロセッサが並列に処理を行なえるため、 segment 内で最大の処理コストのものを合わせることで、全コストとすることができる。

Segmented RD では、各 segment へのリレーションの選択を行なうにあたって、次の二つの heuristics を提案している。また、この SegmentedRD では全体のリレーションサイズとメモリサイズを元に segment 数を決定し、この segment 数から算出したリレーション数をもとに、それぞれのセグメントの内容を決定する。

Minimal Work

これは、単純にそれぞれの segment の処理コストが最小になるようにリレーションを選択するものである。つまり、リレーションがメモリを越えないか、または、予想のセグメント内リレーション数を越えない間、リレーションを選ぶものである。また、この方式で得られる tree の形式を図 2.12の (b) に示す。この方式は Static RD と比較するとその性能が良いが、与えられたリレーションによっては性能にばらつきがでる。この方式では、実際には Path Selection と同じように小さなリレーションから順に選んでしまうために、最後の segment の候補となるリレーションは大きなものになってしまい、 segment 間のサイズバランスが得られず、大きなものをまとめて一つの segment には出来ないため、 segment の個数が増え中間結果をディスクに出す回数が増えてしまうため、全体を通しての最小コストを得られない可能性がある。また、逆に最初に小さいものを選んでしまうために、最初に実行される segment のメモリが有効利用できない可能性もある。

BC

前述の MW の問題点を解決するために、 weight ファクタを入れたコスト式を導入し、各 segment 間でリレーションのサイズをバランスするようにしたものである。つまり、各セグメントごとの処理コストに対して、セグメントの中間結果が十分に小さくなる場合にはその reduction 効果を考えてリレーションを選ぶのである。この方式による tree の形式を図 2.12の (c) に示す。

この二つの heuristics を用いた場合と、 Dynamic RD と Static Optimal RD との性能比較をシ

ミュレーションを用いて行われ、結果から、 MW 及び BC の性能の優位性が示されている。しかし、この segmented RD では、基本的にメモリのサイズで tree を分割しメモリに入り切るまでリレーションを全てハッシュテーブルとして展開するため、 segment の中間結果は必ずディスクに書き戻しているため、 tree のノードが多い、あるいはメモリがかなり小さい場合にはその性能がよいかどうかは疑問が残る。また、 SD 環境でシミュレーションを行なっているため、データの偏り、ネットワークにおけるデータ転送の輻輳などについては 触れられていない。

(2) segmented RD 改良方式

Shiekita ら [104] は、提案していた segmented RD の改良版として、他の研究グループから指摘されていた中間結果のメモリへのロードを念頭において新たに結合木を生成するためのアルゴリズムとして Dynamic Programming(DP) と Greedy Heuristics の二つを提案している。

DP は、 [99] で述べられているように全てのパスのパターンについて、アクセスコストを計算し、コストの多い枝を落していくこと（prune & produce a optimal tree) を繰り返し、最終的に tree を生成する方式である。従来の DP では、１節に書いているように left-deep 木しか扱わなかったが、ここでは bushy, left-deep, right-deep と木の形式に関しても全てをしらべて最もコストの小さいものを得ることを目標としている。

一方、 Greedy Heuristics Plan としては、前の節で提案されている segmented RD 木の BC 方式を改良したものである。

(3) マルチプロセッサ上への並列結合木の最適配置

Chen ら [18] は、 Multi-Way Join の処理スケジューリング（木の生成とプロセッサアロケーション）に関して、ソートマージ結合処理を前提とした場合の検討を行なっている。ここでは、従来から提案してされている LD 木と新たに提案された heuristics による Bushy 木の生成方式について述べられている。また、こうして得られた Bushy 木におけるプロセッサの最適なアロケーションアルゴリズムを提案している。これは、分割された処理単位となる部分木ごとに空いているプロセッサ群を割り当て、後は終了同期ごとに部分木を順次アロケートするものである。

2.5.5 分散メモリ環境

(1) 多重結合演算の演算間並列化

文献 [98] にあるように、 Schnieder は並列データベースシステムの研究がいわゆる一つの結合演算のみ（Intra-Operation Parallelism) の研究に集中しており、 Inter-Operation Parallelism の研究が未だ不十分であるとして、複数の結合演算を含む複雑な問い合わせ処理の SN 環境下における処理方式について検討を行なっている。

実際、一つの CPU では、 Left-Deep 木形式での最適化を行なえば十分であったが、並列環境においては複数の結合演算を並列に処理することが可能となり、結合演算順序が逐次的に決まっている

Left-Deep 木形式が並列化を行なう上で最適な形式かどうかが検討されている。

複数の結合演算を処理する場合の木の形状はおおまかに、 Left-Deep, Right-Deep, Bushy 木の３つのパターンに分けられる。以下に、それぞれの木の形状とその並列処理単位（segment) を示す。また、この議論ではそれ々の結合演算の並列化（Intra-Operation Parallelism) については、従来から提案されているものが応用されるとして特に触れない。

Left-Deep 木

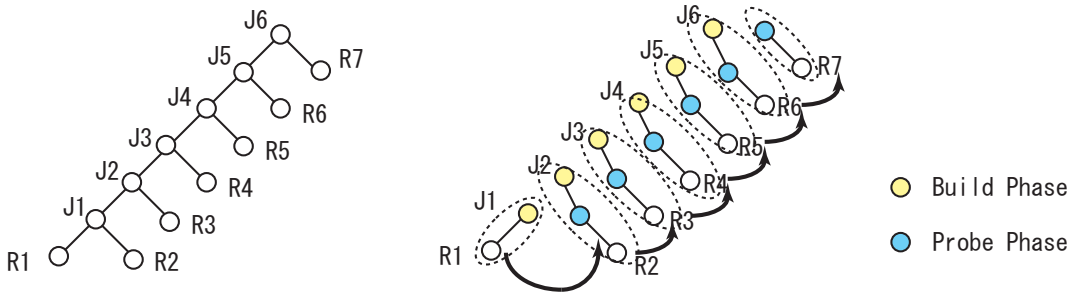

図 2.13: Left-Deep 木形式とその処理の流れ

図 2.13に示すように、それぞれのノードが結合演算を現し、各リーフがリレーションとなる。この形式は最初の結合演算から順にハッシュテーブルを生成し、ソースリレーションを用いてプローブを行ない、結果リレーションをまた、ハッシュテーブルとしてメモリに生成することの繰り返しで、結合処理が行なわれる。この処理の流れを示したのが図 2.13である。この図では、点線で囲われている部分がそれぞれの最も低レベルでの並列処理単位であり、太い矢印が処理中の同期を示している。これにより、 LD で同時に処理を行なえるのはリレーションの scan とハッシュテーブルの生成、または、プローブとその結果のハッシュテーブルの生成である。従って、次に述べる RD と異なり、メモリがいくらあっても、 operation 間の処理の並列度はあがらない。また、 LD では、一つの結合演算の中間結果を次の結合演算のハッシュテーブルと生成しているが、このとき必要なハッシュテーブルのサイズはソースリレーションから生成するよりもそのサイズを推定することが難しい。

Right-Deep 木

図 2.14に RD の木形式とその dependency graph を示す。木形式は LD と対称な形であるが、 dependency graph から得られる処理の流れは、大きく異なる。つまり、システムのメモリ量が十分あれば、左のリーフにある各リレーションはそれぞれ並列に読み出され、ハッシュテーブルを生成することになる。また、プローブフェイズはハッシュテーブルが全て出来た後、パイプライン処理で行なえることがわかる。従って、メモリ量によっては最大 N 個のビルドフェイズが並列に処理できる。以上の処理形態から、 Schneider らは、 Right-Deep 木を用いた処理方式がハッシュテーブル作成時の

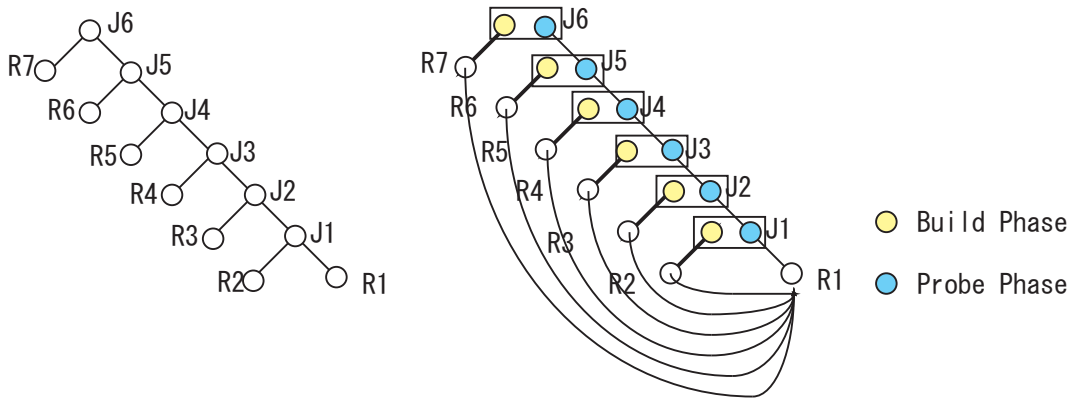

図 2.14: Right-Deep 木形式とその処理の流れ

オペレーション間の並列度の高さとパイプライン形式によるプローブ処理の並列化により、最も並列効果の得やすい方式であるとしている。一方、メモリ量に制限があると、最後には LD と同じ処理に帰着する。

LD と比較して並列化における RD の有用性が確認できたが、ソースリレーションをいくつかまとめて並列にハッシュテーブルを生成するということは、 RD 方式がよりメモリを必要とすることになる。従って、 Shineider らは、 Right-Deep 木に基づき、メモリサイズを考慮に入れた場合の処理方式として Static Rd, Dynamic Rd およびハッシュスケジューリングにより RD の三つの方式を提案している。

- Static RD これは、 Optimizer や実行時スケジューラなどによりメモリに入るようにあらかじめ、木を切るところを決めておくものである。実行時には、それぞれのブレークポイントの所までを RD に従って処理し、中間結果はディスクに書き戻す。この中間結果は次の部分のプローブ時にアクセスされる。この方式では、それぞれのブロック内のハッシュテーブルの生成は同時に行なうことが出来る。しかし、実行時の資源の影響がフィードバックできず、正確な推定ができない場合には性能が劣化することもある。

- Dynamic RD これは、実行時に Bottom-Up にハッシュテーブルを生成し、メモリが一杯になったところで一旦、ハッシュテーブルの生成を止め、プローブ処理を行ない、中間結果をディスクに書き戻す。 Static RD Scheduling と比較すると、ハッシュテーブルの生成が逐次的に処理されるため、並列化できる部分が少なくなるが、データ配置によってはこの方が性能がよいこともある。基本的には Static RD Scheduling とさほどの性能差はない。

- RD with Hash Handling 以上の二つの方法は、最もメモリが少なくなると、 LD と同じ処理をすることになる。ここではより、メモリの有効利用を行なうために、ソースリレーションをハッシュ分割することで、メモリにのるハッシュテーブルを増やし、プローブのパイプラインを長くしている。つまり、リレーションを大きくハッシュ分割して、一方のバケットのみを木

の上に向かって処理していくものである。しかし、この方法ではそれぞれの結合演算ごとに結合属性が異なると、パイプラインによるプローブ処理のときに木の上の方に向かうにつれて、それ々のハッシュテーブルに対応するタプルの数が少なくなり、ディスクに掃き出す中間結果が増える。しかし、Schneider らは、問い合わせ全体の時間は増えるかもしれないが、とりあえず最初の結果が出るという点では、木の途中でブレークポイントを設ける方法よりも早いとしている。

Bushy 木

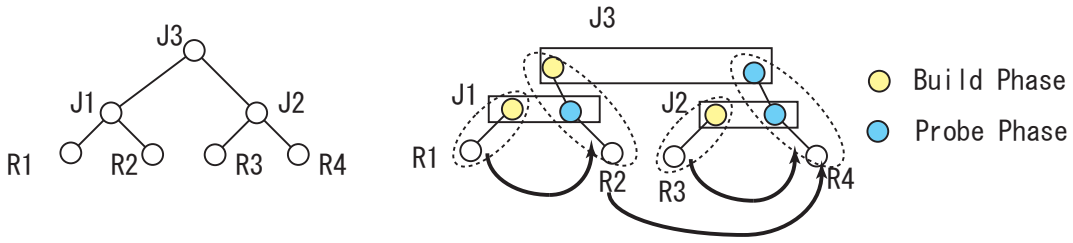

図 2.15: Bushy 木形式とその処理の流れ

この形式（図 2.15）では、上の二つの木と異なり、各ノードの枝の両側がノードであることを許している。従って、上の二つの処理方式と比較して、処理同期をとるのが難しいが、逆に実行時にクリティカルでない部分のスケジュールを適宜に行なうことによって、性能を向上できる可能性がある。

この論文では、LD 及び Static RD についての性能比較をシミュレーションを用いて行なっている。また、構築したシミュレーションにおいては、それぞれのシステム資源のパラメタを Gamma に基づいて設定しており、Gamma の実測値と比較してシミュレーションがほぼ実装した結果と同じであることを示している。それぞれの木の処理は Bottom-Up で行なわれ、結合演算を８回（９個のリレーションがあるとする）行っている。また、ここでは、中間結果は常に結合しているリレーションサイズと同じものになるとしている。つまり、RD の処理方式しては、Static RD でも Dynamic RD でも同じ結果となる。資源パラメタは、メモリが十分にあり、８個のリレーションのハッシュテーブルを展開出来る場合と、メモリのサイズを変換させた場合について、シミュレーションを行なっている。また、データ分布は均一であるが、ディスクへの分散をフルデクラスタリングした場合（５０台のディスクがあると仮定）と、それぞれ１０台のディスクに一つのリレーションが固まっている場合（９０台のディスクを仮定）についてシミュレーションを行ない、結果を報告している。

メモリが十分に多い場合のシミュレーション結果では、ハッシュテーブルを一度にメモリに展開できる Right-Deep 木方式が Left-Deep 方式よりも性能が良いという結果が示されている。同時に並列処理における資源の Contention に関しても実験がなされており、実行処理時間はともかく、データを declustering することで、ディスクのアクセスの衝突が少なくなり、Right-Deep 木における並列処理効果がいっそう、高くなることが示されている。

一方、メモリが少ない場合には、リレーションのハッシュテーブルが２つしか収まらない場合には Left-Deep 木と Right-Deep 木の処理性能がほぼ同じになるが、ある程度複数のリレーションのハッシュテーブルがメモリに展開できる場合には、上述のメモリが大い場合と同様に、 Right-Deep 木の性能がよいことが示されている。

この結果に関しては、 Inter-Operation Parallelism の考察としては、新しい問題定義がなされており、興味深いものであったが、シミュレーションの結果は非常に単純な仮定のもとに行なわれており、また、具体的な実装および最適化手法に関しても概観が述べられているだけであり、最適化などについても触れられていない。

(2) 多重結合演算の実行時最適化 (Hua)

Hua らは [48, 72, 49]、従来から提案している Adaptive Load Balancing Parallel Hash Join を基に、いくつかの Multi-way Join の木に対する実行時最適化をデータの偏りを含めていかに行なうか検討している。彼らは多重結合演算の並列処理方式をいかの４つに分類して性能比較を行なっている。

- L-LB(Linear 木 With Load Balancing) Left-Deep Tree 形式に従って、スケジューリングは行なわれ、それぞれの結合演算ごとに実行時に Load Balancing を考慮した処理（ABJ) を行なうものである。従って、結合演算間の並列度はない。この方式では、結合演算ごとにデータの偏りに対して ABJ を適用しているため、非常に極端に偏っている場合には、各プロセッサモジュールごとの性能を均等にすることはできず、データの偏りに依存した性能になってしまう。つまり、リレーションサイズが小さければ小さいほど、あるいは、プロセッサモジュール数が多ければ多いほど、各モジュール間の負荷のバランスはとりにくくなる。従って、データの偏りが激しい場合には、実際はモジュール数を減らすことで、逆に負荷のバランスを得ることが可能となり、このように一つの結合演算での負荷バランスを考慮すると、モジュール数を減らすことでこの処理負荷を均等にするとともに、他の結合演算を処理する（つまり結合演算間並列化）の可能性が出てくる。

- B-NLB(Bushy 木 Without Load Balancing) ここでは、 Sigapore の Lu ら [75] が提案している Bushy 木生成方式を用いて、結合演算間の並列化をすることで性能の向上を図っている。モジュール間の Load Balancing に関しては、考慮していない。

- NLBO(No Load Balancing Optimization) この方式では、前述の B-NLB を用いて結合演算間の並列化を図るとともに、実行時に個々の結合演算を ABJ で実行することにより Load-Balancing を図っている。

- LBO(Load Balacing Optimization) この方式では、結合演算のスケジューリングを行なう時点で、 Load-Balacing を考慮にいれて最適化を行なっている。また、この最適化をここでは実行時に行なうとしている。処理の流れは以下の３つのフェーズからなっている。

1. Hash Phase
全てのソースリレーションを ABJ と同じく小さな Sub-bucket に分割してディスクに書き戻す。

2. Optimization Phase
ここでは、 Lu ら [75] が提案した Bushy 木の生成方式と同じような形で同時に処理できる結合演算を探索するが、そのときに、 Hash Phase で得られたデータの偏りの情報をもとにプロセッサの割つけと最も処理を行なう上で時間のかかるプロセッサモジュールを見つけ出すことで、処理負荷のバランスとともに最小コストで同時に処理できるリレーションの組合せを見つけることができる。

3.Execution Phase
実際に、 Optimization Phase でスケジュールされた通りに結合演算の処理を行なう。ここで、処理された中間結果はディスクに書き戻される。まだ、処理されていないリレーションがあれば、 Optimization Phase に戻る。

この４つの方式に対して、本報告ではシミュレーションによりよる性能評価が報告され、バケットのサイズの偏り、リレーションサイズの偏りなどのパラメタを変えた結果では、いずれも LBO 方式の性能がよいこと示している。

この方式では、基本的には、 Lu ら [75] が提案した Bushy 木を生成する方式と同じ形式で最適化を行ない、実行時に彼らに提案した ABJ を用いることで、従来の方式では考慮されていなかったデータの偏りに対するマルチウェイジョインの最適化を提案している。シミュレーションの結果から、LBO の性能がすぐれていることがわかる。また、環境によっては Bushy 木の性能が、 Left-Deep 木と同じかそれより良いことが確認できる。 L-LB の方式のところで触れたように、データの偏りとそれに対するプロセッサの並列効果と処理負荷のバランスに関してはおもしろい考え方を提案していると思われる。しかしながら、この生成方式では、中間結果はかならずディスクに書き戻しており、パイプライン効果などについても検討されておらず、メモリの有効利用などとからめて、最適化の余地が十分残されていると思われる。

2.6 並列データベースシステムの実例

2.6.1 共有メモリ環境における並列データベースシステム

(1) XPRS

XPRS[109, 42, 43, 44] は米国 California 大学 Berkley 校で開発された共有メモリ型マシン Sequent 社 Symmetry 上での並列データベースシステムである。 XPRS では、共有メモリ型マシンの特徴である容易にアプリケーションの並列性を引き出し、実装ができること、及び SN アーキテクチャと比較して容易に負荷バランスをとることができることを活かし、実際に Intra-Operation の並列処理とその最適化について考察、実験している。また、 Inter-Operation の並列処理及び処理負荷のバ

ランスに関しては、それぞれの関係演算の特徴をコスト式に反映することで、 CPU 中心の Operation と I/O 負荷の高い Operation を抽出し、システム全体としての CPU-I/O バランスを得られるようなスケジューリングを行なうことで、性能向上を図っている。

XPRS の演算内並列処理 XPRS で行なわれている単一の演算内の並列処理の最適化は、二つのフェーズを通して行なわれる。

フェーズ 1 において、全てのバッファがフリーである場合の最適逐次プランを選びだし、

フェーズ 2 において実行時に実際のバッファサイズとプロセッサの数からこの最適逐次プランを並列化した場合にもっともコストの小さいものを選び出す。

また、この方式の有効性を確認するために、 XPRS では以下のような二つの仮説を立て、実際にベンチマークを用いてその仮説が成り立つことを示している。

- The Buffer Size Independent Hypothesis
  ハッシュジョインのスレッショルドを満たす大きさのバッファであれば、サイズが異なっていても最適な逐次プランは同じである。
- The Two Phase Hypothesis
  共有メモリ環境においては、ベスト逐次プランの並列化がベスト並列プランである。

Hong らは、この仮説に関しては、 XPRS が SE アーキテクチャを採用しているため、十分成り立つとしている。この仮説の上で最も問題になりそうなインデクスを用いた検索への考慮を反映し、逐次プランの中に二つのノード（データスキャンにインデクスを用いるか、逐次アクセスするか、または結合演算を行なうときにハッシュジョインにするかインデクスを持ちいたネストループ処理にするか）をいれておき、システム資源の大きさからいずれかのノードを実行時に選択するということで解決しうるとしている。提案された仮設をウィスコンシンベンチマークを用いて、最適実行時間とわずか数パーセントしか違わず、仮説が XPRS 上ではかなり有効であることを示している。

しかしながら、逐次プランを採用しているということは、複雑な問合せに関しては従来からの Left-Deep Tree を採用していることになり、最近のマルチジョインの研究動向などを考えると、プランの検索範囲を狭めてはいるが、必ずしも最適並列プランであるとはいえない。

XPRS の演算間並列処理 Bushy 木を用いた演算間並列処理方式について、提案されている。 XPRS では、複数のオペレーションがあるときにそれぞれのタスク毎に CPU bound か I/O bound であるかを調べ、それぞれシステム資源を最も有効に利用出来るような CPU bound と I/O bound のタスクのペアを組み合わせて並列に処理を行なうようにスケジューリングする。次に実行時にプロセッサの割当をそれぞれの並列方式（page partitioning または range partitioning）によって多くしたり、減らしたりすることで実行時のシステム資源の有効利用を図っている。これにより、単に Intra-Operation Parallelism を実現したときと比較して多い場合には２５%程度の性能改善が見られると

シミュレーションによる報告をしている。また、実行時にプロセッサ割つけを行なわない場合、逆にプロセッサが使われない部分が多くなってしまい、 Intra-Operation Parallelism による最適化と比較しても、性能が得られない。

(2) DBS3

INRIA で開発されている DBS3 は、共有メモリ環境（MultiMax 520) 及び共有ディスク環境 (Esprit Project) 上に実装されつつある並列データベースシステムである。 DBS3 では、それぞれの Operation 毎に並列可能か、データフローの同期をどこでとるかなどの記述を行なうことで、 Optimization を通してコストを計算し、最適な並列処理方式を得られるようにしている。

(3) Volcano

Volcano では、逐次プランに Exchange Operator という概念を入れることで、並列化できるところを抜きだし、効率のよい並列処理を行なうことを目指している。

2.6.2 分散メモリ環境における並列データベースシステム

(1) Grace

GRACE[61] はハッシュに基づいたアルゴリズムを採用した並列データベースマシンであり、任意の関係代数演算をデータストリームに追随して 0(N) 時間で処理できる。図 2.16に示されるように GRACE のシステム構成は、フィルタリング機能を持ったディスクモジュール、中間リレーションを生成するためのメモリモジュール、ソータを有するプロセッシングモジュールからなり、それぞれリングバスで繋がれた攻勢となっている。 GRACE ではハッシュによる動的クラスタリングを用いて関係代数演算の処理を行う。処理負荷の重い結合演算では、データを互いに独立なクラスタに分解し、これらのクラスタを複数台のプロセッサに割り付けることにより自然な並列処理が実現できる。

図 2.16に GRACE における関係代数演算処理の流れについて示す。 GRACE 上での関係代数演算処理では、まず、ディスクモジュールからデータをメモリモジュール上にステージングする。この時にデータをディスクモジュール上でフィルタリングすると同時にハッシュ関数を用いて分割し、より小さな処理単位 (クラスタ) としてメモリ上に展開する。クラスタをメモリ上に格納する際、 GRACE では各クラスタをある特定のモジュールに割り付けず、図 2.16に示すようにそれぞれのクラスタを各メモリに均等に分割、格納する。この様にすることで、ハッシュ後のデータクラスタの大きさが不均一な場合でもメモリ上にデータが均等に存在することになり、オーバーフロー処理を効率よく行える。プロセッサ群はそれぞれのクラスタを異なるバンクから処理し始め、全バンクをパイプライン的にアクセスすることにより、競合なしにデータを並列に読み出し、処理を行う。図 2.16に各プロセッサがパイプライン的にクラスタを処理する流れを示す。プロセッサは、それぞれ特定のデータクラスタをすべて読み込み、プロセッサ上で処理を行う。図 2.16に示されるように、プロセッサモジュール 0(PM0) がクラスタ 0 の一部をメモリモジュール 0(MM0) から取り込んだとする。続いて PM0 は MM1

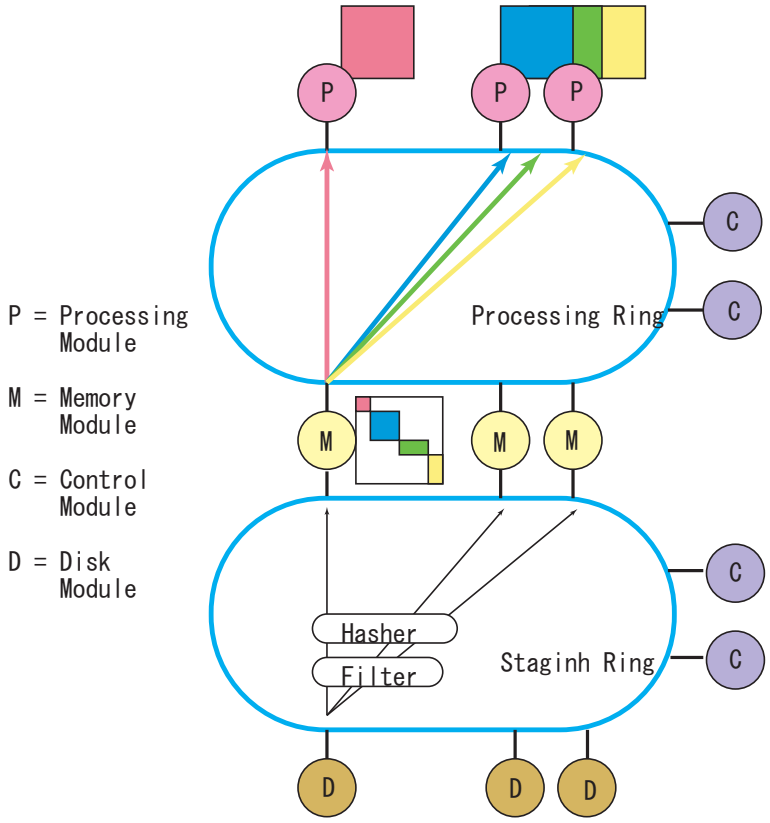

図 2.16: Grace のシステム構成

上にあるクラスタ 0 を取り込みに行く。一方、 PM 1 は PM 0 が MM0 を開放すると同時に MM) 上にあるクラスタ 1 のデータを取り込む。全てのクラスタ 0 を取り込んだ PM 0 は、ローカルメモリ上でクラスタの処理を行う。同様にして PM1、 PM2 がそれぞれ割り当てられたデータクラスタを取り込んで処理を行う。

GRACE では、プロセッサ台数等の物理的なパラメータとバケット数などの論理的なパラメータたは独立しており、リレーションの大きさに応じて処理の並列度を柔軟に設定することができる。クラスタはメモリモジュールにわたって分散されており、それをパイプライン的に処理する。従って生成されるバケット数はリレーションの大きさに応じて制御され、有限なプロセッサ資源を超える物についてはシリアルに処理される。また、プロセッサは O (N) のハードウェアソータを装備しており、データの流れを乱さずに処理を行うことができる。この様に全ての処理が O (N) 時間でディスク流に追随した形で行われ、かつモジュール台数分の並列度が達成されることから任意の関係代数演算が O (N/K) (K: モジュール数) 時間で実行できる。

(2) Bubba

Bubba[8, 122, 23] の試作機は４０ノードの FELX/32 マルチプロセッサと４０台のディスクから構成される。ハードウェアアーキテクチャ自体は共有メモリ構成であるが、 Bubba システム自体は分散メモリ構成を前提に構築され、共有メモリはノード間通信に用いられるのみである。 Bubba では, 通信機構とノード間の実行時処理調停を行う Interface Processor, データの格納とデータベース演算処理を行う Intelligent Repositories、および Checkpoint/Logging Repositories の３種類のノードから構成される。 Bubba は FAD と呼ばれるオブジェクト構造をもつ永続的データを対象とする処理言語をユーザインタフェースとして提供している。 FAD では、従来の関係データベース記述言語 SQL などとは異なり、複雑なオブジェクト処理を支援し、また、データオブジェクトの格納状態により実行時の並列処理化を行う。また、 Bubba では、各ノードに格納されている永続データを仮想空間にマッピングすることで、実行時処理プロセスが従来のファイルおよびページアクセス処理を通すことなく、データへの単純なアクセス支援を行っている。

(3) Teradata: DBC/1012

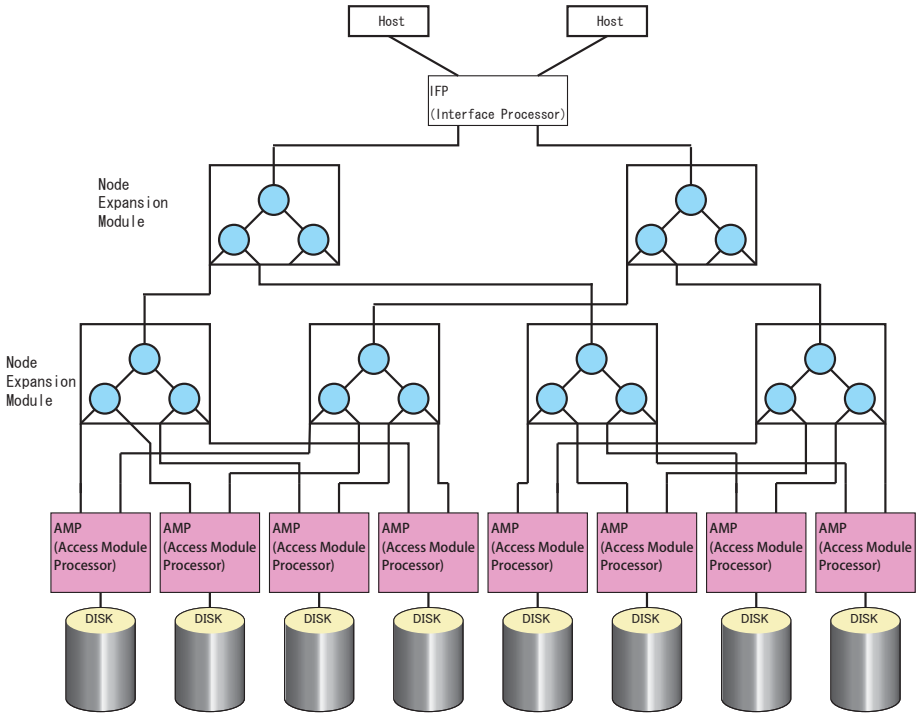

図 2.17: Teradata の構成および Y ネット

Teradata 社は 1000 台規模まで拡張可能な大規模並列データベースマシン DBC/1012 を商用化した [116]。 DBC/1012 は複数の Access Module Processor （AMP）と呼ばれるプロッセサとディスクから構成されるユニットを Y ネットと呼ばれる木構造ネットワークにより結合したアーキテクチャを採用しており、 AMP の数を調整することにより、中規模から大規模サイズ（およそ１０ TeraBytes）

データベースまで柔軟に対応できる。 AMP に加え、 Interface Procesor （IFP）なるプロセッサが配置され、ホストから高速 I/O チャネルやローカルエリアネットワークを通じてアクセスされる。図 2.17に DBC/1012 の概念構成図を示す。 DBC では、関係データベース・モデルを採用しており、ユーザインタフェースとしては IBM/SQL と互換性のある DBC/SQL が提供されている。ユーザは、ホスト上では IBM/SQL 文を用いてバッチ、オンライン・トランザクション、インタラクティブいずれのモードでも利用することができる。発行された問合せは Teradata Direct Program （TDP）を通してシステムに送られる。 TDP では、ブロックマルチプレクサチャネルを通して DBC/SQL 要求をシステムに転送し、システムから結果を受理し、ユーザに返す。

システムでは以下のように並列に問合せを処理する。ホスト上のユーザが発行した問合せ文は DBC /1012 上の IFP に転送される。 IFP は、問合せ文の正当性のチェックを行うとともにアクセス権を確認し、 AMP の処理単位となる処理ステップを生成する。各命令は、 Y ネットを通じてＡＭＰ群にブロードキャストされる。 DBC では並列度を高めるように、マルチ・ステートメント、マクロ、シングル・ステートメントの各レベルで並列性を抽出している。データが複数のディスクに分散している場合には各ＡＭＰ群は互いに独立に各自にデータに対して処理を施す。一方、インデスク等によりデータが絞れる場合には該当のＡＭＰのみを駆動し、他のＡＭＰは別タスクの処理を行う。フィルタ処理、ソータ処理、インでクス管理等はすべて AMP 上のマイクロプロセッサによって実行される。処理結果はＹネットを介して IFP に戻される。

ＹネットはＤＢＣシステムの心臓部であり、この名称はネットワークがＹを逆に木状に組み合わせたその形状からとられている。図 2.17にＹネットの構成を示す。Ｙネットは IFP から AMP 群へのメッセージブロードキャスト、 AMP 間の通信、 AMP 群からの IFP への処理結果の転送、さらに処理結果転送中のマージソート機能等がある。つまり、Ｙネットは単なるネットワークではなくマージ機能を有した能動ネットワークと考えられる。Ｙネットは独立なネットワークが二重化されており、電源も別にとられている。通常は二つのネットワークがトラヒックを二分するが、一方のネットワーク障害時には、もう一方のネットワークだけで続行できるように設計されている。さらにシステムーホスト間でも 1 つのチャンネルに 2 つの IFP を対応させることで、データベースシステムとしての障害回復対策を行っている。

AMP は配下の Disk Storage Unit (DSU) のデータ操作を行う。通常、データは複数の AMP に渡って分散され格納されている。ディスクは、システム領域、基本データ領域、不フォールバック領域の 3 つに分かれており、フォールバック領域にはユーザの指定により、基本データ領域の複製を格納できる。従って一つのディスクに障害が起きても処理を続行することが可能である。また、フォールバック領域をディスククラスタ単位（２～１６台）毎に行うことにより、処理の効率化を図ることができる。

DBC/1012 上での関係代数演算処理は GRACE に極似しており、ハッシュとソートに基づいている。つまり、結合演算ではまずＹネットを介して AMP 群にわたってハッシュによるクラスタリングを施し、各 AMP ではソートに基づき処理がなされる。以上のように DBC/1012 では、処理負荷を分散することで通常のメインフレームと比較して非常に高い性能が得られるとしている。たとえば、

DBC/1012 MODEL3 では、 CPU として、 AMP, IFP ともに Intel 80386 を用いており、メモリは 4MB(8MB に拡張可能)、さらに不揮発性のディスクキャッシュ 2MB で構成されている。この MODEL ３では最大構成で 3000MIPS、 250 台のプロセッサ構成でほぼ 750MIPS と言う性能を達成している。

(4) Gamma

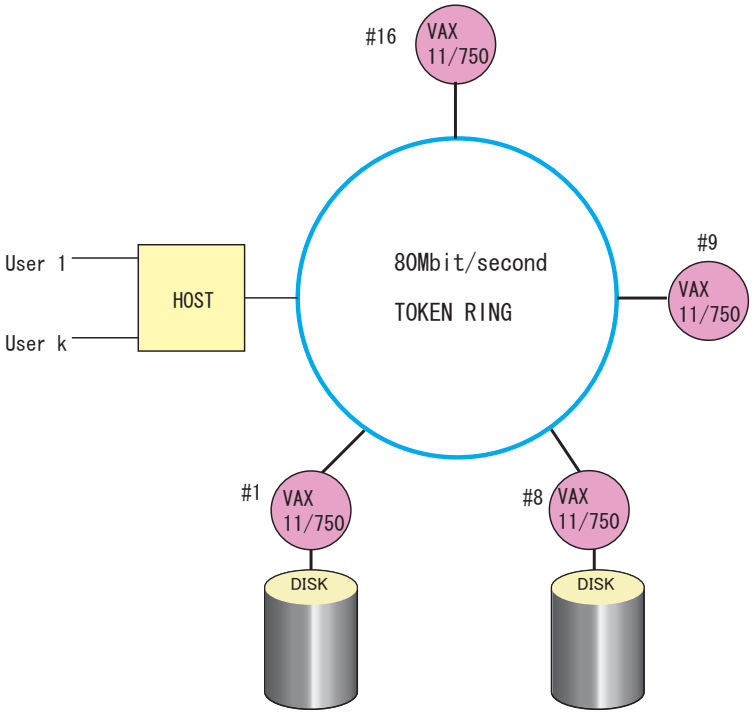

図 2.18: Gamma のシステム構成

Wisconsin 大学で開発された Gamma[28, 27, 29] は Grace と同様に動的クラスタリングアルゴリズムを採用している並列データベースマシンである。図 2.18に示すように、システムは、ノード (プロセッサとディスクの組合せ) 群をリンクバスによって結合するという簡単な構成をしている。 GAMMA での関係台数演算処理は以下のように行われる。まず、各ディスク付きノードからデータが読み出され、ノード上でフィルタリング処理が施された後、スプリットテーブルで定義されているノードへ送られる。 GAMMA では、データ分割の方法として、ハッシュを適用するほか、ラウンドロビン方式、レンジ指定による方式、あるいはハッシュとレンジ指定を組み合わせたものなどがある。データを受け取ったノードは、各々並列に指定された処理を行う。また、結合演算の性能をあげるために Hash Bit Filter の機能も容易されている。 Gamma では、最初のリレーション処理時にデータ分割を行う各ノードで Bit Map Table を生成し、一旦スケジューラに転送する。スケジューラでは、伝送された Bit Map Filter を収集、まとめて次のリレーションを分割するノードへ転送する。次のリレーション

分割時に Bit Map Table を利用して不要なタプルを削除する。

Gamma では、 VAX 11/750(内 8 台がディスク付加、各 2MB のメインメモリ付き) を 80Mbit/sec のリングバスで結合されたハードウェア構成で大規模データベースにおける性能評価が試みられており、 Teradata 社の DBC/1012 AMP 20 台 (ディスクは 40 台) と比較して、選択演算、結合演算ともに高い性能が得られている。これは GRACE で提案されている動向ハッシュクラスタリングが非常に効果的であることを示している。

また、 Gamma 上では、動的ハッシュクラスタリングアルゴリズムに基づく並列アルゴリズムとして、 GRACE ハッシュ方式の改良版である Hybrid Hash 方式が提案されている。

(5) RINDA

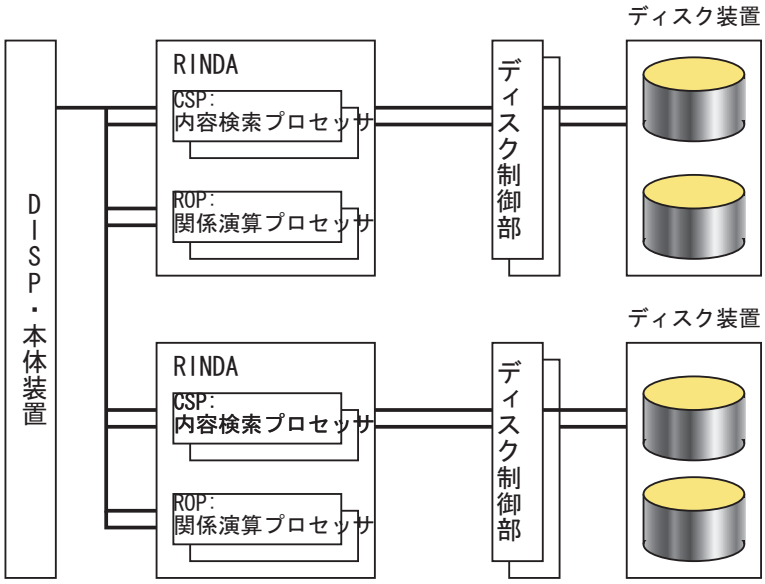

図 2.19: RINDA のシステム構成

RINDA は NTT で開発された関係データベース・マシン [132, 50, 95] であり、汎用計算機では負担の重い処理を専用ハードウェアによって実行することにより、検索処理時間の大幅短縮と CPU 負荷の削減を意図するものである。図 2.19に示すようにシステムは、システムは、 CSP(内容検索プロセッサ) と ROP(関係演算プロセッサ) から構成され、それぞれ独立の I/O インタフェースでホストと接続される。 CSP は、フィルタプロセッサ、 ROP はソータと位置付けできる。二次記憶システムは CSP からもホストからもアクセス可能である。

RINDA における問合せの処理は以下のように行われる。ユーザは、ホスト上で業務プログラムを SQL で記述する。ホスト上の DBMS ソフトウェアが SQL 文の解析、最適化を行い、 RINDA に対する要求の発行、通信の管理などを行う。解析された問合せにしたがって、まず CSP によるデータの検索が行われ、結果はホストに転送される。必要があればホスト上で条件検索を行った後、ホストは ROP に CSP からのデータを転送、 ROP 上でソート処理、結合演算、集計演算などのためにふ

るい落とし処理が行われ、ホストに返送される。結合演算の突き合わせ処理などを行った後、処理結果はDBMSを通じて出力される。

RINDAの特徴の一つはCSPを用いた高速な内容検索処理にある。各ディスク制御装置毎にCSPを設け、ディスクのマルチトラックリード機能を用いることにより、１シリンダ内の全てのページを途中でシーク・サーチ動作なしに読み出す。各CSPは、ディスクに格納されたリレーションを指定された条件でon-the-flyでサーチし、条件に合致したタプルの中から処理に必要なアトリビュートのみを取り出してホストに転送する。このため、CSPの機能として、SQL文で指定された述語判定として定数項との比較、任意の論理式の判定、出力列の抽出、集合関数演算などを専用回路で処理している。さらにリレーションを複数のディスクに分散して格納し、複数のCSPを用いてディスクへの並列アクセスを実現することにより、検索処理の高速化が可能である。

さらに、RINDAでは結合演算を高速化するために専用ハードウェアROPを開発している。ROPは、ホストから転送されてきたタプルを指定された条件でふるい落としおよびソートを行い、その結果をホストへ返送する。RINDAではデータ量が少ない場合にはホスト上でネストループ方式で、多い場合にはホストとROPによるソートマージ方式で結合処理を行う。

(6) Tandem

Tandem Nonstop SQLシステム[115]は４多重ファイバリングで接続されたプロセッサから構成される。Tandemは前述のTeradata, Gamma, Bubbaなどと異なり、フロント／エンドの切り分けはなく、通常のデータベースサーバおよび処理プロセスが同じシステム内で動作する。各MIPSプロセッサごとにディスクがついており、ディスクの内容は二重化されている。また、各ディスクごとに大容量のRAMキャッシュが用意されており、このキャッシュ上にあらかじめ先読みしたデータを保持し、フィルタリング、データ操作などをこれらのディスクサーバ上で行うことで、各ノード間の通信コストをおさえ、高い性能を得ている。

(7) Super Database Computer (SDC)

SDC[113, 114]はハイブリッド並列アーキテクチャを採用しており、全体を無共有型とすることによりシステム拡張のスケーラビリティを、処理ノードを共有メモリ型のマルチプロセッサにすることで、軽い通信コストとノード性能のチューニングを可能としている。また、入出力性能を向上するためにディスクとネットワークに専用の入出力プロセッサを用い、共有バスとネットワークにおいてデータパスと制御パスの分離を行っている。これにより、データインテンシブなデータベース処理(意志決定支援システム等におけるアドホックな問合せ処理)を高速に処理することが可能となる。

SDCのハードウェアアーキテクチャを図2.20に示す。８台のデータ処理モジュール(DPM)と呼ばれる処理ノードを、処理データ用のデータネットワーク(DNet)、フロントエンド計算機との通信に用いるコントロールネットワーク(CNet)の２系統のネットワークを相互に結合した構成になっている。CNetには汎用性を考慮して、10Mbps Ethernetを用いているが、DNetは大量データを転

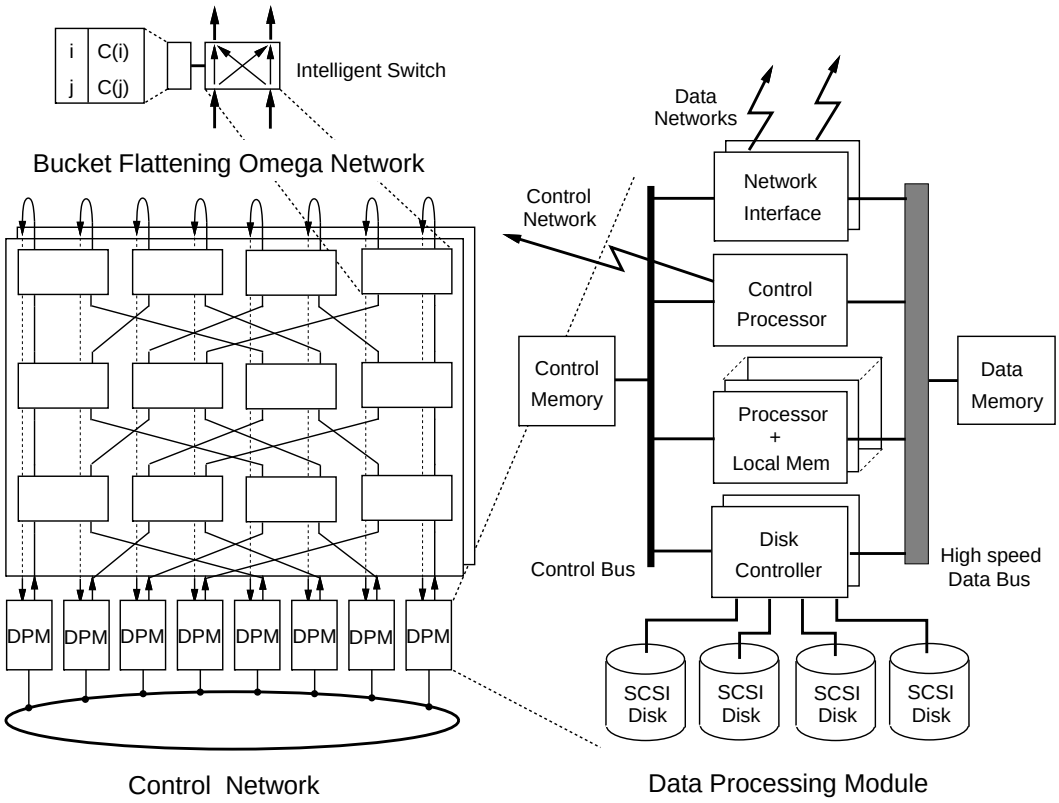

図 2.20: SDC のシステム構成

送するための高速性に加え、並列ハッシュ結合アルゴリズムのハードウェア支援を行うバケット平坦化機能をもつオメガネットワークを独自に開発した。

SDC では、リアルタイムカーネルである VxWorks をコアとして使用しており、その上にデータベース並列処理に特化した専用カーネル DBKernel を実装している。その結果、ハードウェアや物理メモリに直接アクセスすることができ、オーバヘッドを低減している。ただし、ノード無いのプロセッサ間の同期プリミティブなどは備わっていないため、 SDC 専用に実装されている。

また、ディスクおよびネットワークに専用の管理 CPU を用いているたえ、入出力管理デーモンは直接入出力プロセッサ上で動作させている。ディスクに関しては、 SCSI のデバイスドライバと一体化した構造になっており、ファイル単位のリード要求に対してプロトコルが許す限りのブロックを一度に要求し、動的に DMA バッファを確保することで、デバイスに対しするコマンドオーバヘッドを削減している。また、ファイルのリード中に優先順位の高いライト要求などがくると転送中の SCSI コマンドをアボートする処理を行う。

第 3 章

機能ディスクシステム

3.1 機能ディスクシステムとは

機能ディスクシステムは関係データベース処理の高速化を目的とする新しい高性能二次記憶システムである。従来の商用関係データベースシステムが関係代数演算処理手法として単純なネストループ技法、ソートマージ技法を採用しているのに対し、機能ディスクシステムでは動的クラスタリングアルゴリズムを採用し、大幅な処理負荷の低減を図った。また、ハッシュ操作、クラスタ管理機構、物理ブロックからのレコードの切り出し機構、オンザフライ操作等をハードウェア化し、ディスクからのデータ転送速度に追従したクラスタリング処理を可能とすると共に、更に複数台のマルチプロセッサを導入し、生成されたクラスタを並列に処理することにより逸そうの性能向上を図っている。通常、OS とその上に構築されるデータベースシステムの間の不整合性から大きな性能低下を招くが、これを解決すべく、機能ディスクシステムでは動的クラスタリング技法に最適化した専用入出力ドライバ、バッファ管理ルーチンを開発した。データベース処理機構を二次記憶システムに導入することにより、大幅な性能向上を目指す機能ディスクシステムの有効性を確認するために、試作システムを構築し、性能評価を行った。試作システムは１台のディスク、４台の MC68020 の簡単な構成にもかかわらず、ウィスコンシンベンチマークを用いた性能評価に於いて現存の商用関係データベースシステムに比べ著しく高い性能を達成することが出来、本システムの有効性が確認された。

3.2 研究の背景

高度情報化社会の発展により、近年のデータベース技術の進歩には目覚しいものがある。特に、関係データベースは高いデータの独立性、簡易なユーザインタフェース、強固な論理的基盤などのすぐれた特徴を有し、現在、データベーシステムの中心となっている。しかしながら、これらの利点を実現するためシステムに課せられた負荷は巨大なものになり、その高速化、高性能化が大きな課題とされてきた。実際、関係データモデルが Codd によって提案されて以来商用化に到るまでには１０年以上の歳月を要し、更に現時点においてもいまだ充分な性能が得られていると言えない。また、最近の IC 技術の飛躍的な進歩はは計算機システムの急速な進歩を推す原動力となっている。特に，関係データベース・システムの分野では大規模なデータを高速に処理することが急務となっており，この問題を解決するために様々なアーキテクチャの提案，システムの実現が行われているが大きな成功を治めているとは言い難い。

低性能の原因は大きくは次の二つに分けられる。

1. 処理負荷の増大
2. データベースシステムとオペレーティングシステムの不整合性

第一の問題点は関係データベースにおける非手続き的な処理要求自体の負荷が、従来の木構造モデル、ネットワーク構造モデルに基づくナビゲーショナルなデータベース処理に比べて高くなったと考えられる事である。例えば、関係データベースにおいては二つのリレーションの動的な結合を実現するために結合演算が導入されているが、その処理負荷は極めて重く、単純なアルゴリズムでは両リ

レーションのタプル数の積に比例するコストが必要となってしまう。この問題を解決するには、より効率的なアルゴリズムん開発、特に単一 CPU 環境下でのアルゴリズムだけではなく、並列処理環境下での処理アルゴリズムなど新たな処理形態を追究する必要がある。

第二の問題点は関係データベースシステムの実装技術が未だ未熟で充分に効率化が行われていないという点、特にデータベースシステムとオペレーティングシステムの不整合性に基づく非効率性にある。 DB2 や Ingres など汎用機上の現在の商用関係データベースシステムのほとんどは、現存のオペレーティングシステムの上でに構築されており、データベース管理システムはオペレーティングシステムの提供する入出力システム、ファイルシステムに依存しているのが現状である。しかし、ここ数年の研究から従来のアプリケーションの入出力特性とデータベースシステムのそれとは大きく事なり、オペレーティングシステムの入出力道さがデータベースにとって極めて非効率であることが指摘されている [108, 82, 122]。これは二次記憶に対する入出力負荷の重いデータベース処理においては致命的な欠陥といえ、この問題の解決には現存のオペレーティングシステムの二次記憶入出力ルーチン、バッファ管理部をデータベースシステムの特性に適合した形態に変更することが必要である。より直接的なアプローチとしては、データベース専用のオペレーティングシステムの開発が考えられる。

関係データベースシステムの高性能化に対し、データベースマシンの研究並びにその商用化が数多く行われてきた。古くは RAP[90] や CASSM[111] などディスクの各ヘッドにロジックを付加し連続サーチを行うものや連想プロセッサ STARAN を利用した RELACS[5] など専用ハードウェアによって第一の問題点を解決しようとするものであるが、性能／価格比の点で成功しなかった。その後、並列処理に適したハッシュによる関係演算処理アルゴリズムが開発され [61]、処理負荷を従来の全数マッn チング方式では O(NxM) 時間かかっていたものが O($\sum n \times m$) 時間へ大幅に低減することが可能となり (ここでは N、 M は 2 つのリレーション各々のタプル数、 n,m はハッシュによって分割されたクラスタのタプル数を示す)、このアルゴリズムを基に Teradata 社の DBC/1012[116] やウィスコンシン大学の Gamma[27] が構築された。特に DBC/1012 は現在大規模データベース応用において成功をおさめている。また、わが国においてはベクトル処理機能をデータベース処理に適用した日立の IDP[66] や専用ソートハードウェアを搭載した NTT の RINDA[132] などの商用化が活発である。一方、第二の問題点を克服するアプローチは、各種関係データベースソフトウェアに於いて種々の改良がくわえられているものと考えられ、版を重ねるごとに次第に性能の向上が図られてきた。データベース専用のオペレーティングシステムを開発した例としては、 Britton Lee 社の IDM[117] が挙げられる。これは、 Z8000 マイクロプロセッサ上に専用のオペーレティングシステムを用いてデータベース管理システムを構築したものであるが、 Ingres と DBMS のコードはほぼ同容量であるにもかかわらず、性能は数倍向上させることに成功している [122]。

このようにデータベース性能の向上を目的とした種々のアプローチが試みられており、その中のいくつかは市場に於いて成功しているものの、未だ不充分であり、後に述べるように我々が提案する機能ディスクシステムと比べると大きく性能が劣っている。

本章では以上の背景より、上述の二つの問題点を抜本的に解決すべく、「機能ディスクシステム」[56] と呼ぶ高性能関係データベースシステム構築のための新しいアプローチを提案する。機能ディス

クシステムでは、第一の問題点に対し、ハッシュを用いた動的クラスタリング手法を関係データベース処理に対する基本アルゴリズムとして採用し、フィルタリング、クラスタリングを専用ハードウェアで支援すると同時に複数台のマルチプロセッサによる並列処理機構を導入することにより、大幅な性能向上を目指した。第二の問題点に対処するため、既存のオペレーティングシステムの入出力ルーチンを利用することなく、成就の動的クラスタリングアルゴリズムに適合させて新たに入出力ドライバ、バッファ管理ルーチンを開発した。そのアプローチの有効性を明らかにすべく試作システムを構築し、ウィスコンシンベンチマークにより性能評価を行った。その結果、試作システムは極めて簡単な構成にもかかわらず、既存の商用システムに比べて著しく高い性能を達成できることを確認した。

我々は, このデータベースシステムに課せられている高速化, 高性能化の課題について,"関係データベース処理に内在している処理負荷の増大 ","データベースシステムとオペレーティングシステムの不整合性 " という二つの大きな問題点から考察を行い, これらを解決すべく「機能ディスクシステム」なる高性能関係データベースシステム構築のための新しいアプローチを提案した [134]。まず, 関係データベースに内在する処理負荷の増大に対し, ハッシュを用いた動的クラスタリング手法 [61, 26] を関係データベース処理の基本アプローチとして採用し大幅な処理負荷の低減を図った。同時にフィルタリング操作やクラスタリング機構などをハードウェアで実装しディスクからのデータ転送速度に追従したクラスタリング処理を可能とし, また, 複数台のプロセッサを導入して生成されたクラスタの並列処理を行うことで, 一層の性能向上を図った。一方, OS と DBMS の不整合性に関しては, ハッシュを用いた動的クラスタリングアルゴリズムに適した専用の入出力ドライバ, クラスタを単位とするバッファ管理機構等を新たに開発した。機能ディスクシステムの性能を明らかにするために試作機を構築しており, ウィスコンシンベンチマーク [6] を用いてその性能評価を行った。その結果, 試作機は極めて簡単な構成にも係わらず, 既存の商用関係データベースシステムに比較して高い性能を得られることを確認した [56, 134]。さらに拡張ウィスコンシンベンチマーク [28] を用いて 200MB 程度の大規模リレーションに対する性能評価も行い,20 台のプロセッサと 40 台のディスクから構成される Teradata 社の DBC/1012,17 台のプロセッサと 8 台のディスクから構成される GAMMA と比較して 4 台のプロセッサと 1 台のディスクのみで構成される機能ディスクシステムが十分に高い性能をあげていることが確認された [134, 57, 58, 59]。

3.3 既存の関係データベースシステムの低性能性に於ける問題点と機能ディスクシステム

既存の関係データベースシステムの抱える低性能性は前節で述べたように, (1) 処理負荷の増大と (2) データベースシステムとオペレーティングシステムの不整合性という二つの問題に起因すると考えられる。ここでは、それぞれに対しより詳細に考察すると共に機能ディスクシステムにおいて採用したアプローチについて述べる。

3.3.1 処理負荷増大の問題点と機能ディスクシステムにおけるアプローチ

関係データモデルはユーザに対し非手続き的な問合せインタフェースを提供するという点で従来のデータモデルに比べ大きく優れており、また、その処理はレコード単位の操作ではなくレコード全体に対する集合操作を基本としている点が大きな特徴といえる。しかし、実装の観点からは結合演算、射影演算をはじめとする関係代数演算操作の処理負荷は極めて重くなり、エンレガントな関係データモデルの実現の足枷となってきたといえる。とくに結合演算は二つ以上のリレーションの動的な結合を実現するための関係データベース特有の演算であるが、その高速実行は長い間データベース研究の大きな課題であった。結合演算に対する処理方式としては、当初、最も直接的なネストループ方式が採用された。この方式は処理負荷が両リレーションのタプル数の積に比例するため、大規模リレーションの操作では性能が著しく低下する。現時点でも多くの商用システムがこのアルゴリズムを採用している。その後、ソートマージアルゴリズムにより改善が試みられている。これは、両リレーションを結合アトリビュートによってソートすることにより線形時間で結合処理ができることに着目したものである。カリフォルニア大学バークレー校で開発した Ingres はネストループ方式を採用していたのに対し、商用版 Ingres ではソートマージアルゴリズムに変更することにより、後に示すようにその性能はかなり改善された。現行の商用システムの殆どがこれら二つのアルゴリズムのいずれかを採用しており、未だの性能は不充分である。

これに対し、喜連川[61] や Bratbergsengen[12] らはハッシュを用いた動的クラスタリングアルゴリズムを提案した。ハッシュによる結合演算アルゴリズムは、 O(n,m) 時間で処理可能であり (ここでは n,m は二つのリレーションのハッシュにより分割された各々のクラスタのタプル数を示す)、[26, 96] などの性能評価結果から明らかなようにネストループ方式やソートマージ方式に比べてはるかに性能が優れている。さらにこのアルゴリズムは並列処理に適しているという大きな特徴を有していると同時に、結合演算同様処理負荷の重い他の射影演算、集計演算、除算などにも適用することができるという利点がある。

性能に関する第一の問題点を解決するためには効率の良い新しいアルゴリズムの開発が不可欠であり、機能ディスクシステムでは動的クラスタリング手法をその基本アルゴリズムとして採用することにした。動的クラスタリングに基づくアルゴリズムで、まず二つのリレーションの各タプルに対し、結合アトリビュートにハッシュを動的に施し、同じハッシュ値を有するタプル群を一つのクラスタとしてリレーションを分割 (クラスタリング) する。ハッシュ値の異なるタプル官では結合される可能性はないことから、結合処理は分割された各クラスタ毎に行えばよく、その比較処理負荷は大幅に現象することになる。また、クラスタの処理は互いに独立なため、それぞれの処理を並列に実行することが可能であり並列処理に適したアルゴリズムといえる。本アルゴリズムは、ソフトウェアだけによる実装においても優れた性能を示すことは今までにも述べたが、機能ディスクシステムでは更にその高速化を図るため、その一部のハードウェア化を試みた。すなわち、ハッシュ並びにクラスタ化の操作はソフトウェアによって実装した場合には比較的時間がかかることから、機能ディスクシステムではハッシュ回路並びにステージングバッファメモリ上でのクラスタを管理するためのポインタ操作回路をハードウェアかで実現し、ディスクからのデータ流に完全に追従して動的クラスタリング操作を実

現することとした。後に述べるようにマイクロプログラムの制御による専用ディスクコントローラを開発した。さらに生成されたクラスタを並列処理することで一層の高速化を実現するために複数台のマイクロプロセッサを用いることにした。

3.3.2 関係データベースと OS の不整合性という問題点と機能ディスクシステムにおけるアプローチ

DB2 や Ingres など今日の多くの商用関係データベースシステムは既存のオペレーティングシステムの上に構築されており、その提供するバッファ管理ルーチンおよび入出力ルーチンに依存している。 Stonebraker らは、 Ingres 開発の経緯を踏まえて両者の不整合性を大きな問題として取り上げ [108]、その改良の必要を指摘している。通常のオペレーティングシステムは LRU アルゴリズムをそのバッファ置換方式として採用しているが、これは必ずしもデータベース処理に適当ではなく、むしろ最悪のアルゴリズムとなることが最近の研究から明らかになってきている [21]。これは、データベース処理においては、データページの参照に関し処理アルゴリズムに特有の参照パターンが生成され、いわゆるローカリティが見出せないこあとによる。例えば、 LRU 方式を改めページ置換アルゴリズムを工夫することでバッファヒット率が 10-15% 低減できたという報告がある [52]。また問合せ並びに対象リレーションに関する情報を利用する DBMIN アルゴリズムを用いるとシミュレーション評価では既存のバッファ管理方式に比べ、二倍程度のスループットの改善が期待できるとしている [21]。また、入出力オーバヘッドの問題に関しては Ingres の詳細な動作解析によれば、 I/O オーバヘッドが著しく大きくその 60% は、真に必要なデータアクセス以外のものによるとの報告ががある [39]。実際、我々はメインフレーム上の国産関係データベースシステムを利用した結合演算に関し、発行 I/O 数を計測してみた (I/O 回数は、モニタルーチンを稼働することにより、 SVC 呼出として計測することが可能であった)。対象リレーションは、タプル長 186 バイト、タプル数 1,000, 2,000, 5,000, 10,000 (ウィスコンシンベンチマークに準ずる) とし、結合度は 1.0 とする。表 3.1にその結果を示す。この表から I/O 回数はリレーションの入力と結果出力に最低限必要とされる最小必要 I/O 回数と比べて大幅に多いことがわかる。更に、別の問題として現状の入出力ドライバの低速性が指摘されている。すなわち、その肥大化した機能性の結果、一レコードのアクセスにも 5,000 以上の命令サイクルが必要とされており、ディスクのデータ転送スピードには全く追随できず、このような非効率的な入出力ルーチンによって大容量のリレーションに対し 1 ページずつアクセスするため、その性能は著しく低くなってしまう。

機能ディスクシステムは以上の問題に対し、先ず、既存のオペレーティングシステムに依存することなく、基本アルゴリズムとして採用した動的クラスタリング機能に最も適したバッファ管理アルゴリズムを独自に開発することとした。詳細は [135, 87] にゆずるが、動的クラスタリングアルゴリズムではその基本処理単位がクラスタになることから、クラスタを単位としたページ置換方式を新たに開発した。本手法では多数のクラスタが部分的にメモリ上に格納されるのではなく、いくつかのクラスタに対しその全体がメモリ上に展開されるように動的に制御する。このバッファ管理手法により、処理はクラスタ内のデータに集中して行われるため、余分なページの読み書きが一切生じることなく

リレーションの大きさ (タプル数)	I/O 命令発行回数	最小必要 I/O 命令発行回数
1000 件	1040 回	102 回
2000 件	2198 回	204 回
5000 件	6084 回	510 回
10000 件	13445 回	1020 回

表 3.1: 汎用計算機上における結合演算の I/O 命令発行回数

データ容量	ランダム読み出し	シーケンシャル読み出し
1.0MB	7.68 sec	0.38 sec
1.0GB	2.18 hours	5.68 min
1.0TB	93 days	4 days

表 3.2: ディスクに対する読み出し方式と応答時間

効率良い関係データベース処理を実行することができる。また、このように処理アルゴリズムを直接反映した形で入出力制御を行うため、従来のようにオペレーティングシステムの介在による数十%にもおよぶと方向されている不要な入出力が発生することもなくなると考えられる。

一方、入出力ドライバの低速性に関しては、まず、ディスク自身の性能ついて考えて見よう。ここでは、機能ディスクシステム開発時に標準的であった８インチ固定ディスク上の一定容量のデータをページ単位で読み出す場合の所要時間と連続転送により読み出す場合のそれとを比較したものを表3.2に示す。このように、一回一回のページアクセスによるデータの読み出しは著しく低速であり、なるべく連続して読み出すことが処理の高速化に極めて重要であることがわかる。しかしながら、ページごとの処理負荷は数千命令におよび、通常のソフトウェアだけではその性能向上には限界があるとかんがえられることから、機能ディスクシステムではディスクコントローラ上でページ (物理ブロック) からレコードの切り出しを行い、バッファ上にレコード単位で格納する高機能な DMA ハードウェアを設けることにした。これにより、ディスク上の連続ページからなるエクステントデータは途切れることなく転送することができる。また、機能ディスクシステムではより一層の高速化を行うために選択演算処理に関してデータ流に追随して行う (いわゆるオンザフライ処理) 機構ならびに必要な属性のみを切り出す機構もハードウェアとして実現することとしたが、この機能自体は従来の RAP や CAFS[13] などのマシンにもみられるものである。

3.4 本システムの構成

前節まで、関係データベースシステムの抱える本質的な問題点ならびに性能向上のための方策について述べた。この指針に従い、機能ディスクシステムでは、ハードウェア的にはディスクからのデー

タ流に追随してハッシュ操作、レコードの切り出し、選択演算を行う専用ディスクコントローラとクラスタ単位の記憶管理をする専用 DMA 回路並びにクラスタの並列処理を実現するための多重プロセッサ機構を、ソフトウェア的にはデータベース処理に最適化された入出力環境を提供することとした。また、ホストとのインタフェースには関係代数木を採用し、データ処理のみを高速に実行することを目的とし、問合せパーザ、同時実行スケジューラ等はホスト上で管理するものとする。従ってデータベース管理システムを全体をホストから切り離すことを目的とした IDM や DBC/1012 の様なバックエンドマシンとは異なる。つまり、従来の二次記憶システムでは単なるデータブロックの読み書きをそのインタフェースとしていたのに対し、ここではデータベース演算処理機構をディスク側に導入しその高機能化により大幅な性能向上を目指すことから機能ディスクシステムと名付けることとした。なお、以下に述べるハードウェア、ソフトウェア構成からも明らかなように従来の RAP, CASSM, CAFS 等の単純なサーチハードウェアのみをディスクコントローラに付加した古典的なオンザフライサーチディスクとはその実現する機能レベルが大きく異なる。

3.4.1 機能ディスクシステムの構成

全体のシステム構成

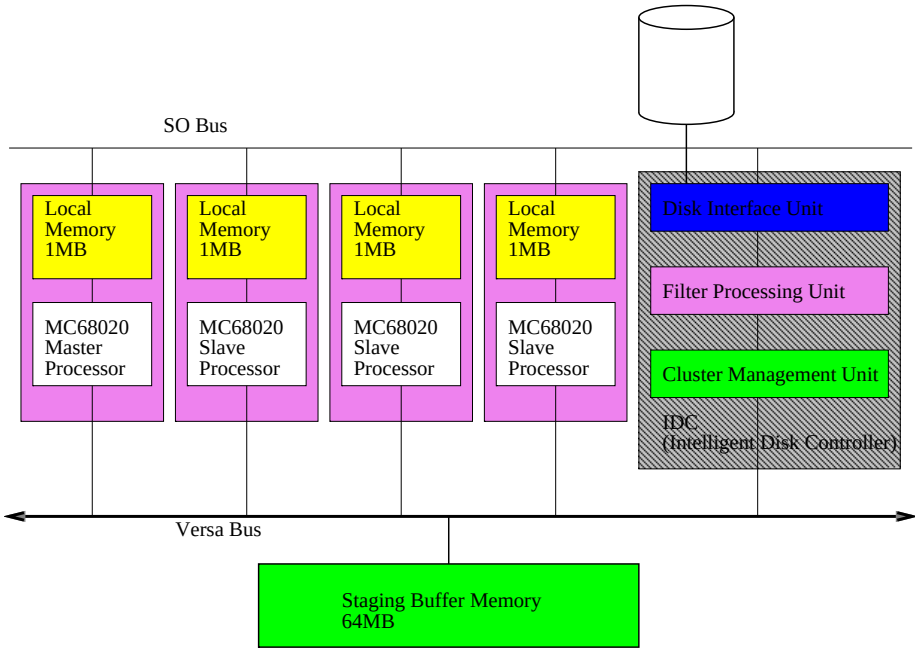

図 3.1: 機能ディスクシステムのハードウエア構成

機能ディスクシステム試作機は，図 3.1に示すように４台のプロセッサ，６ＭＢのステージングバッファメモリ，１台の８インチ・ディスクとＩＤＣ（Intellig ent Disk Controller) と呼ばれる高機能ディスクコントローラから構成されている。プロセッサは，モトローラ社の MC68020 マイクロプロセッサ (16MHz) を使用している。各プロセッサボード上には１ＭＢのローカルメモリがある。こ

れらのプロセッサボード, ステージングバッファメモリ並びにＩＤＣは，モトローラの標準バス Versa バスで結合されており，試作機はバス結合型共有メモリ並列計算機となっている。このプロセッサの内，１台をマスタプロセッサとしてシステムの開発と制御を行っている。また，バスのスループットを上げるために，各プロセッサとＩＤＣは， Versa バス以外にＳＯバスと呼ばれるディスクへのデータ書込み専用バスでも結合されている。システム構築を容, 易とするために既存のボードをなるべく多用することとし，プロセッサボードとメモリボードは市販ボードを利用しているが，ＩＤＣは機能ディスクシステムのために新規に開発されたものである。

インテリジェントディスクコントローラ (IDC) の構成

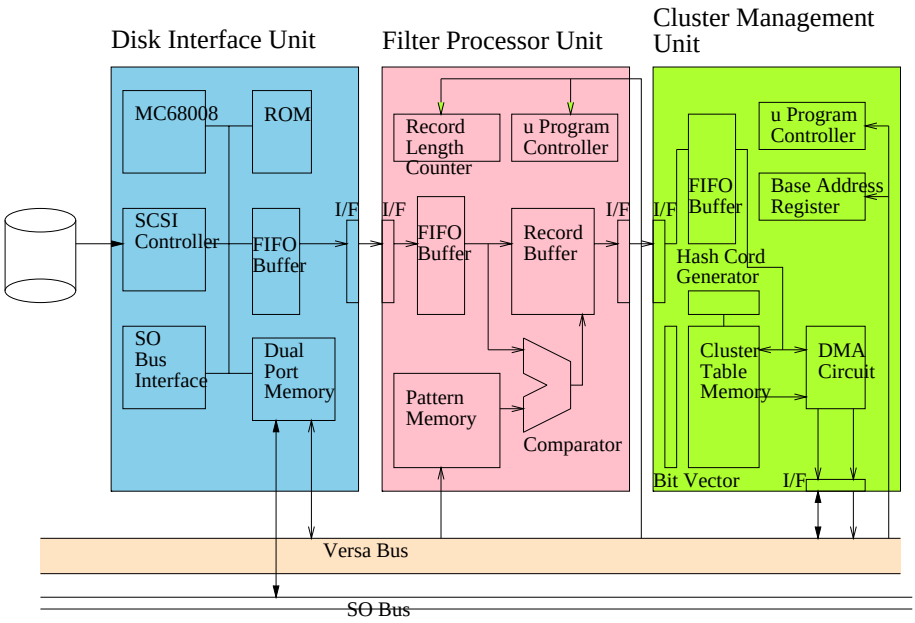

図 3.2: IDC の構成

IDC はディスクインタフェース部, フィルタプロセッサ部, クラスタ管理部のそれぞれ約２００石程度の IC からなる三枚のボードから構成され，ディスクの制御以外に関係データベース演算のハードウェアフィルタの機能，また動的クラスタリング機能も果している。図 3.2にそのブロックダイアグラムを示す。ディスクインタフェース部は、内部１６ビット外部８ビットのマイクロプロセッサ MC68008 を搭載し、 SCSI インタフェースを通し、ディスクの制御を行う。 MC68020 は本ボード上のデュアルポート上にコマンドを書き込むことにより入出力動作を駆動する。フィルタプロセッサ部では FIFO メモリとレコード長カウンタを有し、ディスクからの入力データ流に対しレコードの切り出しを行う。また、 4K バイトの照合メモリを有し、入力データとの比較をオンザフライに行い、選択演算を実行する。各アトリビュート毎の判定結果をに対する論理演算はマイクロプログラムによって実行され、照合メモリは MC68020 から Versa バスを介して設定される。クラスタ管理部は機能ディスクシステムにおいて最も特徴的なハードウェア部位といえ、クラスタテーブルメモリ、 FIFO バッファ、ハッ

シュコード生成器、 DMA 制御回路およびマイクロプログラム制御部からなる。

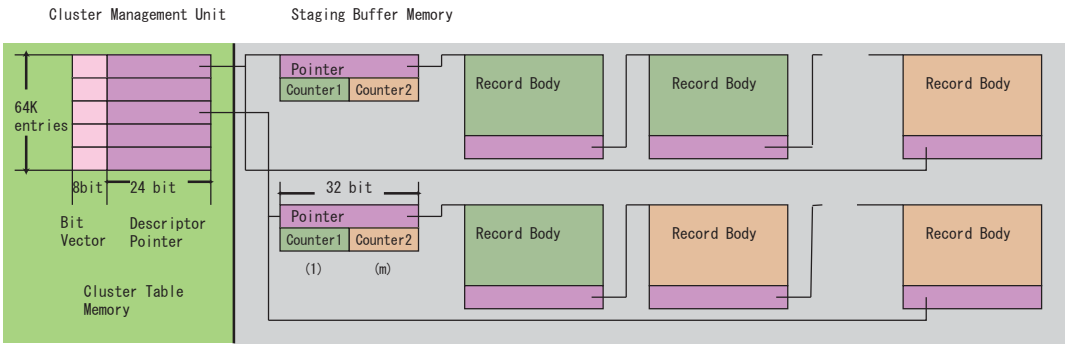

図 3.3: IDC クラスタテーブルの構成

クラスタテーブルメモリは図 3.3に示されるように１エントリ４バイト、６４Ｋエントリあり、各エントリは３バイトのアドレスポインタと１バイト (８本) のビットベクトルから構成される。フィルタプロセッサ部において選択されたレコードは、まずハッシュコード生成器に入り、１６ビットのハッシュコードが生成される。次にこの値をアドレスとし、クラスタテーブルメモリをアクセスし、４バイトの当該エントリを読み出す。この中で３バイトのポインタにより示されるステージングバッファ上のアドレスはクラスタ化されたレコード群をまとめているデスクリプタを指す。デスクリプタはレコード本体への本体へのポインタと二つのカウンタフィールドを持つ。このカウンタは結合演算をする場合、二つのリレーションからそれぞれのいくつのタプルが当該リストに接続されているかの数を示す。このようにディスクからのデータ流はデータインタフェース部、フィルタプロセッサ部を経た後クラスタ管理部に導かれ、結合属性、あるいは射影属性に対しハッシュが施され、その値によってクラスタ化されることになる。ステージングバッファ上へのレコードの格納、各デスクリプタに対するタプルの接続に伴うポインタ操作並びにカウンタ値のインクリメント等は全てクラスタ管理部の DMA 制御部並びにマイクロプログラム制御部によって実現されている。また、クラスタテーブルメモリは、クラスタ化動作と並行してジョイナビリティフィルタの役割も同時に果たしている。すなわち、結合演算では結合対象リレーション (今二つとする) それぞれに対し、１本のビットベクトルを割り付ける。第一のリレーションの読みだし時に各クラスタにはじめてタプルが割り当てられる毎に対応するビットをビットベクトル上でセットする。読み出し終了時点で本ビットがセットされていなかったエントリは１タプルも分配されなかったことを意味し、第２のリレーションの読み込み時にはそのようなエントリに対し、分配されるタプルはもはや結合される可能性はないため、棄却することができる。つまり、ビットベクトルに必要とされる容量はそれほど大きくないため、クラスタ用のエントリに比べてより大きく設定することも可能であるが、ここでは、簡単のためにクラスタ数と同じ大きさのビットベクトルを用意することとした。また、このビットベクトルの操作もマイクロプログラム制御により実現されている。

多重プロセッサ部の構成

クラスタテーブルメモリにより最大 64K 個にクラスタ化されたリレーションはそれぞれのクラスタが複数台のプロセッサによって並列に処理される。多重プロセッサの方式とし、今回の試作機では実装を簡素化するため、最も単純な共有メモリバス結合型マルチプロセッサアーキテクチャを採用することとした。バスも特に高速化を試みず市販ボードの利用を考え、 Versa バスを選択した。従ってバスに結合可能なプロセッサも限られ、今回は４台とした。

3.4.2 機能ディスクシステムの基本動作

機能ディスクシステムの基本動作として、本システムの心臓部である IDC を通してのデータの流れとマルチプロセッサによる並列処理方式について述べる。図 3.4に基本動作におけるデータの流れを示す。データの流れはステージングステップとプロセッシングステップに分かれている。

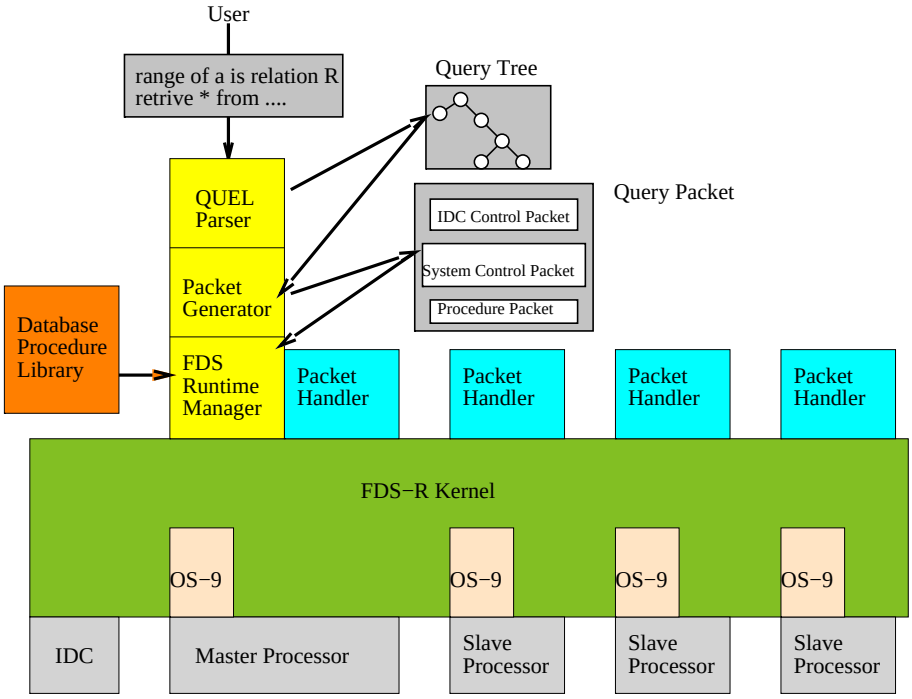

図 3.4: 機能ディスクシステムにおける処理の流れ

ステージングステップ　このステップでは，データがディスクから読み出されＩＤＣを通してステージングバッファメモリに書き込まれるまでを言う。ＩＤＣでは，オンザフライに各タプルをディスクから取り出しフィルタリング処理を行った後，ステージングバッファメモリ上で動的にクラスタリングされる。同時にハッシュテーブルが生成され，ビットマップフィルタリングも行われる。

プロセッシングステップ ステージングバッファメモリ上のクラスタ化されたデータ (バケットと呼ぶ) は，ハッシュテーブルに従ってステージングバッファメモリから各プロセッサ内のローカルメモリへ並列に取り込まれ，各プロセッサ上で処理された後，処理結果はＩＤＣを通してディスクに書き戻される。

このようにステージングバッファメモリ上に収まる大きさのデータに対して，機能ディスクシステムではバケットのサイズを小さくすることでマルチプロセッサによる並列処理が効率良く行われるようにしており，これをファインハッシュ方式と呼ぶ。この方式とバケットの大きさによる性能の変化，並列処理効果等については [58] を参照されたい。

3.5 関係データベース処理方式の実装

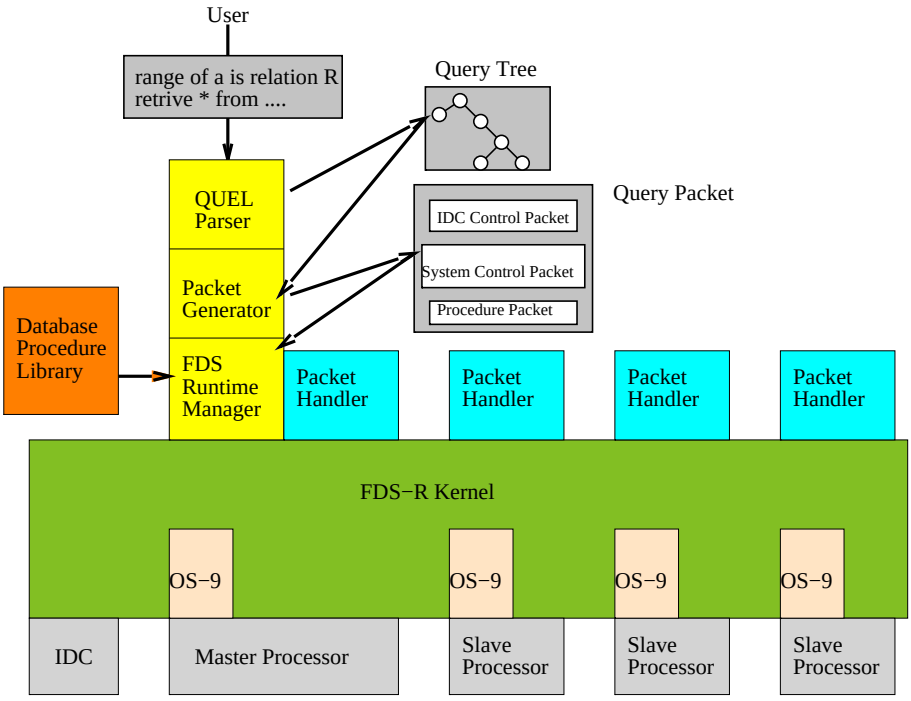

図 3.5: 機能ディスクシステムのソフトウェア構成

機能ディスクシステムでは関係データベース処理システムとして関係データベースシステム INGRES で用いられている QUEL 言語のサブセットを実現しており、ユーザから指定された問合せをいくつかの単純な関係代数演算（選択演算，射影演算，結合演算，集計演算）に展開し，これらの演算を順次処理することにより問合せの実行を実現している。機能ディスクシステムではソフトウェアで処理すれば時間のかかる動的クラスタリング機能をハードウェアとして実装しているため，一層の処理性能の向上が期待できるが，反面，ハードウェアで提供している性能を活かすための演算処理方式を十分考慮しなくてはならない。従って、カーネルレベルで大量のデータに対するアクセスサポート、ハッシュに基づく関係データベース処理方式の支援を実現すると共に、機能ディスクシステムが

提供する高いレベルでのインタフェースに合わせて問い合わせ処理を行ない、単体のデータベース演算にはハッシュに基づく処理方式（GN ハッシュ方式）を採用することで機能ディスクシステムの性能を引き出している。

3.5.1 機能ディスクシステムの関係データベース処理

機能ディスクの関係データベース処理システムを図に示す。ユーザインタフェース (QUEL Parser)、問い合わせ解析システム (Packet Generator)、インタプリタ (FDS Runtime Manager) が前述の機能ディスクカーネルの上に構築されている。各モジュール間のインタフェースを図に灰色のボックスで示す。ユーザはインタラクティブ、あるいは C 言語のライブラリとしてデータベース処理を QUEL 言語を用いて記述することができる。問い合わせ解析システムではユーザインタフェース部で取り出されたデータベース処理記述部を解釈し、問い合わせ木 (Query Tree) として生成する。この問い合わせ解析システムでは、機能ディスクシステムが提供する高い演算処理機能をそれぞれのリーフが対応するように解析を行ない (Query Packet)、インタプリタ部に渡す。インタプリタ部は指定された順に各パケットの処理を行なう。各々のリーフで指定される演算処理の内、特定の定型処理（集計演算、重複除去、結合）についてはライブラリを選択、それ以外の場合には IDC へのパラメタ設定テーブルを生成し、カーネルの実行コマンドを呼び出す。

3.5.2 機能ディスクシステムカーネル

システムコール	説明
read	IDC を通じてのディスク読み出し
write	IDC を通じてのディスクの書き戻し
setIDCinit	IDC の初期化
setIDCfilter	IDC へのフィルタリングパラメタ設定
setIDChash	IDC への動的クラスタリングパラメタ設定
exec	並列処理の実行および共有メモリ上のバッファ環境
send	各プロセッサ間でのデータ送信
receive	各プロセッサ間でのデータの受信
barrier	プロセッサ間の処理の同期
comminit	プロセッサ間のコミュニケーションの初期化

表 3.3: 機能ディスクカーネルのシステムコール

機能ディスクシステムの特徴はハッシュを用いた動的クラスタリングアルゴリズムを支援するためのハードウェアが構築されており，ディスクからのデータ転送速度に追従してステージングバッファ

メモリ上にクラスタを展開することである。そこで，関係データベースシステムのカーネルも通常の汎用 OS を利用した場合のように物理的なページ等にとらわれないクラスタレベルでのバッファ管理，入出力ドライバが構築されている。つまり， IDC を用いたオンザフライ処理を効果的に用いるために, あらかじめステージングバッファメモリを固定的に物理ページ単位で利用するのではなく，ステージングバッファメモリ全体をクラスタに対するフリー領域として利用することで、一回のシステムコールでタプル単位でのデータアクセスを実現すると共にハッシュに基づく関係代数演算アルゴリズムに対する支援を実現している。また、複数プロセッサによるデータベース演算の並列処理も支援しており、指定された台数による並列処理と排他制御、主記憶の分割使用などを実現している。以下に、機能ディスクシステムカーネルに用意されているシステムコールを表 3.3に示す。

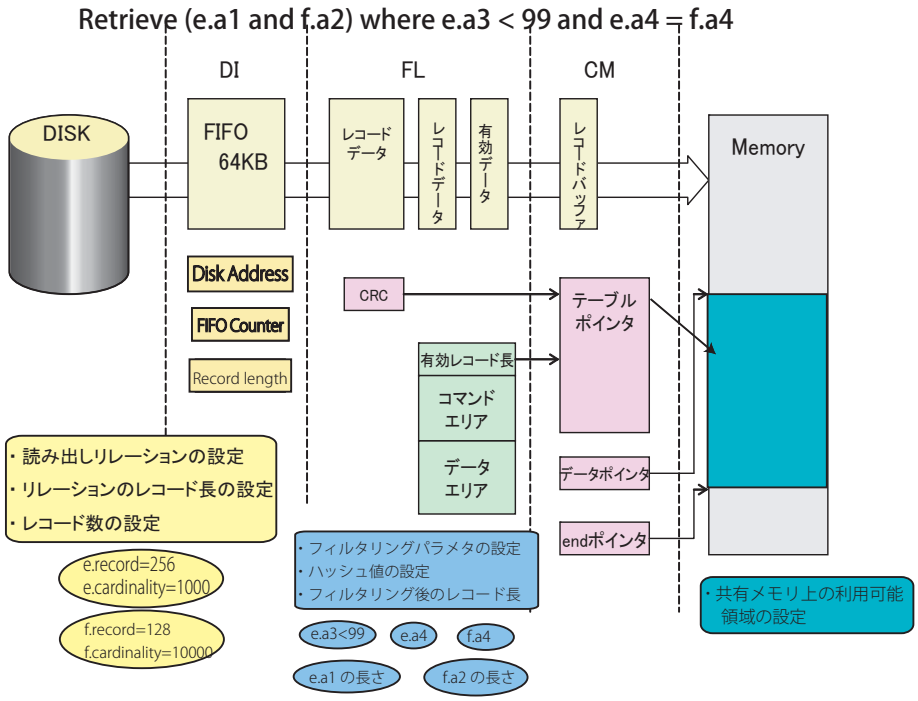

図 3.6: IDC のインタフェース

3.5.3 IDC インタフェース

IDC が提供するインタフェースを図 3.6にしめす。大きくはオンザフライで実行する選択情報の設定と動的クラスタリング機構に設定するハッシュ情報にわかれる。タプル選択情報としては and または or による複数の条件が設定可能であり、例えば、以下のような問い合わせに対しては図 3.6のように設定される。

range of e is relation R

range of f is relation S

retrieve (e.a1 and f.a2) where e.a3 <99 and e.a4 = f.a4

左側のディスクインタフェース (DI) 部に対しては、リレーション R とリレーション S のデータベース情報から、ディスク上のリレーションの位置、タプル長、リレーションのタプル数が設定さえる。上記の例では、リレーション R、 S の情報として、タプル長 256 バイト、 1,000 件、タプル長 128 バイト、 10,000 件が設定されている。中央のフィルタリング (FL) 部に対しては、条件節の条件、属性の位置、属性の長さ等が設定される。上記の例では、 e.a3 に対して「99 より小さい」の条件、次のクラスタリング制御 (CM) 部で利用するための結合演算属性 e.a4, f.a4 を抜き出すための指定、ハッシュ値として CRC の設定、ハッシュフィールドに e.a4 と f.a4 を利用するための指定、さらに結果で利用する e.a1, f.a2 の長さと抜き出すための指定が設定される。右側のクラスタリング制御 (CM) 部に対しては、ハッシュテーブルの先のデータエリアの開始アドレスを設定する。これらのパラメタ設定は実行部において、インタプリタがシステムコールの引数として渡したパケットを設定することで行なわれる。

3.6 基本性能評価

機能ディスクシステム上に実装した関係データベースシステムの性能評価を行なうために、結合演算、集計演算、射影演算を用い、パラメタを変えて実行した。以下に述べる測定結果は，つぎのシステム構成のもとで計測された。プロセッサは， MC68020 ４台（ローカル・メモリ 1MB)，ステージング・バッファ 512MB ， ８インチディスク（NEC D2257) １台を用いた。プロセッサ台数は４台である。また，測定用のデータベースのレコード長は１２８バイト固定であり、リレーションのタプル数はパラメタとして測定毎に変化する。

3.6.1 結合演算

今回の測定では，結合演算を試作機上に実装されている問合せ解析実行システムを用いてＱＵＥＬステートメントとして実行している。性能測定に用いた問合せは以下のとおりである。

range of e is relation1
range of f is relation2
retrieve into tmp (e.all,f.all) where e.a1 = f.a 1 and e.a1 <= α

ここで用いる２つのリレーション・サイズは等しい。また，フィルタリング・ファクタはαによって指定する。この条件によるタプルのフィルタリングおよびフィールド a1 によるクラスタリングは，IDC で実行される。フィールド a1 は４バイトの整数でユニークな乱数であり，この問合せが実行されるとαに応じたタプル数の結果リレーションが生成される。

(1) リレーションサイズによる処理時間の変化

図 3.7にフィルタリング・ファクタ（α）が 1.0 の場合のネストループ方式と Grace ハッシュ方式及びＧＮハッシュ方式の処理時間を示す。ステージングバッファは 512KB, 結合処理時のバケット

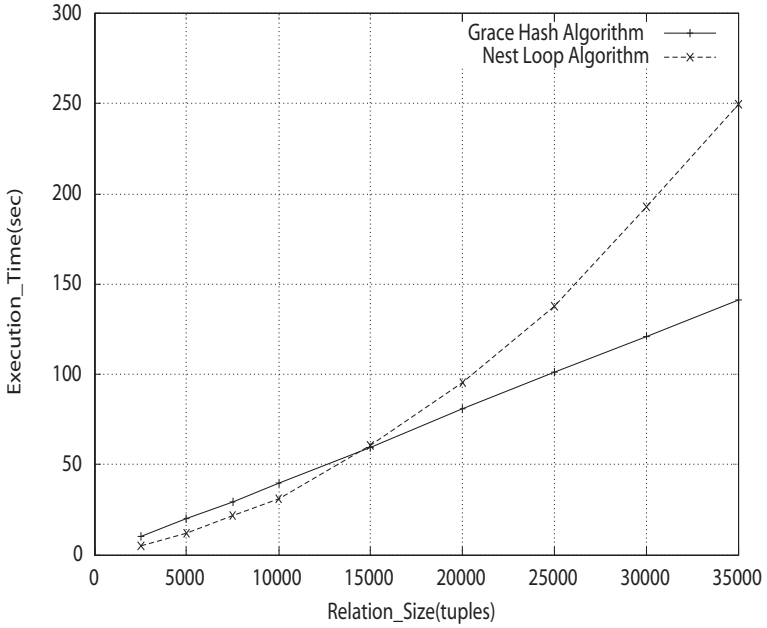

図 3.7: リレーションを変化させた場合の結合演算

サイズは平均 10 タプルに設定されている。図 3.7からわかるようにリレーションが小さい場合はネストループ方式の方が処理時間が少ないが，ステージングバッファサイズの４倍程度のリレーションからは Grace ハッシュ方式が小さくなり，前章の入出力コスト評価式で導き出された結果と同じてある。これは，ネストループ方式におけるデータの読出し時間及びデータ処理時間が二重ループの数に従って増加することによる。全体の処理時間についてみると，ネストループ方式の処理コストがリレーションのサイズが大きくなるにつれて二重ループ数に従って増加するが， Grace ハッシュ方式ではほぼリレーションサイズに比例している。このように，ＧＮハッシュ方式は Grace ハッシュ方式とネストループ方式の利点を取り入れることで高い性能が得られていることが確認できる。

(2) ステージング・バッファサイズによる処理時間の変化

図 3.8にステージング・バッファサイズを変化させた場合のネストループ方式と Grace ハッシュ方式及びＧＮハッシュ方式の処理時間を示す。リレーションサイズは，１００００件であり，図 3.7と同様に，フィルタリング・ファクタ１．０，ステージングバッファは 51 2KB, 結合処理時のバケットサイズは平均 10 タプルに設定されている。この図からわかるように，ネストループ方式におけるリレーションＲの分割数が５より小さい場合には，ネストループ方式の方が性能がよく，それを越えると Grace ハッシュ方式の処理性能が良いことが確認されている。また， Grace ハッシュ方式の性能は，メモリのサイズに係わらずほぼ一定であり，ネストループ方式の処理コストがメモリが小さくなるにつれて急激に増加しているのとは，対照的である。また，ＧＮハッシュ方式で用いている入出力コスト評価式が有効であることが確認できる。

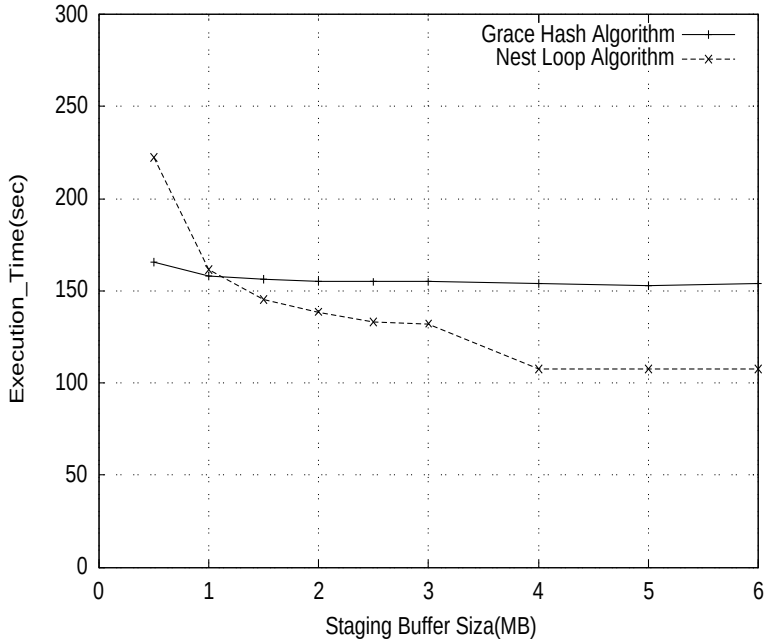

図 3.8: ステージング・バッファサイズを変化させた場合の結合演算

3.6.2 射影演算

射影演算におけるネストループ方式と Grace ハッシュ方式の違いを明らかにするために，それぞれのアルゴリズム毎にリレーション・サイズ，ステージング・バッファ・サイズを変化させて実行時間の計測を行った。性能測定に用いた問合せは以下のとおりである。

range of e is relation1

retrieve unique into tmp (e.a1,e,a2)

フィールド a1 は４バイトの整数であり，フィールド a2 は 28 バイトの文字列である。今回の射影演算では，重複無しのデータを用いた。

(1) リレーションサイズによる処理時間の変化

図 3.9にリレーション・サイズを変化させた場合のネストループ方式と Grace ハッシュ方式の処理時間を示す。ステージングバッファサイズは 384KB ，クラスタ・サイズは平均４タプル，データの重複無しに設定されている。図 3.9からわかるように， Grace ハッシュ方式ではリレーションサイズとほぼ比例して処理時間が増えているが，ネストループ方式ではリレーションサイズの二乗で処理時間が増加している。従ってリレーションサイズが大きい場合の Grace ハッシュ方式の有効性がわかる。また，ステージング・バッファ・サイズを越えたリレーションでも，前述の入出力コスト式から導かれるようにステージング・バッファ・サイズの２倍程度のリレーションまでは，ネストループ方式の処理時間が Grace ハッシュ方式よりも小さい。

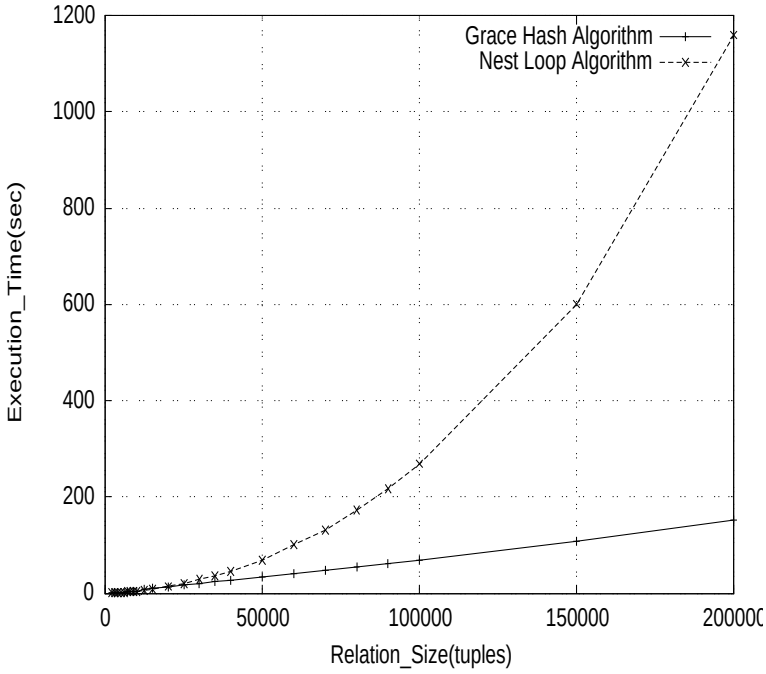

図 3.9: リレーションサイズを変化させた場合の射影演算

(2) ステージングバッファサイズによる処理時間の変化

図 3.10にステージングバッファ・サイズを変化させた場合のネストループ方式と Grace ハッシュ方式の処理時間を示す。リレーション・サイズは 100000 タプル，クラスタ・サイズは平均２タプル，データの重複無しに設定されている。図 3.10からわかるようにネストループ方式の場合には，ステージングバッファが小さくなるにつれて急速に処理時間が増加する。また，前述の結合演算におけるネストループ方式の処理時間の変化と比較し，射影演算では１リレーションの処理であるため，ステージングバッファ・サイズが小さくなるにつれて内側ループで読み込むデータ量が増えるため，処理時間の変化は大きい。一方， Grace ハッシュ方式では，リレーションがステージング・バッファより大きい場合でもステージングバッファサイズによらず，処理時間はほぼ一定である。ステージング・バッファが非常に小さい場合には，入出力クラスタを生成するためのオーバーヘッドが増えるため，処理時間が少し増加する。

3.6.3 集計演算

性能測定に用いた問合せは以下のとおりである。

range of e is relation

retrieve (e.a1 , sum = sum(e.a2 by e.a1))

射影演算と同様にリレーションサイズ ，ステージングバッファサイズによる処理時間の変化について測定した。また，フィールド a1 および a2 は４バイト整数である。フィールド a1 はパーティション・サイズ分の重複があり，このフィールドを用いて IDC がクラスタリングを行う。

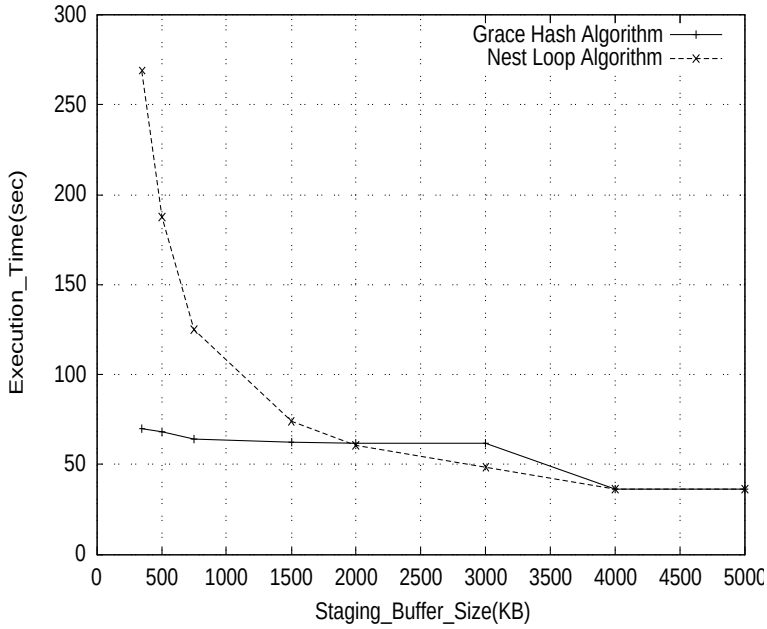

図 3.10: ステージングバッファサイズを変化させた場合の射影演算

(1) リレーション・サイズによる処理時間の変化

図 3.11にリレーション・サイズを変化させた場合のネストループ方式と Grace ハッシュ方式の処理時間を示す。ステージングバッファサイズは 64KB，クラスタ・サイズは平均 10 タプル，パーティション数はリレーションサイズの 0.1 に設定されている。図 3.11からわかるように，射影演算同様， Grace ハッシュ方式ではリレーションサイズとほぼ比例して処理時間が増えているが，ネストループ方式ではリレーションサイズの二乗で処理時間が増加している。また，射影演算と比較してネストループ方式と Grace ハッシュ方式の処理時間の逆転がほとんどないことが観察される。これは，射影演算の結果タプルが 32 バイトであるのに対して，集計演算の結果タプルが 8 バイトと小さく，フィルタリングファクタが非常に小さいためで，前述の入出力コストの比較式から得られる結果と同じである。

(2) メモリ・サイズによる処理時間の変化

図 3.12にステージングバッファ・サイズを変化させた場合のネストループ方式と Grace ハッシュ方式の処理時間を示す。リレーション・サイズは 20000 タプル，クラスタ・サイズは平均 10 タプル，パーティション数はリレーションサイズの 0.1 に設定されている。図 3.12からわかるようにネストループ方式の場合には，ステージングバッファが小さくなるにつれて急速に処理時間が増加する。射影演算におけるネストループ方式の処理時間の変化と比較し，フィルタリング・ファクタが非常に小さいため，ステージングバッファ・サイズが小さくなるにつれて内側ループで読み込むデータ量が増え，処理時間の変化は大きい。一方，射影演算と同様に Grace ハッシュ方式では，リレーションがス

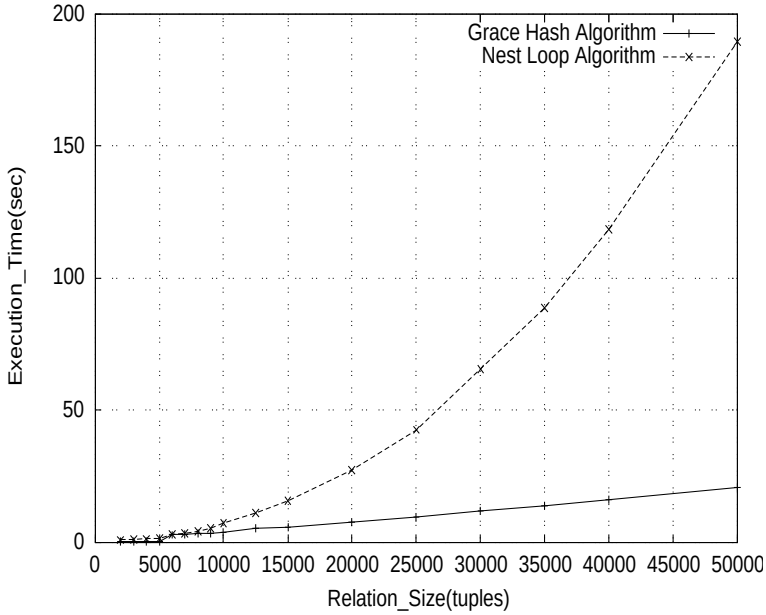

図 3.11: リレーションサイズを変化させた場合の集計演算

テージング・バッファより大きい場合でもステージングバッファサイズによらず，処理時間はほぼ一定である。ステージング・バッファが非常に小さい場合には，入出力クラスタを生成するためのオーバーヘッドが増えるため，処理時間が増加する。

また，入出力コスト式から導かれるように，ネストループ方式と Grace ハッシュ方式の処理時間の逆転がないことが，図 3.12からも確認できる。

3.7 オリジナルウィスコンシンベンチマークを用いたの性能評価

前節では、各基本演算とシステム資源に対する機能ディスクシステムの基本性能を示したが、本節では機能ディスクシステムの有効性を明らかにすべく、広く利用されている米国ウィスコンシン大学で開発された関係データベース用ベンチマーク、ウィスコンシンベンチマークを用いて評価を行った。１９８４年に開発されたオリジナルウィスコンシンベンチマーク [6] は当時の商用関係データベースシステムの性能比較が行われた。

本性能評価に用いた機能ディスクのシステム構成は、８インチディスク１台 (アクセスタイム 28msec, 転送速度 1.2MB/sec)、プロセッサ４台、ステージングバッファメモリが６ M バイトであり、ベンチマークには４つの代表的な関係代数演算 (選択演算、射影演算、結合演算、集計演算) を用いた。表 3.4 - 表 3.7に４つの演算それぞれに対する評価結果を示す。表にはウィスコンシン大学で行われた６つのシステム、即ち、 U-Ingres(大学で開発された Ingres)、 C-Ingres(商用版 Ingres)、 Oracle、 IDMwithoutDAC(データベースアクセラレータ機構なし)、 IDMwithDAC(データベースアクセラレータ機構あり)、 DIRECT の評価値も併せて示す。ここで、 Ingres, Oracle は商用関係データベースであり、 VAX11/750 上での性能評価値が示されている。また、 IDM は前述のとごく、データベース

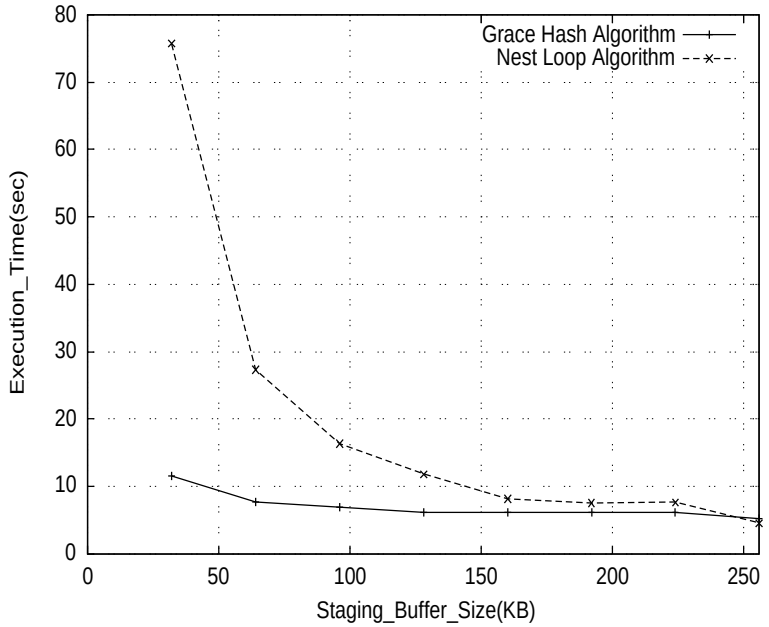

図 3.12: メモリを変化させた場合の集計演算

専用オペレーティングシステムを採用した商用データベースマシン、DIRECT はウィスコンシン大学で開発された 4 台の LSI-11 からなるマルチプロセッサデータベースマシンである。以下それぞれの演算について評価結果を考察する。

3.7.1 選択演算

range of e is tenKtuples
retrieve into tmp (e.all) where e.a0 < 1000

QUEL 言語では以上のように表される単純な選択演算処理ではタプル数 10,000 件のリレーションから e.a0 < 1000 を満たす 1000 件のタプルを取り出し結果をディスクに書き戻す。ここで h、ベンチマークリレーションはインデクスはなく、１タプル１８２バイト、a0 は 2 バイトの整数フィールドで 0 〜 9999 の値がランダムに配置されているとする。表 3.4から分かるように、本選択演算ではリレーションの全スキャンが必要となるが、従来のソフトウェアではページ単位の非効率的な入出力が繰り返されるのに対し、機能ディスクシステムでは専用ディスクコントローラにより連続的なディスクからの読出時間並びにディスクへの結果書き込み時間だけで終了することができ、大きな性能向上が得られる。

3.7.2 射影演算 (重複除去)

range of e is oneKtuples
retrieve unique tmp (e.all)

選択演算 (選択率 １０%)	
U-Ingres	64.4 sec
C-Ingres	53.9 sec
ORACLE	230.6 sec
IDMwithoutDAC	33.4 sec
IDMwithDAS	23.6 sec
DIRECT	46.0 sec
国産関係データベースシステム	10.5 sec
機能ディスクシステム	3.4 sec

表 3.4: ウィスコンシンベンチマーク選択演算評価結果

本ベンチマークは 1000 タプルからなるリレーションを対象とし、重複したタプルの除去を行った後、処理結果をディスクに書き戻す。実際は各タプルの a0 属性はユニークな値を有し、重複は存在しないため、結果リレーションは元のリレーションと同じ大きさになる。表 3.5から分かるように、本処理において機能ディスクシステムは他の最も高速なシステムの 50 倍程度と極めて高い性能を示している。これは、従来のシステムが重複除去に際しソートあるいは単純なネストループ処理を利用し、極めて多数の入出力が行われるのに対し、機能ディスクシステムでは動的クラスタリングアルゴリズムを用いることにより大幅に負荷を減らすことができるためと考えられる。

射影演算 (重複除去)	
U-Ingres	236.8 sec
C-Ingres	132.0 sec
ORACLE	199.8 sec
IDMwithoutDAC	122.2 sec
IDMwithDAS	68.1 sec
DIRECT	58.0 sec
国産関係データベースシステム	9.9 sec
機能ディスクシステム	1.5 sec

表 3.5: ウィスコンシンベンチマーク射影演算評価結果

3.7.3 結合演算

range of e is tenKtuples
range of f is tenKtuples
retrieve into tmp (e.all,f.all) where e.a0 = f.a 0 and e.a0 <= 1000

10,000 件タプルの二つのリレーションに対し a0 属性に関し結合を行う本結合演算処理では、一方のリレーションに関し、条件 a0<1000 が付加されており、 a0 属性は 0 〜 9999 までのユニークな値をとるため、結果リレーションも 1000 タプルとなり、また、結果タプルは 364 バイト (= 182 *2) になる。表 3.6に測定結果を示す。 Oracle, IDM, U-Ingres では単純なネストループを利用しているため、多大な時間を要し、実用に耐えない。 C-Ingres はソートマージアルゴリズムを用い、かなり性能が改善されているものの機能ディスクシステムに比べると２０倍以上低速である。機能ディスクシステムではまず、 a0<1000 の条件が IDC のフィルタプロセッサ部で処理され、本条件を満たす１０００タプル即ち 182K バイトがステージングバッファメモリにロードされる。次にもう一方のリレーションに関し、前述したビットベクトル処理により結合される可能性のない９０００タプルが除去され、有効な１０００タプルが読み込まれ、結合処理が並列に実行される。このように機能ディスクシステムでは GRACE ハッシュ法とビットベクトルによるふるい落とし処理によりディスクから一度リレーションを読み出すだけで結合処理を実行することができ、極めて高い性能を達成することができる。

結合演算 (結合率 １０%)	
U-Ingres	10.2 min
C-Ingres	1.8 min
ORACLE	> 300 min
IDMwithoutDAC	> 300 min
IDMwithDAS	> 300 min
DIRECT	10.2 min
国産関係データベースシステム	41.7 sec
機能ディスクシステム	5.9 sec

表 3.6: ウィスコンシンベンチマーク結合演算評価結果

3.7.4 集計演算

range of e is tenKtuples
retrieve (e.a2 , sum = sum(e.a1 by e.a2))

ここでは a2 は２バイトの整数フィールドで 0 〜 99 の値をランダムにとる。したがって、本集計

演算では 10,000 タプルからなるリレーションは a2 属性によって 100 個のグループに分けられ、各グループに関し、 a1 属性値の和がとられる。表 3.7から分かるように、集計演算においても従来のシステムでは単純なネストループ方式あるいはソートマージ方式が採用されているため、その処理には多大な時間が必要となるのに対し、機能ディスクシステムでは当該リレーションを a2 属性に関し、動的にクラスタリングすることとにより大変効率良く処理することが可能であり、他の演算同様大きな性能向上ができている。

集計演算 (１００パーティション)	
U-Ingres	174.2 sec
C-Ingres	484.8 sec
ORACLE	1487.5 sec
IDMwithoutDAC	67.5 sec
IDMwithDAS	38.2 sec
DIRECT	229.5 sec
国産関係データベースシステム	10.9 sec
機能ディスクシステム	2.5 sec

表 3.7: ウィスコンシンベンチマーク集計演算評価結果

3.8 拡張ウィスコンシンベンチマークによる性能評価

近年の半導体メモリの進歩に伴い、主記憶容量は大幅に大容量化されつつある。１９８４年のオリジナルウィスコンシンベンチマークではベンチマークリレーションは１タプル１８２バイト、 10,000 タプルからなり、その容量は 1.82M バイトであった。この値は、最近の計算機の主記憶容量と比べるとかなり小さいため、また、ユーザはより大規模なリレーションを構築するようになったことから、１９８７年に拡張ウィスコンシンベンチマークが開発された。 a0,a1 等の整数フィールドが２バイトから４バイトに、タプル長が１８２バイトから２０８バイトに変更され、タプル数は 100,000、 1,000,000 件へと拡張された。表４に拡張ウィスコンシンベンチマークによる結合演算の評価結果を示す。ウィスコンシン大学では射影演算や集計演算に関し未だ評価を行っておらず、結合演算のみの評価を行った。比較のためにウィスコンシン大学で開発中の Gamma データベースマシンおよび Teradata 社の DBC/1012 データベースマシンの性能を示す。 200M バイトに及ぶ大容量リレーションを実用的な時間内に処理できるシステムは現時点においても少なく、これらのシステムが選択された。本節の評価における機能ディスクシステムの構成は先と同様、 MC68020 が４台、８インチディスク１台からなる。 Gamma は１７台の VAX11/750 を 80Mbit/sec のトークンリングバスで接続した実験システムであり、１７台の内、８台がディスクを有し、それ以外はディスクレスとなっている。ディスクは通常の８インチ SMD ドライブである。 DBC/1012 は i80286 マイクロプロセッサ２０台からなり、

これらはＹネットと呼ばれる木構造のネットワークで結合されている。各プロセッサは８インチディスクを２台持ち、システム全体で４０台のディスクを有する。

3.8.1 選択演算

選択演算としては以下の問合せを用いて、性能評価を行った。結果はディスクに書き込まれる。

range of e is relation R
retrieve into tmp (e.all) where e.a0 < X

選択演算は最も基本的なシステム性能を示す問合せである。ここで、 where 節の条件 e.a0 < X を機能ディスクシステムでは IDC によりオンザフライに処理される。したがって、機能ディスクシステムにおいては、選択演算のコストはほぼ入出力コストと考えられ、読み込みおよび書き戻しのリレーションサイズに比例して処理時間が増加する。本性能評価においては、選択率 1%,10% に変えて性能評価を行い、その結果を表 3.8,3.9に示す。また、比較のために Gamma および Teradata の結果も併せて示す。表からわかるように、機能ディスクシステムの実行時間が Gamma の約２倍である。機能ディスクシステムがディスク１台であるのに対し、 Gamma がディスク８台の構成、 Teradata がディスク３２台構成であることを考えると、機能ディスクシステムが十分に高い性能を得られていることが確認できる。

Selection Query(1% selectivity)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
Gamma	1.6 sec	13.8 sec	134 sec
Teradata	6.9 sec	28.2 sec	213 sec
機能ディスクシステム	2.6 sec	24.5 sec	239 sec

表 3.8: 拡張ウィスコンシンベンチマークによる選択演算の性能評価結果 (1%)

Selection Query(10% selectivity)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
Gamma	2.1 sec	17.4 sec	182 sec
Teradata	16.0 sec	111.0 sec	1107 sec
機能ディスクシステム	3.5 sec	29.3 sec	285 sec

表 3.9: 拡張ウィスコンシンベンチマークによる選択演算の性能評価結果 (10%)

3.8.2 結合演算

Query1: JoinABprime with non-key attributes 結合演算として、以下に示す問合せ処理を行った。二つのリレーションの属性１ (attribute feild 1) の値が一致した場合に双方のタプルを合わせて、結果リレーションとして生成する。属性１の値はユニークランダムであり、リレーション R はリレーション S の１０倍となっている。したがって、結果リレーションはリレーション S と同じタプル数分だけ生成される。例えば、 10,000 件× 1,000 件の結合演算では、 1,000 件の結果タプルが生成される。

range of e is relation R

range of f is relation S

retrieve into tmp(e.all, f.all) where e.a1 = f.a1

表 3.10の結果から明らかなように機能ディスクシステムは Teradata よりはるかに高い性能を、また Gamma と同程度の性能を達成している。 Teradata や Gamma ではリレーションを複数台のディスクに分散させ、並列にデータを読み出すこととにより入出力スピードを大幅に向上させている。４０台のディスクを並列駆動し、２０台のプロセッサで並列処理する DBC/1012、８台のディスクを並列駆動し、１７台の VAX で並列処理する Gamma に比べ、ただ１台のディスクと４台のプロセッサから構成される機能ディスクシステムはそのシステム資源で正規化すると、極めて高い性能を達成していると言えよう。１００万タプルのリレーションは 208M バイトにおよび、オリジナルウィスコンシンベンチマークの場合のように a0 に関する条件節評価後の中間リレーションがステージングバッファに入り切るようなことはなく、ディスクに対する大量のデータの入出力が何度も生じることになる。このような入出力負荷の重い環境化において効率の良いバッファ管理と高速な入出力ドライバにより機能ディスクシステムは極めて高い性能を発揮することができている。同時にステージングバッファサイズを越えるデータに対しては機能ディスクシステムが関係代数演算アルゴリズムとして採用している GN ハッシュ方式 [57] が Gamma が用いている Simple ハッシュ方式と比較して高い性能をあげているといえる。

Join Query(1)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
FDS-R	3.9 sec	41.8 sec	997 sec
Teradata	34.9 sec	321.8 sec	3419 sec
Gamma	6.5 sec	46.5 sec	2938 sec

表 3.10: 拡張ウィスコンシンベンチマークによる結合演算の性能評価結果

上述のように、機能ディスクシステムは他のデータベースマシンを比較して高い性能を達成しているが、１００万件同士のリレーションでは、ハッシュビットマップ領域を越えているため、機能ディ

スクシステムおよび Gamma ではこのハッシュビット機構を利用することはできない。しかしながら、大規模データの処理に関しては、静的なデータベース情報を用い、さらに IDC の動的データクラスタリングおよびハッシュ関数機構を利用することで結合演算処理の性能を向上することが可能である。ここでは、リレーションサイズをもとに I/O クラスタのためのハッシュ関数を決定し、１００万件のデータがおさまるように上位ビットを利用する代わりに１０万件のリレーションがおさまる下位ビットを利用する。その結果、 I/O クラスタとして１０万件のリレーションに該当しない部分の不要なデータの読み込みおよび書き戻しを低減することで、性能が向上する。表 3.11に表 3.10に示した結果を同じ問合せを relation S の情報をもとにハッシュ関数を選んだ場合の結果を示す。表 3.11から分かるように、性能が倍程度、向上している。

Join Query(1)	
システム名	1,000,000 tuples × 100,000 tuples
Ordinary Hash	907 sec
Appropriate Hash	470 sec

表 3.11: ハッシュ関数による性能評価

Query2: JoinAselB with non-key attributes 以下に示す where 節にて条件 e.a1 < X が指定されている問合せを用いて、性能評価を行った。結果はディスクに書き込まれる。

range of e is relation R
range of f is relation S
retrieve into t (e.all, f.all) where e.a1 = f.a1 and e.a1 < X

ここで、 where 節の条件 e.a0 < X を機能ディスクシステムでは IDC のフィルタリング機構によりオンザフライに処理される。表に問合せの結果を示す。基本性能評価でも確認できたように、機能ディスクシステムの処理時間は入出力のデータサイズに線形に処理時間が増加する。また、表から、機能ディスクシステムと Gamma の結果がほぼ同じことが確認できる。

Join Query(2)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
FDS-R	6.0 sec	57.2 sec	667 sec
Teradata	35.6 sec	331.7 sec	3535 sec
Gamma	5.1 sec	36.3 sec	703 sec

表 3.12: 拡張ウィスコンシンベンチマークによる結合演算の性能評価結果

Join Query(3)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
FDS-R	8.3 sec	77.3 sec	1039 sec
Teradata	27.8 sec	191.8 sec	2038 sec
Gamma	7.0 sec	38.4 sec	731 sec

表 3.13: 拡張ウィスコンシンベンチマークによる結合演算の性能評価結果

Query3: JoinCselAselB with non-key attributes 以下に示す二つの結合演算処理および条件節を含む問合せを用いて性能評価を行った。この問合せでは、リレーション R とリレーション S は同じサイズある。また、条件節 a.a1 < X による選択の結果 (全体の 10% のタプルが選択される)、リレーション R とリレーション S の結合演算処理後の中間結果のタプル数はリレーション T のタプル数と同数になる。

range of a is relation R
range of b is relation S
range of c is relation T
retrieve into tmp(a.all, b.all, c.all) where a.a1 = b.a1
and a.a1 < X and b.a1 = c.a1

ここで、前の Query2 と同様に where 節の条件 e.a0 < X を機能ディスクシステムでは IDC のフィルタリング機構によりオンザフライに処理される。その結果、選択された a.a1 および b.a1(結合演算時に同じ条件を設定することができる) が IDC によりクラスタリングされ、結合処理が施される。その後、結果リレーションはいったんディスクに書き戻される。続いて、機能ディスクシステムでは、改めて中間結果とリレーション T が読み出され、 e.a1 フィールドと c.a1 フィールドでクラスタリングが行われ、結合演算が施される。

表 3.13に Query3 の結果を示す。機能ディスクシステムの結果は Query1 および Query2 の結果と比較すると、十分な性能が得られていない。これは、機能ディスクシステムのディスクへの書き込み実装が不十分でありデータの読み出しと比較しておよそ 4 倍以上の時間がかかるためである。その結果、中間結果の書き戻しと結果リレーションの書き戻しコストの双方のコストが高くなるため、 Query1 および Query2 の結果よりも他のシステムと比較して十分な性能が得られないことになる。

通常、結合演算処理では結合フィールドが異なるため、演算ごとに二つのリレーションごとにハッシュテーブルを生成するが、上記の問合せのように結合フィールドが一緒の場合には二つの結合演算を合わせ、 3 つのリレーションをまとめて読み込んで処理をすることもできる。いったん、 3 つのリレーションのクラスタが生成されれば、突き合わせ処理はそれぞれの I/O クラスタ内で行えばよいため、中間結果をディスクに書き戻し、再度読み出しを行う必要はない。そこで、機能ディスクシステ

Join Query(3)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
2way join	8.3 sec	77.3 sec	1039 sec
3way join	6.7 sec	62.1 sec	792 sec

表 3.14: 拡張ウィスコンシンベンチマークによる結合演算の性能評価結果

ム上で、３つのリレーションの選択条件およびクラスタリング情報を指定して、結合演算処理を行った。表 3.14にその結果を示す。中間結果に関するディスクアクセスがなくなったため、性能は大幅に向上している。

3.8.3 射影演算 (重複除去)

以下に示す重複除去を含む射影算の問合せを用いて性能評価を行った。すべてのタプルはユニークな値をもっており、実際には重複除去処理で消えるタプルはない。結果リレーションはディスクに書き戻される。

range of e is relation R
retrieve unique into tmp(a.all)

機能ディスクシステムでは、リレーション R の属性１の値を用いてクラスタリングを行っている。クラスタリングされたデータはステージングバッファ上に保持され、並列に処理される。

表 3.15に測定結果を示す。表 3.15の結果からわかるように、 100,000 件の処理時間は 10,000 件のほぼ２０倍かかっている。これは、 100,000 件の処理を行う場合には機能ディスクシステムでは Grace ハッシュ方式が用いられていることによる。 Grace ハッシュ方式では、まず、ソースリレーションがハッシュ関数によってクラスタリングされ、それぞれのクラスタはいったんディスクに書き戻される。その後、それぞれの I/O クラスタごとに読み出され、重複除去の処理がなされるた後、結果リレーションをディスクに書き戻す。したがって、重複したデータがないため、同じサイズのデータが２回ずつディスクからの読み出しと書き戻しが生じることとなる。そのため、ステージングバッファにおさまる程度のデータサイズの処理と比較してディスクコストが倍になる。そのため、１回のディスクの読み書きのみの 10,000 件の処理時間 (13.9 sec) に対して、データサイズが 10 倍となり、さらに 2 倍のディスクアクセスがかかるため、 100,000 件の処理時間が約２０倍 (286 sec) となる。

3.8.4 集計演算

以下の問合せを用いて性能評価を行った。この問合せでは、属性 a2 でグルーピングし、それぞれのグループごとに属性値 a1 の値を総計する。

Projection Query(Duplicate Elimination)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
FDS-R	1.7 sec	13.9 sec	286 sec

表 3.15: 拡張ウィスコンシンベンチマークによる射影演算の性能評価結果

Aggregation Query(Partitions $\bar{1}$00)			
システム名	10,000 tuples	100,000 tuples	1,000,000 tuples
# of partitions	100	1,000	10,000
FDS-R	2.9 sec	24.9 sec	307 sec
Teradata	8.9 sec	24.8 sec	175.8sec
Gamma	2.9 sec	19.5 sec	185.1sec

表 3.16: 拡張ウィスコンシンベンチマークによる集計演算の性能評価結果

機能ディスクシステムでは、属性 a2 の値でクラスタリングを行い、それぞれのグループごとの総計を並列に処理する。

range of e is relation R
retrieve (e.a2, sum=sum(e.a1 by e.a2))

表 3.16は集計演算処理の結果を示す。 1,000,000 件では Grace ハッシュ方式が用いられている。この結果から分かるように、集計演算の処理時間は射影演算と異なり、リレーションサイズに線形に増加しており、 1000,000 件の処理時間は 100,000 件のほぼ１０倍となっている。これは、集計演算では、ステージングバッファ上で処理に必要なフィールドは属性 a1 と a2 の４バイトのみであるため、Grace ハッシュ方式においても I/O クラスタのサイズがソースリレーションサイズと比較して十分に小さい。そこで、全体の処理時間としては、ソースリレーションの読み込み時間が支配的になるためである。また、機能ディスクシステムの結果は Gamma や Teradata の結果と比較すると、 1,000,000 件の場合は処理時間がかかっている。これは、機能ディスクシステムではディスク１台のみの環境で Grace ハッシュ方式を用いて、 I/O クラスタの書き戻し、読み出しが行われるのに対し、 Teradata では複数のディスクで読み出されたデータが Y ネットを用いることで自然に上位にむかって集計処理が行われ、また、 Gamma でも、主記憶上にすべてのデータがロードできるため、ソースリレーションを複数のディスクで読み出すだけであるため、ディスク処理の並列効果が大きいことによる。

3.9 不均一分布データにおける性能評価

本節では、不均一なデータに対するハッシュを用いたアルゴリズムの性能について我々が試作した機能ディスクシステム上で検討を行なう。機能ディスクシステムは関係データベース演算処理の性能向上を目指した超高速二次記憶システムであり、ハッシュを用いた動的クラスタリング機能をハードウエアとして実装することで、ディスクからのデータ転送速度に追従した処理を可能としている。また、複数台のプロセッサを導入することで並列処理による性能向上を図っている。データの不均一性に対するハッシュアルゴリズムの評価においては、機能ディスクシステム上で実装されているＧＮハッシュ方式を用いて、不均一なデータに対する並列処理の効果およびオーバフローバケットに対するＧＮハッシュ方式の効果について詳細な解析を行う。

3.9.1 データの不均一性とハッシュを用いたアルゴリズム

ハッシュを用いた結合演算アルゴリズムでは、ソースリレーションをハッシュ関数を用いて小さなバケットに分割する。一般にハッシュアルゴリズムの検討を行なうときには、ソースリレーションの分布が一様であり、かつ、バケットが一様に分割されることを前提に性能評価が行なわれているが、実際のデータベースにおいて完全に一様なデータ分布であることはほとんどない。また、一旦ハッシュ関数を適用した後、バケットのデータ分布が一様であることは保証できない。つまり、分割後のバケットのサイズが同じとは限らない。したがって、ここでは、分割後のバケット・サイズが一様にならない場合のアルゴリズムの性能について考察を行なう。まず、ハッシュを用いたアルゴリズムに対して不均一な分布のデータが与える影響についてＧｒａｃｅ　Ｈａｓｈ方式を例にとって考える。Ｇｒａｃｅ　Ｈａｓｈ方式は、データ分割フェーズ（スプリットフェーズ）と結合処理フェーズ（プロセッシングフェーズ）にわかれており、スプリットフェーズでは、ディスク内のソースデータをメモリに格納できる大きさになるように予めソースリレーションのサイズ等から分割数を予測してクラスタ（Ｉ／Ｏクラスタ）に分割を行なう。続いて、これらのＩ／Ｏクラスタを再び、メモリ上にロードし、結合演算処理を行なうことになる。そこで、本ハッシュ方式においてデータの分布が不均一であるとは、つぎの二つの場合に分けて考えることができよう。第一にハッシュ関数を用いて大規模データをメモリに入るようにＩ／Ｏクラスタに分割する時、分割されたＩ／Ｏクラスタのサイズそのものが不均一である場合が考えられる。Ｉ／Ｏクラスタを生成するときに用いるハッシュ関数を次のメモリ上での結合演算処理に用いられる分割用ハッシュ関数と区別するためにスプリット関数と呼び、Ｉ／Ｏクラスタが均等に分割できないことを’スプリット関数におけるデータの不均一性’と呼ぶ。第二に、分割された各クラスタ毎の処理においてメモリ上の処理におけるデータの不均一分布が考えられる。各クラスタ毎のメモリ上での処理については、ネストループ方式、ソートマージ方式、ファインハッシュ方式などが考えられるが、本評価ではファインハッシュ方式を用いてメモリ上でＩ／Ｏクラスタを小さなバケット（プロッセシングクラスタ）に分割し、それぞれのプロセッシングクラスタは、ネストループ方式で処理を行なうものとする。このメモリ上でのプロセッシングクラスタの分割時の不均一性を’ハッシュ関数における不均一性’と呼ぶ。本方式の評価においては、この二つの場合に分

けてそれぞれ評価を行ない、データの不均一性に対するハッシュを用いたアルゴリズム性能について検討する。

機能ディスクシステムにおける処理と上記の二つの不均一性はつぎのように対応している。まず、’ハッシュにおけるデータの不均一性’とは機能ディスクシステムのステージングバッファ上に展開されたバケットのサイズが不均一であることに対応している。機能ディスクシステムではＩＤＣが用意しているハッシュ関数を用いて分割されたバケットを複数台のプロセッサで並列に処理を行なっている。今回の評価では、タプルの分布は正規分布に従って変化させ、その処理性能を解析する。一方、’スプリット関数におけるデータの不均一性’については、リレーションがステージングバッファ・サイズよりも大きい場合にＩ／Ｏクラスタを生成する時点でのデータの不均一性とさらにＩ／Ｏクラスタを処理する際のバケットの不均一性の二種類の影響が考えられる。後者については、’ハッシュ関数における不均一性’として解析できると考えられる。そこで、すでに機能ディスクシステム上に実装されているＧＮハッシュ方式を用いてＩ／Ｏクラスタのサイズが不均一になった場合についての性能評価を行なう。

3.9.2 ハッシュ関数におけるデータの不均一性に対する性能評価

前節で延べたように、ハッシュ関数におけるデータの不均一性に関しては、大容量ステージングバッファに IDC の動的クラスタリング機構を用いて生成されたデータバケットの並列処理効果が期待される。

機能ディスクシステム上でデータがメモリにはいる場合の処理方式についてはすでに図 3.4に摸式図を示した。機能ディスクシステムでは、まず、データをＩＤＣを通してメモリ上にロードする。このとき、ＩＤＣの動的クラスタリング機能を用いてディスクの読みだしに追従してプロセッシングクラスタの生成が行なわれる。これらのハッシュを用いてさらに小さく分割されたプロセッシングクラスタを複数台のプロセッサが並列に取り込むことで処理が行なわれる。処理結果は、各プロセッサからディスクに書き戻される。

性能評価は以下に示すような結合演算処理を実行した。

range of e is relationR
range of f is relationS
retrieve (e.all, f.all) where e.a0 = f.a0

ここでは、ハッシュ関数における不均一分布の性能評価として、機能ディスクシステム上でのデータ処理時間を測定して考察する。ディスクからのソースデータの読みだし時間や処理結果の書き戻し時間は扱うデータの総量のみに依存しており、データ分布の変化とは関係なく一定のため、ここではメモリ上に展開されたプロセッシングクラスタの処理時間のみを取り上げる。また、フィールドａ０は２バイトの整数であり、図 3.13に示すような正規分布をしているものとする。各プロセッシングクラスタのサイズは、次の式に従うものとする。

h: プロセッシングクラスタ数

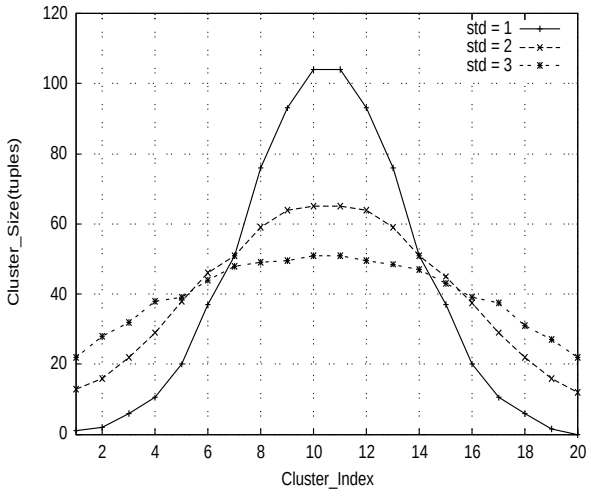

図 3.13: プロセッシングクラスタのサイズ

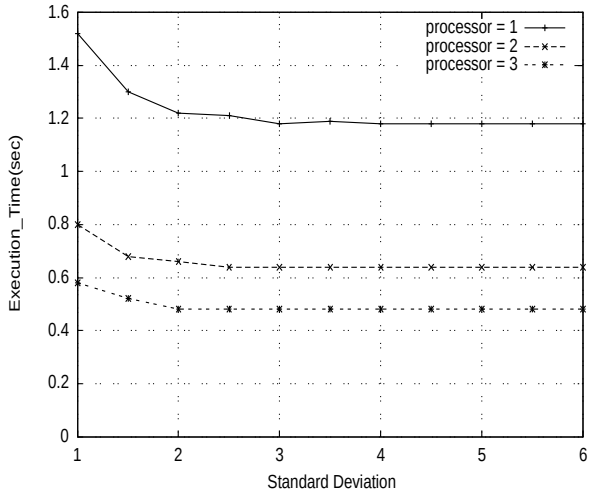

図 3.14: 複数プロセッサによる並列処理効果

R,S: ソースリレーションサイズ（ＳはＲと同じとする）

σ： 標準偏差

ｉ： クラスタインデクス（１<= ｉ <= ｈ）

$$S(\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^6 \exp^{\frac{-(x-3)^2}{2\sigma^2}} dx$$

とすると、プロセッシングクラスタｉは以下のように表される。

$$i = \frac{\frac{1}{\sigma\sqrt{2\pi}} \int_{\frac{(i-0)6}{h}}^{\frac{i6}{h}} \exp^{\frac{-(x-3)^2}{2\sigma^2}} dx}{S(\sigma)} * 2R$$

機能ディスクシステム上でのプロセッシングクラスタの処理性能について図 3.14に示す。ここでは、リレーションＲとＳのタプル数をそれぞれ１０００、プロセッシングクラスタ数を２０、プロセッサを１ ３台に変化させている。図 3.14は縦軸に処理時間、横軸に標準偏差を示す。正規分布を用いているため、標準偏差の値が小さいほどデータの偏りが激しいといえる。この図 3.14から標準偏差が小さい場合を除いてほぼ一様分布の処理時間に収束していることがわかる。また、プロセッサ台数が増加するにつれて並列処理の効果により、標準偏差が小さくても一様分布と同じ処理時間に収束することが確認できる。また、プロセッサ２台と３台について比較すると標準偏差が１の場合のプロセッサ３台の処理時間はプロセッサ２台の場合の一様分布の場合の処理時間より小さく、ハッシュ関数におけるデータの不均一性はプロセッサの並列処理効果により十分対処できることが確認された。

3.9.3 機能ディスクシステムにおけるＧＮハッシュ方式

機能ディスクシステムではＧｒａｃｅハッシュ方式とネストループ方式を組み合わせた GN ハッシュ方式を採用している。本方式では、ソースリレーションのサイズとフィルタリングファクタを用

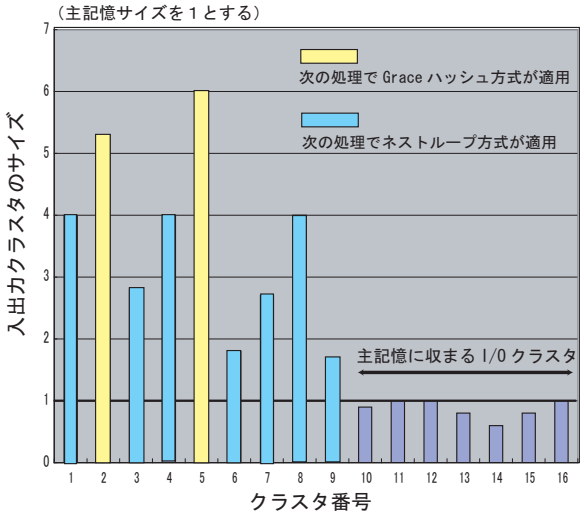

図 3.15: Ｉ／Ｏクラスタの分布

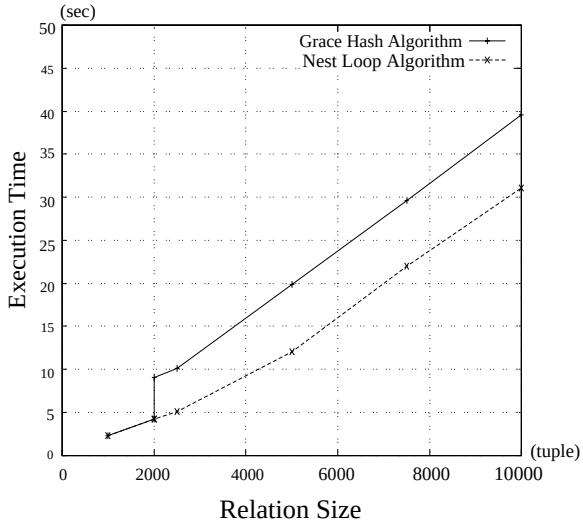

図 3.16: Grace ハッシュとネストループアルゴリズム (拡大図)

いて算定される入出力コストの小さい方式を Grace ハッシュ方式またはネストループ方式から実行時に選択して用いている。第６章にて改めて詳述するので、本節では簡単に処理の流れを紹介する。

本方式はスプリットフェーズとプロセッシングフェーズの二つから構成される。まず、スプリットフェーズで、ソースリレーションの一部を読み込むことで、フィルタリング・ファクタを推定し、この値とソースリレーションのサイズから入出力コストを算出する。データがメモリに比較して十分大きい場合にはＧｒａｃｅハッシュ方式の処理コストが小さいが、メモリの数倍程度であれば、ネストループ方式の方が処理コストが小さい。これは、Ｇｒａｃｅハッシュ方式では、クラスタ生成のためのデータ書き戻し時間とクラスタ読み出し時間が必ず必要になるが、ネストループ方式では、常にデータの読み出し時間のみでよいため、メモリの数倍程度のリレーションでは、内側のリレーションを２－３回読み出すだけでよいためである。そこで入出力コストから、二つのどちらの方式を用いるか決定し、さらに生成するＩ／Ｏクラスタの個数やネストループ方式のループ回数を算出する。これらの入出力コスト式については、第６章にて詳細に述べている。Ｇｒａｃｅハッシュ方式が選ばれた場合には、スプリット関数を用いてソース・リレーションの分割が行なわれる。一方、ネストループ方式では処理ループの回数などが決定される。このとき、Ｇｒａｃｅハッシュ方式で分割されたＩ／Ｏクラスタがメモリを越えるオーバーフローＩ／Ｏクラスタであるなら、再帰的にスプリットフェーズを呼び出すことで改めて二つの方式のうち、適当な方式がそのオーバーフローＩ／Ｏクラスタに適応される。例えば、図 3.15のようにデータ分割後のＩ／Ｏクラスタのサイズが分布している場合、メモリに収まるようなＩ／Ｏクラスタ (１０番以降) の処理については問題ないが、メモリを越えたＩ／Ｏクラスタ（１番から９番）については、改めてそのクラスタ・サイズから処理方式が決定される。この場合、図 3.15の１番めのクラスタは、ネストループ方式で処理され、２番めのクラスタは、Ｇｒａｃｅハッシュ方式で再度分割される。同様にｎ番めまでのＩ／Ｏクラスタの処理方式が決定さ

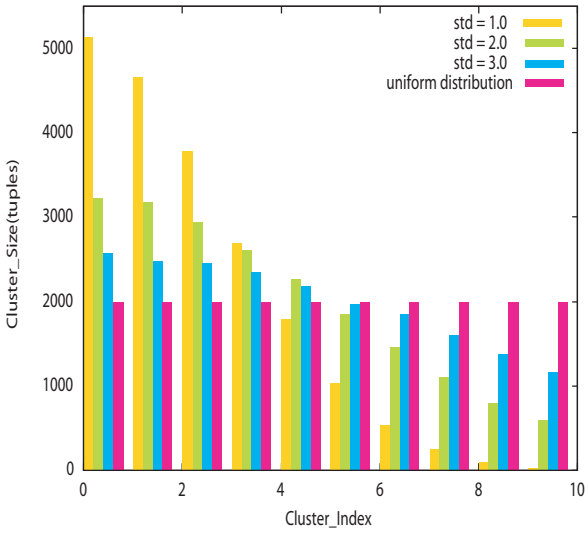

図 3.17: 正規分布の I/O クラスタサイズ

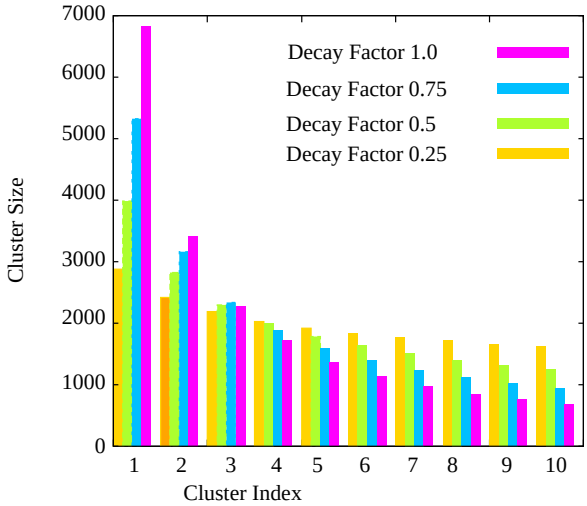

図 3.18: Zipg 分布の I/O クラスタサイズ

れ、その処理にしたがって再度分割が行なわれる。スプリットフェーズが終了するとプロセッシングフェーズに入り、Ｉ／Ｏクラスタ単位に３節で述べたようにハッシュ関数を用いた結合演算処理を行なう。

3.9.4 スプリット関数における不均一データの性能評価

データが不均一な場合のＧＮハッシュ方式の性能は、つぎのように考えられる。

データがメモリの数倍程度の小さいリレーションの場合 ソース・リレーションのサイズから、ネストループ方式が選択される。ネストループ方式では、読み出された総データ量は一様分布の場合と同じであり、前節で述べているように極端な分布を除いては、プロセッシング時間は並列処理の効果もあって一様分布の場合と同じ処理性能がえられるため、データの分布による処理時間の変かはほとんどない。

大規模データの場合 まず、Ｇｒａｃｅハッシュ方式が選択されスプリット関数を用いてソース・リレーションの分割が行なわれる。このとき、Ｉ／Ｏクラスタのサイズがメモリを越える場合には前述のように再帰的にネストループ方式かＧｒａｃｅハッシュ方式のいずれかが選択される。通常、一旦分割されたＩ／Ｏクラスタは極端なデータ分布でない限り、メモリの数倍程度の大きさになると期待される。図８にネストループ方式とＧｒａｃｅハッシュ方式の処理性能をメモリの数倍程度のリレーション・サイズのときについて示す。この図からわかるようにＧｒａｃｅハッシュ方式ではＩ／Ｏクラスタの生成を行なうためにＩ／Ｏコストが２倍になるため、わずかでもメモリを越えると処理時間が急激に増加する。しかし、ネストループ方式では連続的に処理時間が変化している。したがってこの二つの方法を組み合わせることで、総処理時間として最適の時間が得られることになる。

ここでは、不均一データとしてクラスタのサイズが正規分布および Zipf 分布に従ったの場合の機能ディスクシステムの処理性能を検討する。

Ｉ／Ｏクラスタのサイズが正規分布に従う場合には図 3.17に示すようなＩ／Ｏクラスタの構成となる。ここでは、各Ｉ／Ｏクラスタのサイズ分布を

Ｍ： メモリ・サイズ

Ｒ： リレーション・サイズ（ＳはＲと同じとする。）

σ： 標準偏差

ｈ： Ｉ／Ｏクラスタ数（（Ｒ＋Ｓ）／Ｍ）

ｉ： Ｉ／Ｏクラスタのインデクス（１＜＝ｉ　＜＝ｈ）

とした場合に次の式に従って算出する。

$$S(\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^3 \exp^{\frac{-(x-3)^2}{2\sigma^2}} dx$$

として各Ｉ／Ｏクラスタ・サイズは以下のように表される。

$$I/OCluster_i = \frac{\frac{1}{\sigma\sqrt{2\pi}} \int_{\frac{(i-0)3}{h}}^{\frac{i3}{h}} \exp^{\frac{-(x-3)^2}{2\sigma^2}} dx}{S(\sigma)} * R$$

Ｉ／Ｏクラスタのサイズが Zipf 分布に従う場合には図 3.18に示すようなＩ／Ｏクラスタの構成となる。ここでは、各Ｉ／Ｏクラスタのサイズ分布を

Ｍ： メモリ・サイズ

Ｒ： リレーション・サイズ（ＳはＲと同じとする。）

ｚ： decay factor

ｈ： Ｉ／Ｏクラスタ数（（Ｒ＋Ｓ）／Ｍ）

ｋ： Ｉ／Ｏクラスタのインデクス（１＜＝ｋ　＜＝ｈ）

とした場合にｋ番目のＩ／Ｏクラスタ・サイズは以下の式に従って算出する。

$$Cluster_k = \frac{R}{k^2 * \sum_{k=1}^{h} \frac{1}{k^2}}$$

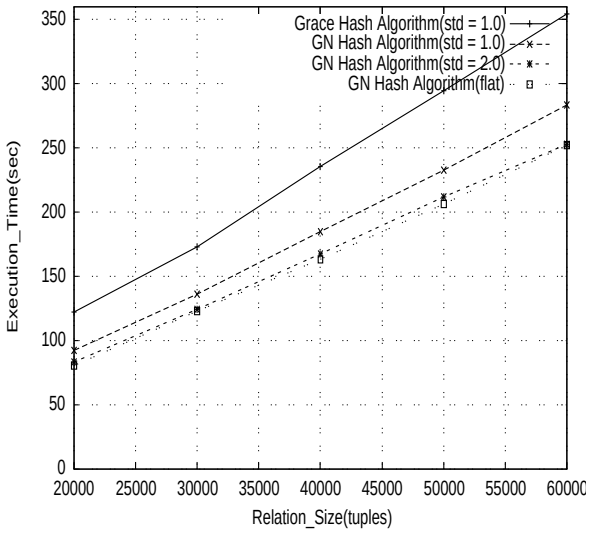

図 3.19: 正規分布の場合の処理時間

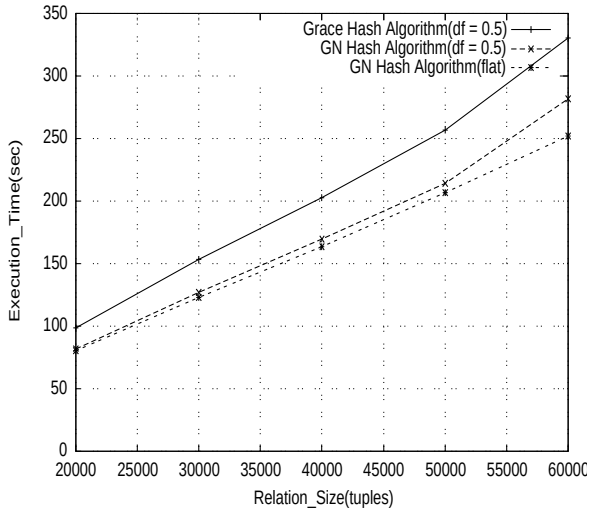

図 3.20: Zipf 分布の場合の処理時間

リレーション・サイズの変化による処理性能

ここでは、正規分布に従ったＩ／Ｏクラスタの処理時間について、リレーション・サイズを２０、０００件から６０、０００件まで変化させた結果について図 3.19に示す。タプル長は１２８バイト、ステージング・バッファ・サイズは５１２ＫＢで二つのリレーションＲとＳは同じ分布であるとする。また、結合度は１であるため、処理結果は、２５６倍との２０、０００件から６０、０００件のタプルが生成されることとなる。この図からわかるようんに分散の変化によらず、ほぼ、リレーション・サイズに線形に処理時間が増加していることがわかる。また、いずれの分散でもＧｒａｃｅハッシュ方式のみで処理するよりも処理時間が短いことが確認できる。一方、分布が極端に偏っている場合には、処理時間が増加しているが、分布の偏りがそれほどでもない場合には、ほぼ、一様分布の処理時間と同じである。

同様に、 Zipf-like 分布に従ったＩ／Ｏクラスタの処理時間について、リレーション・サイズを２０、０００件から６０、０００件まで変化させた結果について図 3.20に示す。タプル長は１２８バイト、ステージング・バッファ・サイズは５１２ＫＢで二つのリレーションＲとＳは同じ分布であるとする。また、結合度は１であるため、処理結果は、２５６倍との２０、０００件から６０、０００件のタプルが生成されることとなる。この図からわかるようんに分散の変化によらず、ほぼ、リレーション・サイズに線形に処理時間が増加していることがわかる。また、いずれの分散でもＧｒａｃｅハッシュ方式のみで処理するよりも処理時間が短いことが確認できる。一方、分布が極端に偏っている場合には、処理時間が増加しているが、分布の偏りがそれほどでもない場合には、ほぼ、一様分布の処理時間と同じである。

分散の変化による処理性能

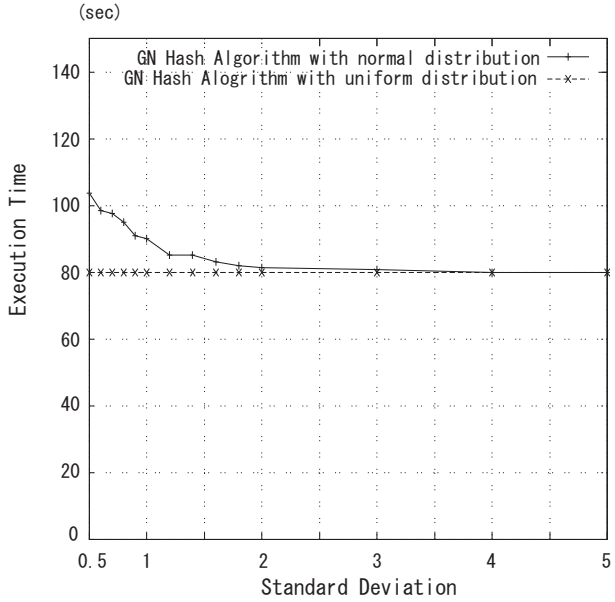

図 3.21: 標準偏差を変化させた場合の結合演算

２０、０００件のリレーションを用いて、I/O クラスタのサイズが正規分布 (図 3.17) 従うとし、標準偏差σを変化させた場合の処理時間の変化について図 3.21に示す。図からわかるように、標準偏差が非常に小さい場合（つまりデータの偏りが激しい場合）には処理時間が大きくなるが、標準偏差が 1.0 以上の場合は一様分布の処理時間とほぼ同じであることがわかる。

3.10 本章のまとめ

機能ディスクシステム（Functional Disk System with Relational database engine : FDS-R）は，プロセッサ本体と二次記憶システム間のＩ／Ｏボトルネックの問題に着目し，二次記憶システムを単なる記憶媒体ではなく，それ自身がデータに対して高レベルな処理機能を持つことにより，関係データベース・システムとしての二次記憶の性能向上を図ったものである。特に，大容量のステージング・バッファと複数台のプロセッサを導入することで，大規模データ処理を効率良くかつ高速に行っている。また，関係代数演算を支援する専用ディスク・コントローラを新しく開発し，専用ハードウェアによる ” 機能 ” も実装している。すでにＦＤＳ－Ｒ第１版の試作機を開発し，基本性能についての計測を行い，一般の商用データベース・システムと比較して高い性能を得られることを確認した [134]。また，ＦＤＳ－Ｒ上におけるデータベース・システムのプロトタイプとして，QUEL Subset System の実装を行っている。続いて第１版で得られた知見をもとに新たにＦＤＳ－Ｒ第２版を開発し，大規模リレーションに対する関係代数演算の処理方式の考察を行い，本システムに適合した方式の実

装を行た。第１版におけるデータ処理は，ＩＤＣによるデータのフィルタリング後の処理結果はステージング・バッファに収まるものとして実現されていた。これに対し，第２版では，大容量データの処理方式として，さらにＩＤＣに対する制御とステージング・バッファに対する管理方式を追加することにより，ステージング・バッファから溢れるデータに対する処理をサポートした [59]。

さらに、機能ディスクシステムにおける関係代数演算処理方式として本システムに最適化したＧＮハッシュ方式を実装した。本方式では，実行時に入出力コストを比較することで処理方式をハッシュを用いたネストループ方式と Grace ハッシュ方式のいずれかに決定する。機能ディスクシステムでは，汎用 OS を用いた場合に通常データベースシステムに生じるオーバヘッドコストを大幅に削減することで関係代数演算の処理コストは入出力バウンドとなっており，ＧＮハッシュ方式を有効に実装することが可能である。 GN ハッシュ方式で用いられている入出力コストの算出式及び評価式について検討を行い，試作機上での計測結果から本方式におけるアルゴリズム選択評価方式が有効であることを示した。また、メモリのサイズに係わらず、 Grace ハッシュ方式の処理時間がほぼ一定であることから、機能ディスクシステムが大規模リレーションの処理において有効であることが確認できた [59]。

第 4 章

共有メモリ計算機における並列ハッシュ結合演算処理

4.1 共有メモリ計算機と並列データベース処理

１９７０年代 E.F.Codd によって提案された関係データベースは、強固な論理基盤、非手続き的なユーザインターフェイスなど多くの特長を有する反面、処理負荷は重く、その商用化には１０年以上の歳月を要した。関係データベースシステム実用化の最も大きな課題の一つはその性能向上にあり、関係演算の高速化に関して多くの研究がなされてきた。とりわけ２つのリレーションの動的な結合を可能とする結合演算は、単純な手法では両リレーションのタプルの積に比例した処理時間が必要となるため最も処理負荷が重く、その性能向上を目的として種々のアルゴリズムが開発されてきた。

ネストループ技法はアルゴリズムが簡単であり、小規模なリレーションに対しては中間リレーションを生成しないため最も高速な手法であるが、大規模なリレーションに対しては負荷が過大となり適用できない。これに対しソートマージ技法が用いられたが、ソート自体負荷が重いためより高速化が望まれてきた。８０年代に入りハッシュによる結合演算技法が開発され、ほぼタプル数に比例した処理時間で結合演算が可能となった [26, 12, 61]。このハッシュによる結合演算技法は現在最も高速な結合アルゴリズムであると考えられている [26, 25]。

一方、近年、単一プロセッサの性能限界から、複数個のプロセッサを用いた並列プロセッサの実用化が進み、メインフレームやワークステーションのサーバ機では、マルチプロセッサアーキテクチャを採用する事例が増えつつある。上述の如く、関係データベース処理は負荷が重く、並列マシンによる性能向上の可能性が模索されている。本章では現行のメインフレームやワークステーションに見られる共有メモリマルチプロセッサ上での関係データベース、とくに結合演算の並列処理技法について検討するとともに商用マルチプロセッサであるシンメトリ S81 上に実装し、評価を試みる。

従来の多くの研究は、シミュレーションに基づくものであり、実際に実装された例は少ない。ウィスコンシン大学ではトークンリングで結合した疎結合マシン上での並列処理に関し種々の実験を行なっている [25, 28] が、共有メモリマルチプロセッサに関しては研究がなされていない。日本では、清木らがワークステーションの複合体や、共有メモリマルチプロセッサなど一般化された並列システム上での並列データベース処理に関し実験を行なっている [63, 64] が、結合演算はネストループ技法を用いておりハッシュ結合演算技法の評価は行なわれていない。また、 NTT のグループでは、データベースマシン RINDA の開発と共に筆者らと同様に共有メモリマルチプロセッサ上で関係データベースの並列処理に関する研究を行なっており [41, 40]、適応的なページサイズの変更による負荷分散の効率化を試みているが、測定はディスクの入出力は考慮せず全てのリレーションが主記憶に常駐されると仮定しており、想定している環境が異なる。なお、主記憶上の結合演算の並列処理に関しては著者らも既に性能評価を行なっている [136]。

Lu らは、共有メモリマルチプロセッサの上でのハッシュ結合演算技法の性能をシミュレーションによって評価することを試みている [74]。従来の単一プロセッサ環境での評価では、排他制御が必要となるハッシュテーブルや出力バッファへの書き込みに関し競合を考慮しておらず不十分とし、競合確率を算出し、より正確な評価を行なうとともにハイブリッドハッシュの改良版を提案している。彼らの主張する競合が最も顕著に表れるのはスプリットフェイズにおけるバケットバッファであり、例

えばリレーションが小規模で分割数を２とするような時に、ディスクを６台駆動しプロセッサを１０台以上駆動させると問題が出るとしている。我々の実装では出力バッファをディスクごとに設け分散化しているため競合が減っており、またタプルの移動全体をクリティカルセクションとせずアドレスポインタの変更のみを排他制御の対象とすることからも競合は減少しており、Lu らの論文が取り上げる問題は実装上の工夫で十分回避できると考えられる。この他 Lu らはディスクの台数を一定とし、プロセッサの台数を変えるシミュレーションを行なっているが、測定範囲はほとんど CPU バウンドな環境下であるため、その結果は当然のものであり興味深いとは言えない。データベース処理では主記憶に展開されたデータに対する複数プロセッサによる並列処理と同時に入出力の並列化による高速化が重要と考えられる。我々は与えられた入出力性能の下でどの程度の並列性が内在するか明らかにした後に入出力を含めたスケーラビリティ、並列台数効果を明らかにしており、視点が大きく異なっている [62, 141]。すなわち、入出力デバイスとＣＰＵの双方共とも拡大可能な計算機環境に於いて両者の負荷を均衡させた上でそれぞれの台数効果の確認を試みている。

以上の如く、近年ワークステーション等で広く採用されつつある共有メモリマルチプロセッサ上での並列ハッシュ結合演算技法の実装並びにその評価に関する研究は未だ十分なされていない。本章では共有メモリマルチプロセッサに適したハッシュ結合演算技法の並列化を行なうとともに商用マルチプロセッサに実装し、詳細な評価を行なうことによりその有効性を明らかにする。

以下、第 4.2節で従来提案されてきた種々のハッシュ結合演算技法を紹介した後、第 4.3節において本稿で使用する商用共有メモリマルチプロセッサ、シンメトリＳ８１の構成を紹介するとともに第 4.2節で述べた GRACE ハッシュ結合演算技法の並列化ならびにその実装方式について考察する。第 4.4節では、シンメトリＳ８１上に実装した実験システムを種々の側面から評価する。即ち、結合演算の並列処理効果について詳細な測定を行なうとともに、ほぼ理想的な台数効果が得られることを示す。第 4.5節は本章のまとめであり、共有メモリ計算機上における並列データベース処理の今後の課題について触れる。

4.2 ハッシュに基づく結合演算技法

4.2.1 ハッシュに基づく種々の結合演算技法

ハッシュによる結合演算手法は、前節に述べたように、多くの場合最も高速な手法と考えられており、その最も基本的な手法が GRACE ハッシュ結合演算技法 [61] である。これは対象とする２つのリレーションの結合属性に同一のハッシュ関数を施していくつかのバケットに分割し、その後に同じハッシュ値を有するバケット同士でつき合わせ処理を行なうことにより結合演算結果を得る手法である。異なるバケット間では結合される可能性がないことに着目し、ハッシュを施しリレーションを分割するという前処理を導入することにより、処理負荷を大幅に低減することが可能となっている。

ハイブリッドハッシュ結合演算技法 [26] は単純なシンプルハッシュ法と GRACE ハッシュ法を融合させた手法であり、その名称もこの事実に依っている。 GRACE ハッシュ結合演算技法では前処理としてのハッシュ分割時には主記憶を入出力バッファとしてしか使用していないため、その利用効

率が低い。ハイブリッドハッシュ法ではこの点に着目し、１つ目のバケットをハッシュ分割時の余った主記憶空間を利用して処理することにより、特に主記憶に比べてそれほど大きくない比較的小規模なリレーションに対する処理性能の向上を図っている。

このような改良を行なったとしても、ハッシュ関数の分割のゆらぎを常に避けることは不可能であり、ハイブリッドハッシュ結合演算技法において必ずしも１つ目のバケットが残った主記憶空間を十分に利用するという保証はない。動的 GRACE ハッシュ結合演算技法 [87] は主記憶に残すバケットを動的に選択する適応的な手法であり、不均一なデータ分布に対しても安定な性能を得ることを目的として改良されている。更に最近ではハッシュ技法の並列化に伴うスキューの取り扱いを含めたアルゴリズムの改良が種々試みられている [126, 60]。このようにハッシュ結合法には種々の変形が存在するが、ここでは簡単のため、基本型である GRACE ハッシュ結合演算技法に関し共有メモリマルチプロセッサ上での並列化を試みる。

4.2.2 GRACE ハッシュ結合演算技法

本節では、基本となる GRACE ハッシュ結合演算技法の処理の流れに関して簡単に述べる。本ハッシュ法では２種類のハッシュ関数（スプリット関数、ハッシュ関数）が用いられる。前者はリレーションを主記憶より小さな複数のバケットに分けるために、後者は主記憶上でタプルのつき合わせ処理を行なうために用いられる。結合演算の対象となるリレーションを R、 S、結果リレーションを T とし、R の大きさは S 以下とする。また、スプリット関数は１から N までの整数、ハッシュ関数は１から M までの整数を関数値として返すとする。このとき、 GRACE ハッシュ結合演算技法は次のような２つのフェイズからなる。

スプリットフェイズ

本フェイズでは１つのリードバッファと N 個のライトバッファを主記憶上に割り当てる。まず R を１ページずつリードバッファに読み込み、タプルの結合属性に対しスプリット関数を適用する。この時の関数値を i とすると、 i 番目のライトバッファにそのタプルを転送する。ライトバッファが一杯になると内容をバケット Ri($1 \leq i \leq N$) としてディスクに書き出す。こうして R を N 個のバケットに分割し、 S についても同様に N 個のバケット Si に分割する。

ジョインフェイズ

本フェイズではリレーション R および S 共用のリードバッファ１つと結果リレーション T 用のライトバッファ１つを主記憶上に割り当てる。残りの領域は M 個のエントリを持つリレーション R のバケット用のハッシュテーブルとして利用する。ジョインフェイズではリレーション R のバケット Ri に対し以下のビルドサブフェイズを実行した後リレーション S のバケット Si に対しプローブサブフェイズを実行する。これを全てのバケットに適用する。

ビルドサブフェイズ

まず、 Ri バケットを１ページずつリードバッファに読み込み、各タプルの結合属性に対

しハッシュ関数を適用する。この時の関数値を m とすると、ハッシュテーブルの m 番目のエントリにそのタプルを追加する。バケット Ri の全てのタプルに対して上記の処理を繰り返す。

プローブサブフェイズ

Si バケットを１ページずつリードバッファに読み込み、各タプルの結合属性に対しハッシュ関数を適用する。この時の関数値を m とすると、ハッシュテーブルの m 番目のエントリを検索し、同じ結合属性を持つタプルがあれば結果タプルを生成してライトバッファに転送する。ライトバッファが一杯になると内容を結果リレーション T としてディスクに書き出す。バケット Si の全てのタプルに対して上記の処理を繰り返す。

GRACE ハッシュ結合演算技法の各フェイズにおけるデータの流れを図 4.1に示す。

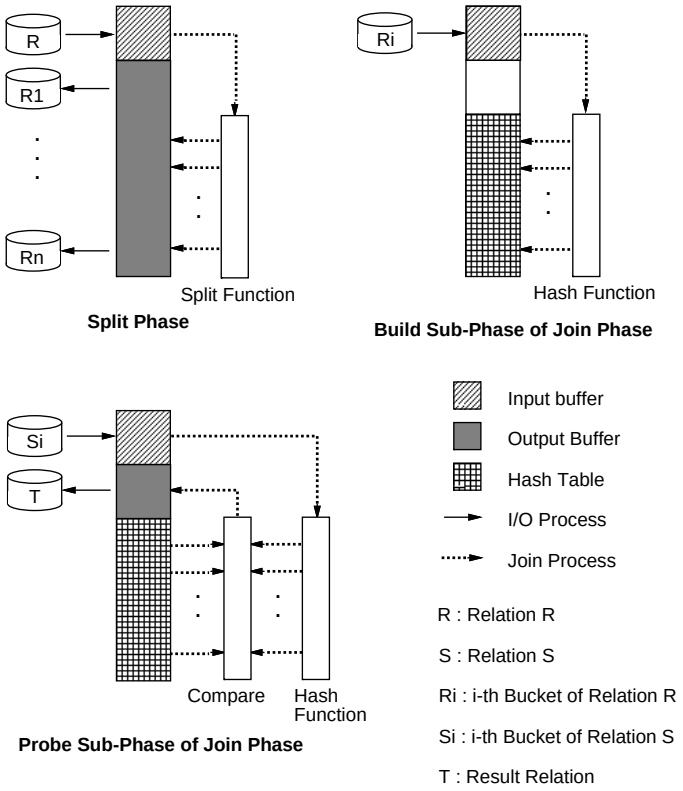

図 4.1: GRACE ハッシュアルゴリズム

4.3 共有メモリマルチプロセッサへの実装

4.3.1 実験環境

ワークステーションからメインフレームに至るまで近年マルチプロセッサ化が進んでおり、このような環境でデータベース処理がどの程度高速化可能であるかを明確にすることが本研究の目的であ

るが、現行の商用ワークステーションでは未だ並列度は低く設定されており、接続可能なプロセッサの数も１０台程度と限定されている。一方、近年のマルチプロセッサにおけるキャッシュコヒーレントプロトコルに関する研究の進展により、より多数のプロセッサを共有バスに接続することが可能になりつつある。このような背景から、シーケント社シンメトリＳ８１を用いて、並列処理による性能向上に関する評価を試みることにした。シンメトリは i80386(16MHz) を用いたプロセッサボードを高速共有バス (80MB/ 秒) を介して結合したアーキテクチャをとっており、図 4.2に示されるように本評価においては１８台のプロセッサ、４台のディスクコントローラ、８台のディスクからなるシステムを用いた。ここでＤＣＣなるディスクコントローラは１つあたり２つのチャネルを有し２台のディスクを同時に駆動することが可能となっている。共有メモリは４０ＭＢ、ディスクは１台あたり約６００ＭＢの容量を有する。シンメトリシステムでは UNIX に並列処理機構を強化した DYNIX と呼ばれる OS が採用されており、ユーザによって生成されたプロセスは空いているプロセッサに動的に割り振られ並列に実行される。

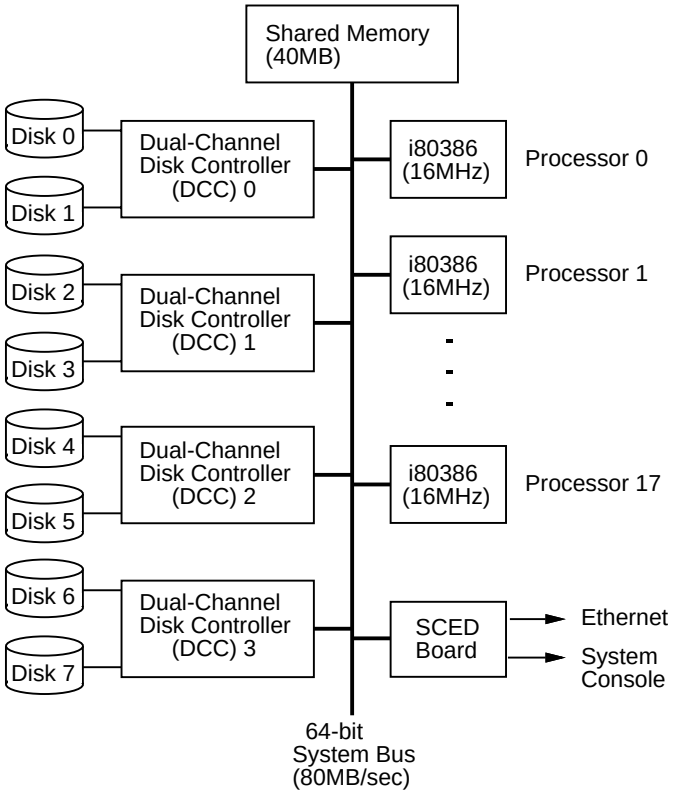

図 4.2: 本実験で使用したシンメトリ S81 のハードウェア構成

4.3.2 共有メモリマルチプロセッサに対する GRACE ハッシュ結合演算技法の並列化

上述の共有メモリマルチプロセッサＳ８１に対する２章で示した GRACE ハッシュ結合演算技法の並列化について検討する。前章で述べたように GRACE ハッシュ結合演算技法はそのアルゴリズ

ムの性質から容易に並列化できるが、単純な並列化を採用しても、 Lu らの結果からもわかるように共有資源への不必要な排他制御による競合などがおきて、並列効果を得られない [74]。そこで、今回の実装においては、ディスクを複数台用いることによる入出力性能に対する並列効果を最大限に得られるような実装を目指した。図 4.1から分かるように、 GRACE ハッシュ結合演算技法では、ディスクから読み出したデータを主記憶にロードし、主記憶上で必要な処理を行ない、データをディスクに書き戻すという一連の流れの繰り返しであり、入出力動作と主記憶上の処理を分離して並列化を考慮することができる。従って、以下では GRACE ハッシュ結合演算技法の各フェーズ及び入出力動作について内在する並列性を抽出し、並列効果が最大限得られるような並列処理方式を検討する。

（1）スプリットフェイズの並列化

スプリットフェイズでは、演算対象のリレーションをリードバッファに読み込み、タプルの結合属性に対しスプリット関数を適用し、関数値に応じたライトバッファにタプルを転送する。主記憶上のタプルに対するスプリット関数の適用処理は多数のプロセッサを用いて並列に実行することが可能である。このとき、タプルの転送は同一ライトバッファに対する同時書き込みを防ぐために排他制御が必要となるが、通常バケットの数はプロセッサの数に比べると多いため衝突は少なく、またタプルの転送時間を考慮するとタプル長が極端に短くない限り、ロックのオーバヘッドは問題にならない。

（2）ビルドサブフェイズの並列化

スプリットフェーズと同様に主記憶に読み込まれたタプルに対するハッシュ関数の適用処理は並列に実行することが可能である。このとき、タプルの転送は同一ハッシュエントリに対する同時書き込みを防ぐために排他制御が必要である。一般にハッシュエントリの数はプロセッサの数に比べ極めて多く、そのオーバヘッドは問題にならない。

（3）プローブサブフェイズの並列化

バケット Si のタプルの結合属性に対するハッシュ関数の適用、ハッシュエントリの検索、結合演算等の処理は並列に実行することが可能である。このとき、結合演算結果のライトバッファへの転送は排他制御が必要であるが、実際には多数のハッシュエントリの検索に時間がかかり、また結合率は通常低いため衝突は少なく、並列に実行することが可能である。

（4）入出力の並列化

前述のフェーズ毎の考察で述べたように、複数プロセッサによる並列処理に加えて、複数ディスクによるデータの並列入出力により性能を大幅に向上させることが可能である。特にデータベース処理では実行時間の多くは２次記憶装置のアクセスで消費されており、複数ディスクの並列駆動は極めて有効と考えられる。ディスクの台数に比例して入出力の並列度を抽出するためには各ディスクのデータ量を均等化することが望ましいため、今回の実装では関係データベースシステムにおける基本要素であるタプルを単位としたソフトウェアによるストライピングの適用を試みることとする。これにより、データベースの各タプルはラウンドロビン手法でディ

スク毎に均等に配置され、各々のディスクは独立に入出力動作可能であるため、台数分の並列効果が期待できる。これは、デクラスタリングとも呼ばれ、 GAMMA,XPRS 等の実験システムでも採用されている。

4.3.3 実装方式

前節での検討から明らかなように、 GRACE ハッシュ結合演算技法は多大な並列性を内在しており、適切な並列化を行なうことにより大きな性能向上が期待できる。本節では GRACE ハッシュ結合演算の並列処理のための実装方式について検討する。

前節で述べたように、 GRACE ハッシュ結合演算技法では入出力動作と主記憶上での処理を分離することができる。今回の実装ではそれぞれを 2 種類のプロセス、即ち入出力プロセスならびにジョインプロセスで独立に行なうこととした。入出力プロセスはディスクの読み書きを管理し、ジョインプロセスはタプルの比較や転送などその他の共有メモリ上での処理を行なう。すなわち、入出力プロセスは図 4.1において実線の矢印で示されているデータ転送を行ない、ジョインプロセスは点線の矢印で示されている処理を行なう。また、データベース処理においてその実行時間の大半は 2 次記憶装置へのアクセスに費やされており、ディスクを出来る限り動作状態とするため、各ディスクにつき 1 つの入出力プロセスを生成し、各入出力プロセスは特定のディスクのアクセスのみを管理するものとする。

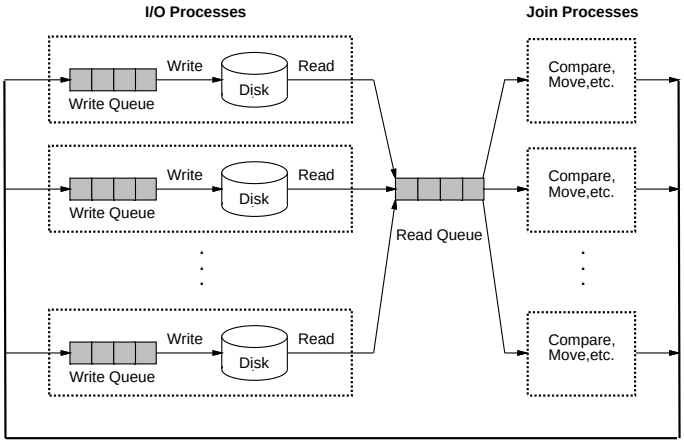

図 4.3: プロセス間のデータの流れ

これら二つのプロセスは、主記憶上に設けられた共有ページキューを通してデータ転送、データ処理を行なう。基本的には、ディスク上のすべてのデータを入出力プロセスがページ単位で共有ページキューを通して主記憶に転送し、ジョインプロセスは、データを読み込まれたページがある限り、所定の処理を行なうこととなる。今回の実装では、主記憶上に構成される共有のフリーページ管理キューを用意し、入出力プロセスはディスクから 1 ページ分のデータをフリーページ上に読み込み、リードキューに追加する。ジョインプロセスはリードキューから 1 ページとり、各フェイズに応じた処理を

タプル毎に行ない、結果を同様にフリーページキューから獲得したライトバッファに書き込み、一杯になるとライトキューに追加する。各入出力プロセスは固有のライトキューを持っており、ジョインプロセスは各ライトキューへの出力データ量を管理する変数に従ってデータを分散し、ディスク毎のデータ量が均等になるようにする。共有キューによるデータの通信の様子を図 4.3に示す。

以上のように、本処理では、入出力プロセスにより生成されるページによって駆動される。従って、入出力処理の効率化が全体性能に与える影響が大きいことから、入出力動作を出来る限り、中断しないようシステムを構築した。すなわち、共有メモリ上における共有キューを採用し、入出力時のバッファを各ディスクに固定せず、すべてをフリーページとしてキューからのポインタで管理している。これにより、各ディスクの読み込みはキューからのページ獲得の時のみ競合するだけで、入出力プロセス同士はほぼ独立に動作でき、十分並列効果が得られる。また、入力動作を優先的に行なうために、出力バッファが一杯になった時、あるいは一連の動作の終了時のみ入力動作が中断され、出力動作に切替えられる。一方、各ジョインプロセスはリードキュー上のページを処理後、優先的にフリーキューに登録する。

分散メモリマシンではメッセージパッシングにより実体を送受する必要があるのに対し共有メモリマルチプロセッサでは共有メモリ上のデータ移動を多くの場合ポインタの付け換えで実行することが可能であり、本研究での実装においてもデータの転送量を極力減らすよう工夫している。例えば、ビルドフェーズでは、リードバッファからハッシュエントリ領域へのデータ転送をせず、各タプルに対してポインタをはることで、リードバッファをそのままハッシュエントリ領域として用いている。これにより、ロック時間がタプル転送時間ではなくポインタの付け替え時間のみとなり、競合が避けられると同時に、タプルの転送がなくなることでジョインプロセスの負荷が減り、入出力プロセスが読み込むデータを遅滞なく処理することができる。

4.4 並列 GRACE ハッシュ結合演算技法の性能評価

4.3.1節に示した構成のシンメトリＳ８１マルチプロセッサ上に、 4.3.3節で述べた方針に従い並列 GRACE ハッシュ結合演算技法を実装し、種々の側面から性能評価を行なうことにより、共有メモリマルチプロセッサの並列関係データベース処理に対する有効性を明らかにする。

本節の性能評価に用いるリレーションのフォーマットはウィスコンシンベンチマーク [6] に基づいており、タプル長２０８バイト、結合演算の対象となるアトリビュートは４バイト整数であり、リレーションのカーディナリティをＮとすると０〜（Ｎ－１）のユニークな値を有し、その出現順序はランダムに設定されている。対象リレーションの大きさは指定しない限り３０万タプルとする。結合率は１００％、即ちＮタプルの２つのリレーションの結合結果はＮタプルから構成され、各タプルは４１６バイトとする。演算対象となるリレーションは８台のディスクに対してタプル単位で均等に分配されている。また、ハッシュ結合演算途中で生成される中間リレーションも同様に生成される。また本評価では共有メモリとして８ＭＢを使用した。なお、以下の測定はすべて１０回の実測値の平均値を示す。

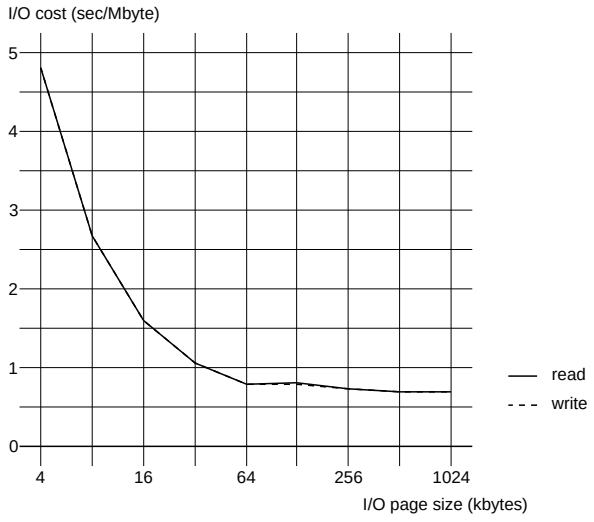

図 4.4: ページサイズと入出力性能

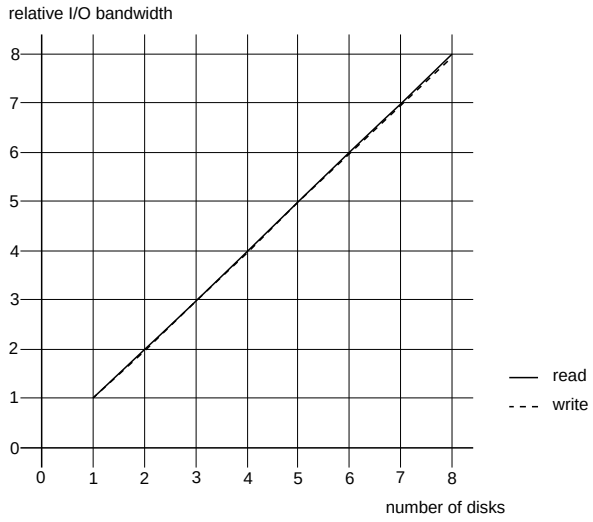

図 4.5: ディスクの並列駆動による入出力性能の向上

以下、まずページサイズ、並列駆動ディスク数の変動に対する入出力性能を測定した後、スプリットフェイズ、ジョインフェイズの各フェイズの並列性を駆動プロセッサ数を変化させることにより測定する。最後に結合演算全体としての並列処理効果について評価する。

4.4.1 ページサイズと入出力性能

一般にページサイズはプログラムの参照の局所性を考慮し、適切な大きさが決定され、通常５１２バイト〜４ｋバイトの数値が採用されることが多い。ここではプログラムではなくデータベースそのものを格納する際のページサイズについて検討することとする。最近のメインフレームにおける拡張アーキテクチャでも同様にプログラムとデータのページサイズを別々に設定可能となっている。

関係データベースにおける問い合わせ処理では一般に連続したデータをアクセスすることが多い。インデックスの利用も考えられるが、非クラスタリングインデックスの利用はその選択率がかなり小さい場合にしか効果を発揮せず、連続アクセスの高速化は必須である。このような背景から、ここではページサイズを変えながら入出力時間を計測した。図 4.4に１Ｍバイト当たりの読みだし時間を示す。図から明らかなように、ページサイズの増大とともに入出力性能は向上するものの、６４ｋバイト程度で飽和し、それ以上の改善は少ない。この結果に基づき、以下の計測においてはページサイズは６４ｋバイトと設定することとした。

4.4.2 並列入出力性能

前節において実験環境を示したが、シンメトリにおいて８台ものディスクを並列駆動するような事例は稀であることから、まず並列入出力に関する性能を確かめることとした。同時に駆動するディスクの台数を変化させ、入出力性能を測定した結果を図 4.5に示す。読み出し、書き込みいずれの場

合にもほぼ理想的な並列駆動効果が得られることがわかった。ただし、実験の過程で入出力バッファの先頭アドレスによって入出力性能が変化する傾向が見受けられた。入出力バッファバウンダリを変化させつつ８台の並列入出力性能を測定した所、バッファバウンダリが５１２バイト未満では、書き込み性能が低下することがわかった。また読み出しの性能はバッファバウンダリに殆んど依存しない。この結果に基づき、以下の実験において入出力バッファは全てその先頭アドレスが５１２の倍数になるように設定することとした。なお、図 4.5の並列ディスクアクセスによる性能はこのようなバッファを用いて測定した。

4.4.3 スプリットフェイズの並列処理効果

4.3.2節では GRACE ハッシュ結合演算技法の並列処理について各フェイズに分けて検討した。ここではスプリットフェイズの並列処理効果について測定する。動作させるディスクの数、すなわち入出力プロセスの数をパラメータとし、ジョインプロセスの数を変化させて、スプリットフェイズでの消費時間を測定した。結果を図 4.6に示す。

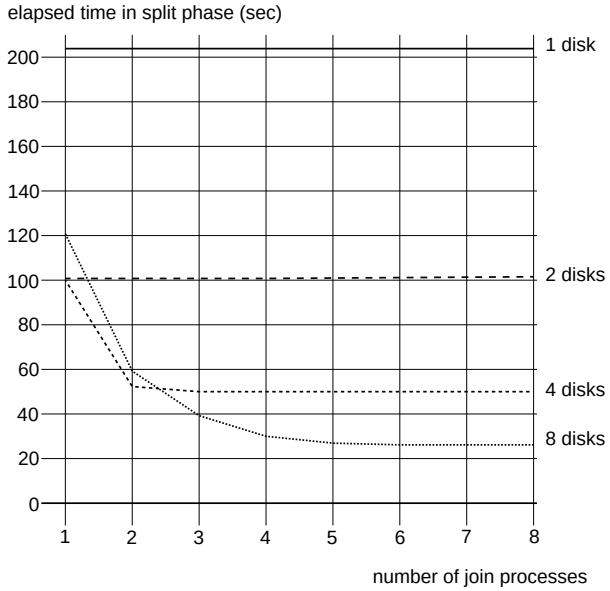

図 4.6: ジョインプロセス数とスプリットフェイズの実行時間

ここでジョインプロセスの数とは正確にはジョインプロセスを実行するプロセッサの数を指している。またディスク駆動台数をパラメータとしているが、前節で述べたように各ディスクに対して１台のＩ／Ｏプロセス（プロセッサ）が割り当てられ、データのディスクからの読み出しおよび結果の書き込みを司っている。

１台または２台のディスク使用時はジョインプロセスの数は性能に影響を与えないことがわかる。しかし、４台または８台のディスクを使った時は、消費時間はジョインプロセッサの数の増加に伴い減少し、次第に一定の値に漸近する。これはディスクからのデータの流れに十分追従できるだけのジョ

インプロセスが供給されている、すなわち処理が入出力バウンドになっていることを示している。最高性能を保つための最小のジョインプロセス数は、ディスク４台時で３、ディスク８台時で６である。このことから、スプリットフェイズにおいて、ディスク１台からのデータを遅れなく処理するには、入出力プロセス用に１台、ジョインプロセス用に 0.75 台で十分であることが分かる。

スプリットフェイズの並列台数効果については、紙面の都合上グラフをまとめて後節の図 4.11に示すが、図から明らかなように理想的な台数効果が得られた。ここでデータベース処理は、主記憶上での処理を主体とする科学技術計算とは異なり、入出力負荷の占める割合が大きいことから、並列台数効果を示すグラフは横軸をプロセッサ数とするのではなく、ディスクとディスク１台当たりに必要なプロセッサを単位としている点に注意されたい。

実際の関係データベース処理では、結合演算だけが単独で行なわれることは少なく、以下の例のように選択演算を伴うことが多い。

```
select * from A,B
where A.key = B.key       (1)
and   A.a < X             (2)
and   B.b > Y             (2)
```

上の条件節（where 以下）において、(1) は結合演算を表す。その他の述語 (2) の数は条件節の複雑さを表す。上記 SQL 文は選択演算と結合演算を分離して各々独立に処理することも可能であるが、通常の SQL コンパイラは結合演算のスプリットフェイズにおいて条件節を判定することで効率化を図っている。スプリットフェイズの消費時間はこの条件節の複雑さによって大きく変化する。

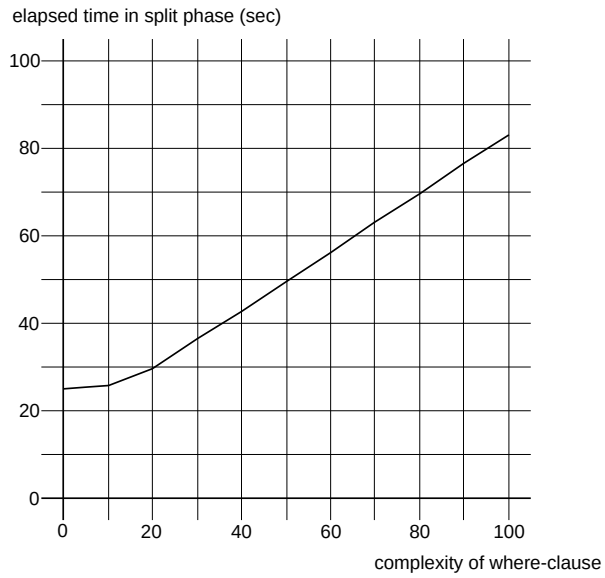

図 4.7: 述語数とスプリットフェイズの実行時間

８台のディスク、８つの入出力プロセス、８つのジョインプロセスを駆動し、述語数を０から１

００まで変化させて、スプリットフェイズにおける消費時間を測定した。結果を図 4.7に示す。条件節がごく簡単なときには消費時間は述語数によらずほとんど一定である。これは、ジョインプロセスの数が十分であるために、追加した条件節を含めた処理がディスクの速度に十分追従できるだけの速度で行なわれていることを示している。ジョインプロセスの数に余裕があることは図?? から明らかである。一方、述語数が２０を越えると、消費時間は条件節の複雑さにほぼ比例して増加していく。これはジョインプロセスの数に余裕がなくなり処理が CPU バウンドになったことを示している。

4.4.4 ジョインフェイズの並列処理効果

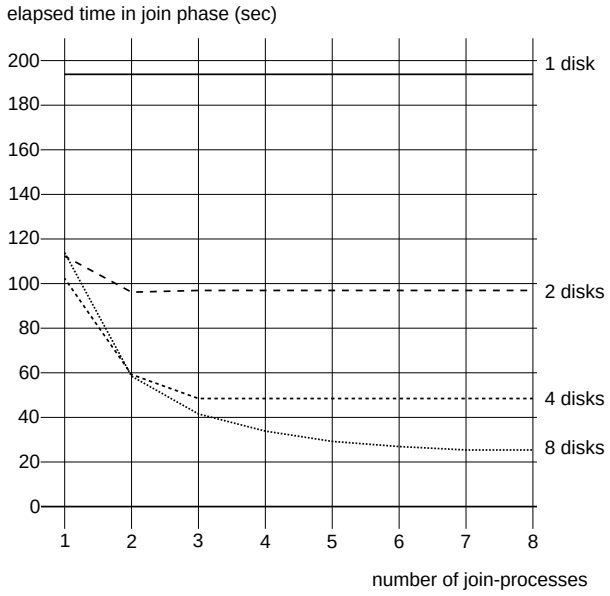

図 4.8: ジョインプロセス数とジョインフェイズの実行時間

本節ではジョインフェイズの並列処理効果について述べる。スプリットフェイズの測定と同様に、動作させるディスクの数、すなわち入出力プロセスの数をパラメータとし、ジョインプロセスの数を変化させて、ジョインフェイズでの消費時間を測定した（図 4.8）。ディスク１台使用時はジョインプロセスの数は性能に影響を与えない。しかし、２台以上のディスクを駆動すると、消費時間はジョインプロセッサの数の増加に伴い減少し、次第に一定の値に漸近する。これは処理が入出力バウンドになったことを示している。最高性能を保つための最小のジョインプロセス数は、ディスク２台時で２、４台時で３、８台時で８である。このことから、ジョインフェイズではディスク１台からのデータを遅れなく処理するには入出力プロセス用に１台、ジョインプロセス用に１台、で十分であることが分かる。

ジョインフェイズにおいて性能に最も大きな影響を与える要因としてハッシュテーブルのエントリ構成が挙げられる。ハッシュエントリ数が多いほど、ビルドサブフェイズ時でのタプル挿入のロック競合が生ずる割合が減少し、またハッシュエントリが多いことは逆に１つのエントリ当たりのタプルリスト長が短くなり、プローブサブフェイズ時のサーチ時間が減少することになる。この様にハッ

シュテーブルのエントリ数を増加させることにより、メモリの消費は増大するが性能は向上すると考えられる。

８台のディスク、８つの入出力プロセス、８つのジョインプロセスを駆動し、１ハッシュエントリ当たりのタプル数を変化させてジョインフェイズにおける消費時間を測定した（図??）。ハッシュテーブルにおける１エントリ当たりの平均タプル数を横軸とする。図?? の結果と異なり、ハッシュエントリ当たりのタプル数にほぼ比例して消費時間が増加している。これは図 4.8から分かるように、８ディスク、８ジョインプロセス時にはジョインプロセスの余裕が殆んどなく、そのためハッシュを粗くすることによる CPU 負荷の増加が直接性能に影響を与えるためと考えられる。このようにハッシュテーブルを細かくとることは性能向上に大きく寄与することが明らかであり、他の測定ではハッシュエントリ当たりのタプル数は１と設定している。

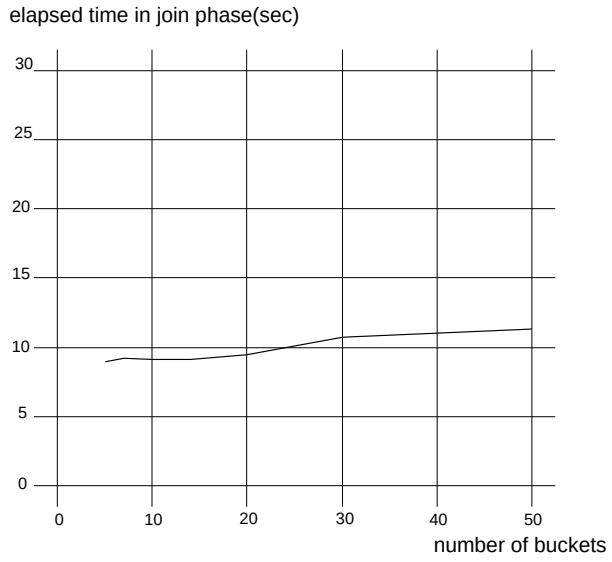

図 4.9: ジョインフェイズにおけるバケット数と実行時間

ハッシュテーブルにおけるエントリの粗さと同様にスプリットフェイズで生成するバケットの数も性能を左右するパラメータとなる。ハッシュテーブルの場合はなるべく細かく分割することが望ましいという結果が導かれたが、バケットの分割に関しては、極端にバケット数を多くしない限り、分割数は性能にあまり影響を与えない。図 4.9に、横軸をバケット数、縦軸をジョインフェイズ実行時間とした評価結果を示す。なおこの測定に限り、評価環境をタプル数１０万件、主記憶１６Mバイトと設定した。バケット数が５個から５０個になっても、その性能はほとんど変わりがない。これは、逆に見ると、１個のバケットサイズが２万件でも２千件でも処理時間が変わらない、つまり主記憶量が小さくなっても処理時間が変わらないことを意味している。つまり、 GRACE ハッシュ結合演算技法の性能が主記憶容量にそれほど依存しない性質が並列化しても変らないことを示している。また、バケット数を大きくすることに生じるオーバヘッドは、バケット切替えによるものであるが、実験によりこれは充分に小さいことがわかる。この事実は、ハッシュ結合技法がマルチユーザ環境に於

いても、良好な特性を示唆するものと言えよう。

4.4.5 GRACE ハッシュ結合演算技法の並列処理効果

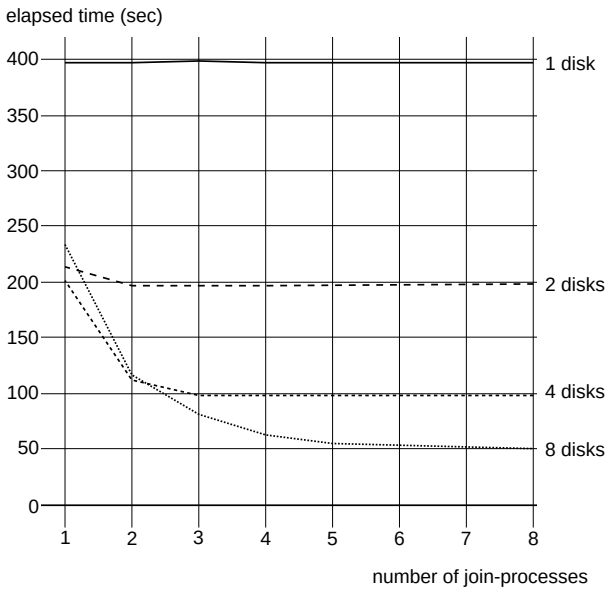

図 4.10: ジョインプロセス数と総実行時間

ここでは前節までの入出力自体の性能と各フェイズにおける並列処理効果の評価に基づき結合演算全体の性能と並列処理効果について検討する。第 4.4.3、 4.4.4節と同様に動作させるディスクの数、すなわち入出力プロセスの数をパラメータとし、ジョインプロセスの数を変化させて、結合演算にかかる消費時間を測定した（図 4.10）。前節までの評価から明らかなように、スプリットフェイズに比べジョインフェイズの方が負荷が大きく、ハッシュジョイン全体の性能特性はほぼこれら２つを加えた形となっている。

図 4.11にシステムのスケーラビリティを示す。ここで横軸はディスク台数で正規化したシステム構成、即ちディスク n 台のときにはＩ／Ｏプロセッサ n 台、ジョインプロセッサ n 台を駆動しており、この時の結合演算性能を縦軸としている。図から明らかなように、ディスク８台時の性能が若干低下しているものの（１台時の７．７倍の性能）、本研究の実装手法によればほぼ理想的なスケーラビリティを達成できることが確認された。

従来のネストループ結合演算技法やシンプルハッシュ結合演算技法が両リレーションのカーディナリティの積に比例した処理時間を必要とするのに対し、 GRACE ハッシュ結合演算技法は両リレーションのカーディナリティの和に比例した時間、即ち線形時間で処理することが大きな特徴となっている。８ディスク、８入出力プロセス、８ジョインプロセス時に、リレーションの大きさを変化させて演算時間を測定した結果を図 4.12に示す。図から明らかなように、ほぼリレーションの大きさに比例した時間で演算が終了しており、本研究で示した並列化実装手法により我々の提案した並列 GRACE ハッシュ結合演算技法が理想的に実装されていることが分かる。

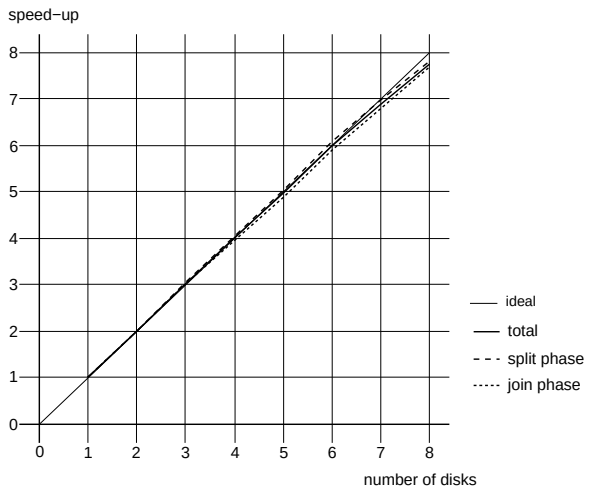

図 4.11: 並列アクセスによる結合演算の性能向上

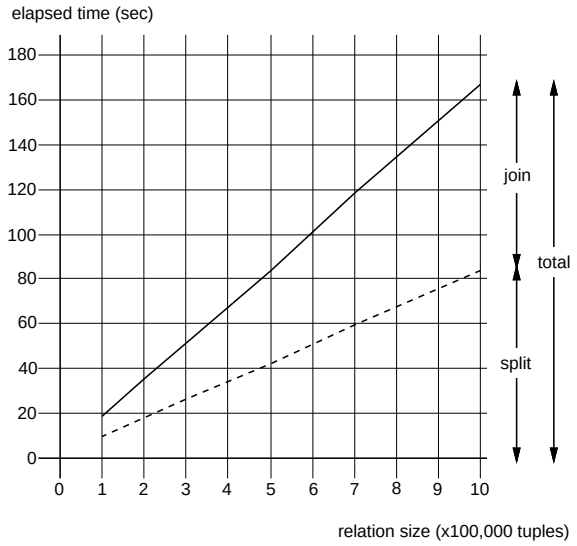

図 4.12: リレーションサイズと実行時間

4.5 本章のまとめ

近年、ワークステーション等でバス結合型共有メモリマルチプロセッサアーキテクチャが広く採用されつつあることから、本章では並列処理による関係データベース処理の高速化の可能性を明確化することを目的とし、最も負荷の重い結合演算を対象に、商用マルチプロセッサであるシンメトリＳ８１に GRACE ハッシュ結合演算技法を実装すると共に評価を行ない、ほぼ理想的な並列処理効果を確認することができた。すなわち、ハッシュ結合演算技法の並列化に関し考察し、共有メモリマルチプロセッサに適したプロセスモデルを明らかにするとともにシンメトリＳ８１上に実装し、スプリットフェイズ、ジョインフェイズ各々に関し並列度を評価し、ディスクからの入出力に追従した処理が可能であることを示した。さらに、ディスク１台当たりに対しプロセッサを２台ずつ増加させることにより、システムを理想的にスケールアップ可能なことを明らかにし、共有メモリマルチプロセッサにより関係データベース処理を効率良く並列処理できることを示した。特にデータベース処理は入出力負荷が大きく、駆動ディスク台数を考慮に入れた並列処理効果の評価が不可欠である。しかしながら、多数の CPU を搭載した共有メモリ計算機が商用化された時点で、このような点に関する十分な性能評価は報告されていなかった。今回のディスク８台、プロセッサ１６台からなる比較的大規模な商用マシンの上での GRACE ハッシュ結合演算技法の実装とその性能評価結果により、関係データベースシステムにおける共有メモリ計算機を示すことがいち早くできた。

本章ではリレーションのデータ分布は一様であり、リレーションはほぼ等しい大きさのバケットに分割でき、バケットは一様なハッシュテーブルに展開可能であるとした。データの分布が不均一な場合の評価は今後の課題である。

第 5 章

分散メモリ計算機における並列多重結合演算処理の最適化技法

5.1 多重結合演算スケジューリングとは

近年、高性能なマイクロプロセッサの開発、大容量メモリの低価格化、高速ネットワーク、ディスクの低価格化、小型化などを背景に商用並列計算機が多数登場し、多くの関係データベースシステムが並列計算機の上に実装されるようになってきた. 並列関係データベースにおける並列処理の研究では関係データベースにおける演算内並列処理, 複数演算間並列処理の研究が負荷の重い多重結合演算を中心に多く行なわれているが, 多重問合せ間並列処理に関しては, いまだ十分な研究が行なわれていない [98]. 特に, 単一問合せ内の並列多重結合演算スケジューリングの最適化については多くの研究 [19, 71, 75, 98, 104, 107, 131, 130] が行なわれているが, この結果を基にした並列多重問合せの最適化についてはほとんど研究されていない. また、提案された多重結合演算処理アルゴリズムのほとんどは、共有メモリ環境を対象としており、分散メモリシステムにおいあて、入出力コスト、ネットワーク転送コストなどの資源の消費について考慮したアルゴリズムの報告は少ない。

[98] にて述べられているのように、計算機技術の進歩により、 CPU 性能およびメモリサイズは年々増加している。この結果、関係データベース処理においては、より多くのリレーションが主記憶上にロードできることになる。そこで、演算間並列処理を実現した多重結合演算の新たな並列処理方式について考察する必要ある。 IBM の system R[99] では多重結合演算の最適化として演算の処理順序を決定するだけであり、多重結合演算の並列処理については考慮していない。 [98] では、演算間の並列処理を実現することで、従来の結合演算処理を順次行う手法より性能が良くなることが報告されている。しかしながら、この報告の主たる関心は分散メモリシステムにおける演算間並列処理の重要性を示すことであり、多重結合演算処理の最適化は考察されていない。 [44] では、並列問合せ処理の最適化が議論されているが、全共有システムにおける並列タスク処理の CPU および入出力の制限について考察している。 [19, 75] では、多重結合演算処理の最適化のためのグリーディアルゴリズムを提案している。しかしながら、ハッシュテーブルを作成する際の主記憶のサイズとソースリレーションサイズに関して議論しているが、複数個の結合演算処理が終了したところで、中間結果はつねにディスクに書き戻される。さらに、提案されたアルゴリズムは共有ディスクシステムにおける環境のみを想定している。 [71, 104, 131] は、多重結合演算の最適化手法を提案し、主記憶を最大限利用し、中間結果の書き戻しコストを削減することで、性能が改善されると報告している。 [131] で提案されているアルゴリズムは共有システムおよび分散メモリシステム、双方の計算機環境を対象としているが、いかにシステム資源の利用のバランスを図っているか、特に分散メモリ環境におけるネットワーク転送と入出力アクセスのオーバラップについては、 [131] では明確に記述されていない。 [104] らは、 dynamic programming の並列化を扱っているが、彼らは共有システム上の多重結合演算のコスト関数のみを検討している。 [107] では、多重結合演算のコスト式が詳細に検討されているが、彼らが研究に用いているプラットフォームも共有システムである。

本章では、我々は主記憶の利用および CPU 処理のみならず、ネットワークの転送幅も考慮に入れた新しい多重結合演算処理プランの生成方式について提案する。提案したアルゴリズムでは、多重結合演算の並列処理単位として” バランスシード木” という概念を導入する。 n 段の多重結合演算を

パイプライン方式で処理する際、 n 個のデータストリームがシステムを結合するネットワーク上を流れる。近年、通信バンド幅は急速に大きくなって来ているが、ディスクアレイの進展はまた入出力データストリームの転送幅も増加させている。これは、数個以上の結合演算を行う場合、容易にネットワークが一杯になってしまうことを意味する。結合演算のパイプライン処理中にネットワークのバンド幅を効率よく使うためには、バランスされたシード木が生成される。そのとき、主記憶の利用率、入出力アクセスとネットワークの転送量に関してコスト式を用いて均衡化を図るなど、分散メモリシステムでは、システム資源を消費しないよう、ある程度の制限を設けなくてはならない。これらの均衡化シード木が組み合わされて、最終的に処理順序の定まった木となる。制限条件は、ネットワーク転送と入出力アクセスがちょうど協調して動作できるように決定しなくてはならない。我々のアルゴリズムでは、パイプライン処理のレベルごおに主記憶の利用率を優先することでシステム資源消費の均衡を図り、その結果主記憶の利用を考慮しただけの従来の方式よりも性能が改善される。

本章の構成は以下のよになっている。次節にで、新たな多重結合演算処理アルゴリズムについて詳細に述べる。 CPU 処理、入出力アクセスおよびネットワーク転送のシステム資源を考慮したコスト式を導入する。続いて、システム資源に限りがある亜倍のバランスシード木の生成方針について述べる。 5.3では、 5.2で導入したコスト式を用いたシミュレーション結果について報告する。我々のアルゴリズムが生成する多重結合演算処理スケジュールは従来の方式の結果よりも、質の良いスケジュールが得られる。最後 5.4にて本章のまとめを述べる。

5.2 Balanced Seed Tree アルゴリズム

提案するアルゴリズムは、問合せ処理の最適化と同時にシステム資源の利用を考慮した多重結合演算処理のスケジューリングを行うものである。

バランスシード木 (Balanced Seed Tree (BST)) という概念を導入し、無共有システム環境での多重結合演算の実行中のシステム資源の消費の均衡化を図る。バランスシード木 (BST) はネットワークのバンド幅を優先にシステム資源を利用し、続いて、主記憶の利用率を考慮する。つまり、ネットワークを通じてパイプライン処理されるデータストリームの並列処理の単位が BST である。最終スケジュール木は、 BST の候補集合から選ばれた木が組み合わせられることで生成さえる。その結果、BST の概念を用いることにより、 BST 単位での分散メモリシステムでの結合演算処理のコストを扱うことで、ネットワークバンド幅を対象としたシステム資源の消費に関して均衡化を図ることが容易となる。

5.2.1 アルゴリズム概観

表 5.1 は提案するアルゴリズムの処理の流れを記述している。提案するアルゴリズムは以下の三個のモジュールとそれを利用したループからなる。

- makeBST バランスシード木を指定されたシードリレーションから生成する

```
NEW ALGORITHM :
memsize = total amount of local memory of each node
relSET = all source relations
join_graphSET = all join graph edges
seedSET = pair of relations in join_graphSET
balanced seed-treeSET = resultSET = φ
while(seedSET ≠ φ ){
    get one_seed from seedSET;
    seedSET = seedSET - one_seed;
    balanced seed-treeSET = makeBST(one_seed);
    while (balanced seed-treeSET ≠ φ) {
        resultSET = makeresult(one_seed, balanced seed-treeSET);
        tmptree = costtree(resultSET);
        if( cost of tmptree < cost of minimum tree){
            minimum tree = tmptree
        }
    }
}

makeBST(one_seed) :
relset = all relations - one_seed;
while(relset != φ) or (any relation of relset does not satisfies the strategies listed below){
    find a combination of relation called "balanced
    seed-tree" from relset by using one_seed
    as a starting base
     strategy 1.
        generate a short right-deep tree in which the
        network bandwidth is fully consumed
        and whose hash tables and temporary result can
        be held in memory
     strategy 2.
        generate a short left-deep tree whose temporary
        result can be held in memory.
     strategy 3.
        generate a short right-deep tree in which all join
        operations can be processed in memory and
        whose temporary relation
        is written to disk and its cost is I/O dominant
     strategy 4.
        mark source relations which fit in memory
     balanced seed-tree_set +=
                    a generated balanced seed-tree
    relset = relset - a generated balanced seed-tree
}
return balanced seed-tree_set and the rest of relset

makeRESULT(balanced seed-treeSET) :
while(balanced seed-treeSET ≠ φ ){
    make result tree in a bottom-up manner using
    a full search
}
return set of result trees
```

表 5.1: Pseudo Code of the Proposed Algorithm

- costtree 与えられた部分木候補の処理コストを計算し、最小コストとその部分木を返す
- makeRESULT 与えられた結果木 (あるいは候補木) から幅優先検索でバランスシード木の候補、あるいは、結果となる組み合わせを求める

主ループでは、結合グラフのエッジで結ばれたリレーションのペアが、バランスシード木の生成のベースとして選ばれる。続いて、 makeBST が呼び出され、バランスシード木の集合画結果として makeRESULT の引数として渡される。 makeRESULT は引数として与えられた全てのバランス木集合が処理されるまで、その集合から最終結合演算処理プランを生成する。 makeRESULT で生成された結果木のコストは costtree を呼び出すことで計算される。メインループは上記の処理を、結合グラフの全てのリレーションの組み合わせを使われるまで、実行される。

システム資源の利用率の均衡化、特にネットワーク転送と入出力アクセスを均等に消費するためには、バランスシード木の集合の生成が必要である。なぜなら、分散メモリシステム環境では、いったんデータストリームがネットワークを通じて大量に転送される、あるいは、ディスクから全部読み出されないと、並列処理は実行されないからである。 [104] らは, 並列処理に dynamic programming 手法で、候補木の数を絞るために高いコストの枝を刈り取る。その結果、中間結果を書き戻す枝のほとんどは刈り取られてしまう。対照的に、我々の方式では、中間結果を書き戻す枝でも条件によっては破棄されす、バランスシード木の候補として選ばれる。例えば、ネットワークバンド幅が入出力転送コストよりも比較的小さい場合は、バランスシード木のプローブフェーズでネットワークコストが非常に大いな場合は、主記憶にリレーションを保持せずとも、ネットワークのコストと入出力コストは相殺できるからである。このようなとき、バランスシード木のレベルでは処理コストが大きくなるが、結果として、処理コスト最適となる。

5.2.2 Balanced Seed-Tree

この節では、与えられたシステム資源とデータベースの静的な情報からバランスシード木を生成する手法について述べる。

制限のあるシステム資源をバランスよく利用するために、バランスシード木を生成するために、二つの評価ポイント、消費条件と主記憶条件を組み合わせた四つの方針を導入する。消費条件とは、対象となっている” バランスシード木” のプローブフェーズにおきて、ネットワークを通してのデータ転送コストがディスクからの読み込み、および必要であれば、中間結果の書き戻しコストを超えないかどうかを意味する。主記憶条件とは、” バランスシード木” の左の葉が主記憶に収まるか、あるいは、中間結果もまた主記憶に収まるかどうかを意味する。この二つの条件を評価することで、バランスシード木のコスト式はシステム資源、特に入出力アクセスコストとネットワーク転送コストの重なりを考慮した式として導き出される。

(1) バランスシード木のコスト

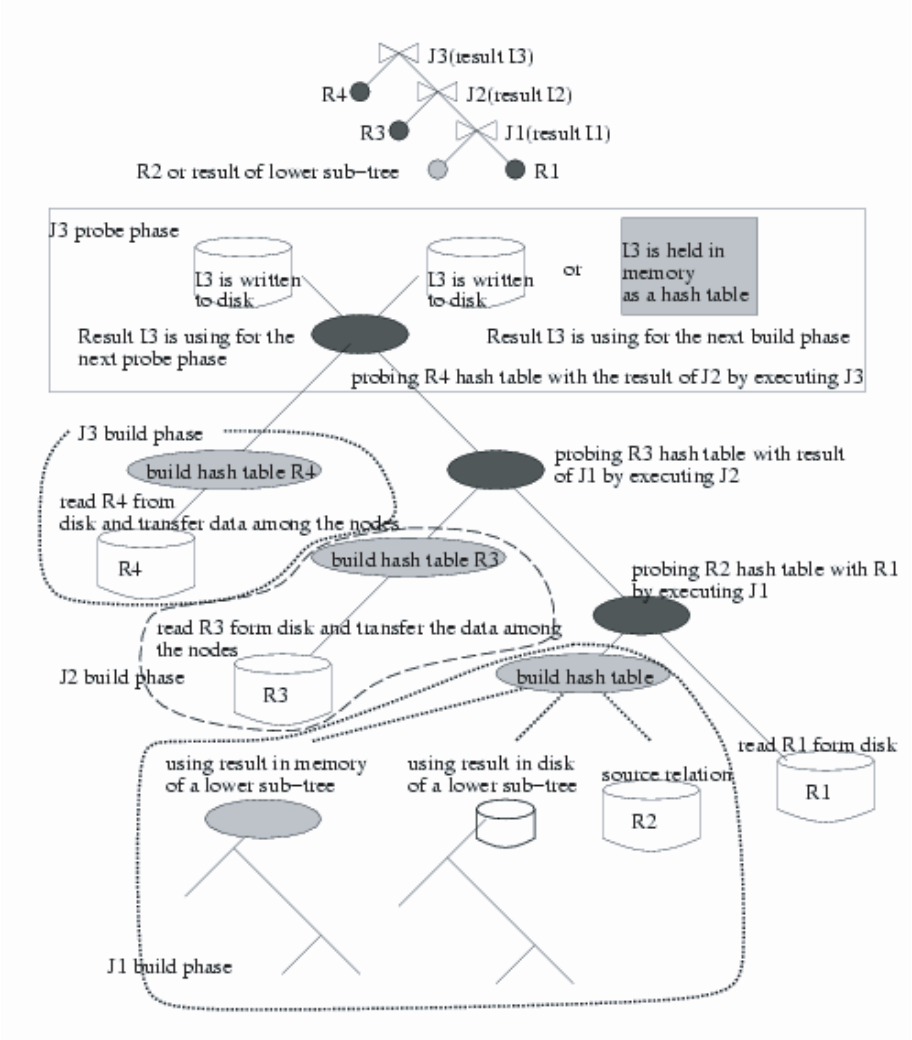

図 5.1: バランスシード木

例として、図 5.1の上部に示すバランスシード木のコストを表 7.1のパラメタを用いて考えて見る。ここでは、分散メモリシステムは N 台のプロセッサ、総計 $M = m \times N$ の主記憶、 N のディスクから構成されるとする。

分散メモリアーキテクチャでのハッシュ結合アルゴリズムの処理では、入出力アクセス、ネットワーク転送、 CPU 処理はほとんどオーバラップすることができる。逆に考えると、ビルドフェーズとプローブフェーズのコストは上記の三つの要素の中で最も支配的なコストによって決定すると考えられる。ハッシュ結合アルゴリズムでは、ビルドフェーズとプローブフェーズは独立に実行されるため、本節のバランスシード木の処理コストも以下のように表す。

$$
\begin{aligned}
Sub\ Tree\ Cost &= Build\ Phase\ Cost_{seed\_tree} \\
&\quad + Probe\ Phase\ Cost_{seed\_tree}
\end{aligned}
\tag{5.1}
$$

パラメタ	説明
N	プロセッサとディスクの台数
$M(m)$	全メモリサイズ (ノードごとのメモリサイズ)
$R(r)$, $S(s)$	リレーション R と S の全サイズ (ノードごとの部分リレーションサイズ)
$\|x\|_{ioseq}$	ソースリレーションから x タプル読み出すコスト
$\|x\|_{ioran}$	中間結果を x タプル読み出すコスト、あるいは、結果を書き戻すコスト
$\|\|x\|\|_{network}$	x タプルを異なるノードへとネットワークを通じて転送するコスト
x_{move}	x タプルを　タプルの分割後、I/O バッファからネットワークバッファへの転送コスト、または、ハッシュ後、ネットワークバッファからハッシュテーブルへの転送コスト
x_{split}	x タプルをクラスタへ分割するコスト
x_{hash}	x タプルをハッシュするコスト
$y * x_{probe}$	y で構成されているハッシュテーブルを x タプルで走査するコスト

表 5.2: コスト式のパラメタ

まず、図 5.1の例では、左のリーフとしてＲ２、Ｒ３、Ｒ４の三つのリレーションがある。全てのハッシュテーブルはソースリレーションＲ２、Ｒ３、Ｒ４から生成されるから、ビルドフェーズのコスト式は以下のように表される。$r_i(i = 2, 3, 4)$ をそれぞれのローカルディスクに格納されている R2,R3,R4 の分割されたリレーションとする。全リレーションサイズは、ソースリレーションは均等に格納できるため、 $R_i(i = 2, 3, 4)$ is $r_i(i = 2, 3, 4) \times N$ となる。

ここで、 $R_2 + R_3 + R_4 < M$ 、 $r_2 + r_3 + r_4 < m$ と仮定する。

$$
\begin{aligned}
& Build\ Phase\ Cost_{seed\_tree} \\
= \ & Build\ Phase\ Cost_{J1} + Build\ Phase\ Cost_{J2} \\
& + Build\ Phase\ Cost_{J3} \\
\simeq \ & \max(|r2|_{ioseq}, ||r2||_{network}, \\
& r2_{split} + r2_{move} + r2_{hash} + r2_{move}) + \\
& \max(|r3|_{ioseq}, ||r3||_{nework}, \\
& r3_{split} + r3_{move} + r3_{hash} + r3_{move}) + \\
& \max(|r4|_{ioseq}, ||r4||_{nework}, \\
& r4_{split} + r4_{move} + r4_{hash} + r4_{move})
\end{aligned} \tag{5.2}
$$

図 5.1の下部に示すように、最後に得る多重結合演算処理では、バランスシード木が組み合わされ

ることになる。ここで例として用いているソースリレーションＲ２の代わりに、下位レベルのバランスシード木によって中間結果が生成されていた場合、ビルドフェーズのコスト式は以下の式 (5.3 or 5.4) によって表される。下位のバランスシード木で生成される中間結果が主記憶に収まる場合、ハッシュテーブルの生成コストは下位のバランスシード木のコストに含まれる。そこで、ビルドフェーズのコストは以下のように表される。

$$Build\ Phase\ Cost_{seed\_tree} = Build\ Phase\ Cost_{J2} + Build\ Phase\ Cost_{J3} \tag{5.3}$$

バランスシード木の中間結果がメモリに収まらない場合、または、ネットワーク転送コストが大きい場合には、中間結果はローカルディスクに書き戻される。そして、次のビルドフェーズとして中間結果がディスクが読み出され、ネットワークを通じて当該のノードへ転送される。この場合、中間結果の読みだしにはランダムな入出力アクセスコストが適用される。

$tempres$ を各ローカルディスクへ格納された中間結果のサイズとする。この場合、ビルドフェーズのコストは以下のように表される。

$$\begin{aligned}
&Build\ Phase\ Cost_{seed_tree} \\
&= Build\ Phase\ Cost_{J1} + Build\ Phase\ Cost_{J2} \\
&\quad + Build\ Phase\ Cost_{J3} \\
&\simeq \max(|tmpres|_{ioran}, ||tmpres||_{network}, \\
&\quad tmpres_{split} + tmpres_{move} + tmpres_{hash} + tmpres_{move}) \\
&\quad + Build\ Phase\ Cost_{J2} + Build\ Phase\ Cost_{J3}
\end{aligned} \tag{5.4}$$

ビルドフェーズでは、入出力アクセスとネットワーク転送は一つの結合演算ごとにパイプライン的に並列に処理されるが、プローブフェーズは複数並列に処理することが可能である。したがって、プローブリレーションＲ１の入出力アクセスとネットワーク転送がオーバラップできる条件について考えなくてはならない。中間結果Ｉ３が主記憶に収まる場合、プローブフェーズのコストは以下のように表される。

$$\begin{aligned}
&Probe\ Phase\ Cost1_{seed_tree} \\
&\simeq \max(|r1|_{ioseq}, \\
&\quad ||r1||_{trans} + ||i1||_{trans} + ||i2||_{trans} + ||i3||_{trans}, \\
&\quad r1_{split} + r1_{move} + r1_{hash} + r1 * r2_{probe} \\
&\quad + i1_{split} + i1_{move} + i1_{hash} + i1 * r3_{probe} \\
&\quad + i2_{split} + i2_{move} + i2_{hash} + i2 * r4_{probe} \\
&\quad + i3_{split} + i3_{move} + i3_{hash} + i3_{move})
\end{aligned} \tag{5.5}$$

全体のコストという観点から、システム資源の均衡のとれた利用について考えると、中間結果をディスクに書きこんだほうが、コストが小さくなる場合もある。その場合のコストは以下で表される。

$$
\begin{aligned}
& Probe\ Phase\ Cost2_{seed\_tree} \\
\simeq\ & \max(|r1|_{ioseq} + |i3|_{ioran}, \\
& ||r1||_{trans} + ||i1||_{trans} + ||i2||_{trans}, \\
& r1_{split} + r1_{move} + r1_{hash} + r1 * r2_{probe} \\
& +i1_{split} + i1_{move} + i1_{hash} + |i1 * r3|_{probe} \\
& +i2_{split} + i2_{move} + i2_{hash} + i2 * r4_{probe})
\end{aligned} \tag{5.6}
$$

(2) Balanced Seed-Tree 生成手順

本節では、バランスシード木および素の集合の生成方針について、詳細に述べる。すでに、二つの評価点、システムリソースの消費条件と主記憶へロード可能かどうかの条件の組み合わせで四つのバランスシード木の生成方針があることは述べた。

最初の方針 (strategy 1) では、最も強い条件がバランスシード木のビルドフェーズに適用される。つまり、資源の消費条件に関しては、入出力コストを超えない範囲で最大にネットワークの転送を行うものを候補として選び、主記憶条件としては、全てのハッシュテーブルと中間結果が主記憶収まるものを作る。もし、この条件のもとで、バランスシード木が生成出来ない場合、以下に述べる strategy2,3,4 と条件を緩めて行く。

strategy1

この方針では、生成されるシード木が主記憶を十分に利用していなくても、ネットワークコストとして最大となるバランスシード木を生成する。

Right Deep の形状をしたバランスシード木を生成するためには、二つ以上のリレーションが結合グラフのエッジを共有していなくてはならない。また、それらのリレーションのハッシュテーブルおよび中間結果は主記憶に収まらなければならない。さらに、式 (5.5) を用いることで, 入出力コストが全体のコストとして支配できてあったとしても、ネットワークコストを最大にする組み合わせを見つける。また、全コスト中、入出力コストが支配的であるなら、基本組み合わせに加え、より小さな主記憶量しか利用しない他の組み合わせについても探す。

strategy 1 を満足するリレーションの組合わせが見つからない場合には、バランスシード木を生成するために、 strategy 2,3,4 と順次、条件を緩めて、適応していく。

strategy2

この strategy では、潜在的な主記憶利用率を最大限にするリレーションの組み合わせを探す。

原則として、バランスシード木は Right Deep 木の形状である。しかしながら、結合率が小さく、中間結果がソースリレーションサイズより小さくなる場合、 left deep の形状をしたバラン

スシード木を生成する。短い left deep 木は、複数のハッシュテーブルを主記憶に保持する必要はなく、常に一つのハッシュテーブルと中間結果を保持すればよい。短い left deep 木はあるだけの主記憶を利用しないが、代わりに残った主記憶を最終スケジュールプランをバランスシード木の集合から生成する際に、主記憶の上位レベルでの利用する機会を増やすことになる。

strategy 1 でバランスシード木に利用されなかったリレーションから、短い left deep 木を構成可能なリレーションの組合せを選ぶ。その組合せの内、最大のソースリレーションよりも生成された left deep 木の最大主記憶利用率が小さいものを捜し出す。

strategy 2 を満足するリレーションの組合せが見つからなかった場合は、次の strategy 3,4 を適用する。

strategy3

このストラテジーではネットワークを使いきらない程度のデータストリームが転送できるバランスシード木を生成する。

strategy 1, 2 を満足しないリレーションの中から、プローブリレーションの読みだしと中間結果の書きもどしの入出力コストが支配的なリレーションの組合せを捜し出す。このストラテジーの条件の理由は、いったん、ネットワークコストが支配的になると、転送待ちのタプルが増え、結果として入出力コストとネットワーク転送はオーバラップしないためである。

このストラテジーは式 (5.6) の二つの項から算出される。つまり、一番目の項である入出力コストがネットワークコストの項よりも大きければよい。入出力コストが支配できあるリレーションの集合から、ネットワークコストはより大きく、主記憶利用率は小さい組合せを探す。

strategy3 を満足しないリレーションが残っていれば、 strategy 4 を適用する。

strategy4

この条件はモジュール makeRESULT が最終スケジューリングプランを作る際の情報を提供するためにある。

上記の条件を満たすことのなかったリレーションの内から、ハッシュテーブルまたは結果リレーションが主記憶に収まるリレーションの組合せを探す。

for the others

残ったリレーションは makeRESULT 内で最終プランに利用される。

これら四つのストラテジーを適用した後は、バランスシード木の集合と利用されなかったリレーションが残る。これらを組み合わせて、 greedy に最終スケジュールプランを生成し、そのコストから最適なものを選択する。

5.3 評価結果

本節では、我々が提案するバランスシード木 (BST) を他の Left Deep 木 (LD)、 Right Deep 木 (RD)、 [19] らが提案した最小ワークロードをベースにしたセグメント Right Deep 木 (MW)、枝のコストバランスを考慮した方式 (BC)、および [104] らが提案した中間結果に主記憶を利用する (BCM) の結果と評価する。 [131] で提案された ZigZag 木は BCM とほぼ同じ木を生成すると考えられるが、ZigZag 木の生成方式が明確になっていないため、この評価では採用しない。並列 dynamic programming 方式を、全空間を探索し、最適解を求めるために用いる。枝の切り方の条件が変化すれば、 dynamic programming の探索空間のサイズと結果の質も変わってしまう。しかしながら、枝の切り落とし条件は [104] では、簡単な例が示唆されるに留まっており、その条件を決定することは本節の内容と離れてしまうため、ここでは議論しない。異なる並列プランを評価するために、本節では以下で述べる人工的に生成されたデータベースと問合せを用いる。

5.3.1 シミュレーション環境とパラメータ

生成されたプランの性能指標として、生成されたプランの実行時間と最適解の実行時間の比を用いることとする。ここでいう最適解とは、 dynamic programming で枝刈を行わず、全空間を探索した解のことである。前述の全ての方式の実行時間は、節 5.2で提案したコスト式を用いた。システムパラメタは表 5.3に示す。 CPU 性能は 100 MIPS とし、 1 クロックあたり 1 インストラクションが実行されると仮定する。表 5.3のパラメタで示すように、我々は現在入手可能な典型的は並列マシンとディスクの値を用いている。ハッシュ結合アルゴリズムのプローブ、スプリット、ハッシュは同様の研究を行っている [104] らのパラメタを参照し、他のハッシュベースの結合演算処理のコストは [98, 74] に基づいている。

このシミュレーションでは、それぞれの多重結合演算プランの特性を引き出すために、 [19] らが記述している結合グラフと同様のものを用いている。図 5.2に用いた結合グラフを示す。

実験用に５５００のデータベースを生成した。それぞれのデータベースのリレーション数は６から１２で、平均リレーション数は [104] らのデフォルト値と同じ８リレーションとする。

リレーションのサイズは主記憶の 10% から 200% まで変化する。結合属性の選択率は 0.0001% から 0.01% まで変化する。本節では、結合属性の選択率とは、二つの対象リレーションのサイズの積と中間結果のサイズの比とする。

データベースを人工的に生成するために、上述の３つのパラメタがランダムに決定去れ、ソースリレーションはすべてのディスクに均等に分割される。結合演算の結合属性はそれぞれに異なる。

5.3.2 ネットワークバンド幅が限られている場合の結果

本節では、ネットワークバンド幅が限られている場合、それが我々のアルゴリズムによって生成される結合スケジューリング木にどのような影響を与えるかについて、議論する。図 5.3は、ネットワークバンド幅が変化させた場合の最終スケジューリング木を示す。以下の議論を明らかにするため

パラメタ	値
ノード数	16
プロセッサのクロックレート	100MHz
全手記憶サイズ	32MB
ディスクのシーケンシャル読み出しのデータ転送速度	4.8MB/sec
ディスクのランダム読みだしと書き込みの転送速度	2.4 MB/sec
ネットワークの転送速度	24MB/sec
ディスクバッファ (ネットワークバッファ) からタプルを転送するための命令数	50
ソースリレーションをスプリット分割する命令数	100
結合属性でハッシュする命令数	100
ハッシュテーブルをタプルがプローブする際の命令数	500

表 5.3: アーキテクチャおよびシステムのコスト表

に、図 5.3内の大変簡単な 8 リレーションの問合せ例について考察する。すべてのソースリレーションおよび中間結果のサイズは同じとする。他の LD、 RD、 MW、 BC、 BCM のアルゴリズムはネットワークバンド幅の影響を受けないので、ここでは比較として BCM の結果を例示する。 BCM で生成されたスケジュール木も、ネットワークバンド幅の影響を明らかにするために、図 5.3内に示している。図 5.3では、 BCM の木の構成単位となるセグメントおよび BST の構成単位となるバランスシード木の部分を点線で示している。それぞれの木のコストはこれらの単位ごとの木のコストを積算したものである。

表 5.4は結果木およびそれぞれのセグメント木あるいはバランスシード木ごとの入出力コストおよびネットワーク転送コストを示す。この表では、入出力コストおよびネットワーク転送コストよりもはるかに小さい CPU コストは省いている。二つのアルゴリズムの結果は簡単に比較できるようにするために、入出力コストおよびネットワーク転送コストはそれぞれのタプルひとつ分の入出力値で置き換えている。例えば、 BCM のセグメント 1 の最初のカラムは、セグメント 1 のビルドフェーズの入出力コストであり、 3000 となっている。これは、セグメント 1 のビルドフェーズの入出力コストが 3000 タプルをディスクから読みだすコストと同じであることを意味する。また、 BCM のセグメント 1 の二番目のカラムは、セグメント 1 のビルドフェーズのネットワークコストを意味し、 600 である。これは、セグメント 1 のビルドフェーズのネットワークコストがディスクから 600 タプル読みだすコストと同じであることを意味する。

図 5.3 and 表 5.4からネットワークバンド幅が変化すると、 BST では結果の木の形状が異なっていることがわかる。このシミュレーションでは、入出力のバンド幅よりもネットワークバンド幅が五倍より大きい場合には、 BCM と BST の木の形状は同じになる。しかしながら、ネットワークバンド幅が小さくなると、 BST では、結果の木は幾つかのバランスシード木の組み合わせとなる。表

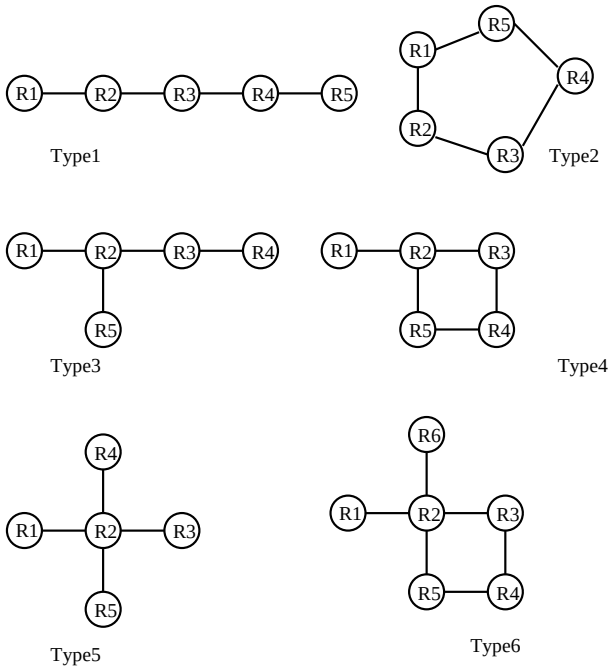

図 5.2: シミュレーションで用いられる結合グラフ

5.4に示されるように、 BST では、限られているシステム資源の消費を均衡化するため、それぞれのバランスシード木のプローブフェーズの入出力コストとネットワーク転送コストは均衡化し、入出力コストが支配的となっている。一方、 BCM のセグメント 1 のプローブコストはネットワークコストが支配的となっている。この結果、ネットワークバンド幅が細くなっても、 BST では入出力コストが支配的なバランスシード木の組み合わせにより、ネットワークバンド幅が十分にある場合と、最終木のコストは同じになる。対照的に、 BCM のコストはネットワークコストを考慮していないため、劣化する。

5.3.3 生成されたスケジュール木の品質

本節では、シミュレーションの結果を報告し、従来の手法と我々の手法を比較し、提案する手法が質のよいスケジュールプランを生成できることを示す。

(1) ネットワークバンド幅に制限がある場合

図 5.4に全てのシミュレーション結果のコストを分類したものを示す。これらの結果のコストは、最適解のコストで正規化されているため、最適解を導出した場合のコストが 1 となる。図 5.4では、各行がそれぞれの手法の結果を示し、行内の箱のサイズが導出された結果が最適解で正規化された場合にどの範囲に収まっているかを表す。ここでは、左の箱から順次、最適解 (C=1), 最適解から劣化が 10% 以内に収まっている ($1.0 < C < 1.1$), 最適解から劣化が 20% 以内、 30% 以内、 50% 以内、

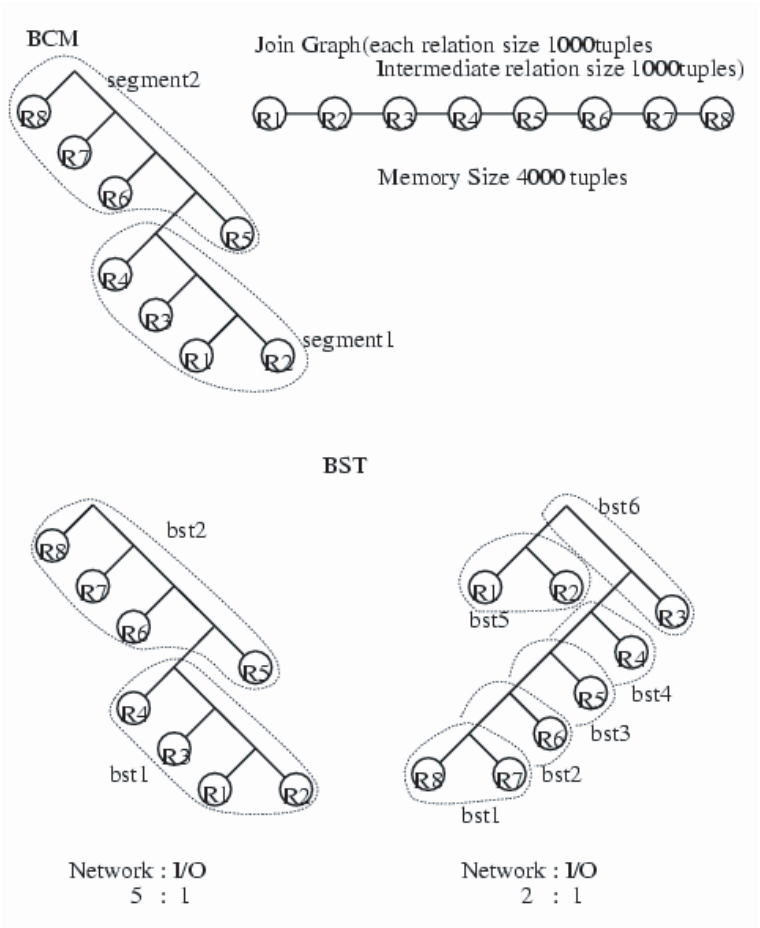

図 5.3: ネットワークバンド幅が変更された場合の木の結果

2 倍以内 (1.5 < C < 2.0)、3 倍以内、3 倍より大きい場合に分類している。最も左側の青い箱がそれぞれの手法が最適解を生成する率を示している。例として、Left Deep(LD) 手法の結果を見る。この結果から、LD では得られた解のおよそ 60% 近くが最適解であることが分かる。一方、最適解よりも 3 倍以上劣化してしまう解が約 5% あることが分かる。

この図から、我々が提案する BST では、90% の結果が最適解であることが確認できる。さらに、我々の手法で生成される最も劣化している解でも、最適解の 1.3 倍の劣化で収まっていることが分かる。従来の手法では、いずれの手法でも最適解の 3 倍よりも劣化した結果が生成されていることと比較し、提案する手法で生成される結果の質が良いことが確認できる。最適化では、最適解が得られることと同時に、悪い結果を生成しないことも重要である。図 5.4から、我々の提案する手法が悪い結果の生成を抑制することも確認できる。

これに対し、RD,MW と BC では、生成される結果の半分以上が最適解の 1.5 倍よりも大きくなっている。これは、これらの手法では中間結果を上位の部分木のハッシュテーブルとして主記憶にのせていないからである。

この図から、LD と BCM では、半分以上の結果が最適解であると同時に、それ以外の結果が大

	segment# or bst#	Network Bandwidth : I/O Bandwidth							
		5 : 1				2 : 1			
		Build		Probe		Build		Probe	
		io cost	net cost	io cost	net cost	io cost	net cost	io cost	net cost
BCM	segment 1	3000	600	1000	800	3000	1500	1000	2000
	segment 2	3000	600	2000	800	3000	1500	2000	2000
	total	9000				10000			
BST	bst1	3000	600	1000	800	1000	500	1000	1000
	bst2	3000	600	2000	800	0	0	1000	1000
	bst3	–	–	–	–	0	0	1000	1000
	bst4	–	–	–	–	0	0	1000	1000
	bst5	–	–	–	–	1000	500	1000	1000
	bst6	–	–	–	–	0	0	2000	1000
	total	9000				9000			

表 5.4: Cost of Result Trees by Varying the Network Bandwidth

変劣化していることが分かる。つまり、これらの手法では、最適解が得られない場合、大幅な処理コストが必要となる手法が生成されている。これらの理由はそれぞれの手法の方針に起因する。 LD では、リレーションサイズの小さいものをその木の最初に選択する。したがって、上位の木のソースリレーションと中間結果が大きくなりがちであり、結果として、木の上位で主記憶からソースリレーションのハッシュテーブルが主記憶から溢れる可能性が高くなる。 BCM の結果も LD と同様である。これは、中間結果リレーションのサイズが大きくなり、主記憶に収まらなくなると、全てのリレーションはネットワークコストの大きさに依らず、必ず書きもどすという方針を用いるためである。その結果、 BCM の最終的なスケジュール木の形状は LD と同じ形状となる。

(2) ネットワークバンド幅に制限がない場合

本節では、ネットワークバンド幅に制限がない場合のシミュレーション結果について報告する。ネットワークバンド幅に制限がないとは、データストリームが常に遅滞なく転送されるだけネットワークバンド幅が十分あることを意味する。この場合、我々の提案する資源消費の均衡化は、結果の生成におていさほど大きな効果を得られないかもしれない。

図 5.5はネットワークバンド幅に制限がないシミュレーション結果を示す。この図でも、図 5.4と同じく、最適解で正規化されたコストを用いている。図のそれぞれの行が、各手法の全ての結果を表し、行内の箱の横のサイズが生成された結果が最適解と比較して、どの程度の劣化で収まっているかの範囲を示している。範囲の区分は図 5.4と同じである。最も左側の青い箱が生成された結果が最適解である率を示しており、順次、右側に向かって結果が劣化していく。

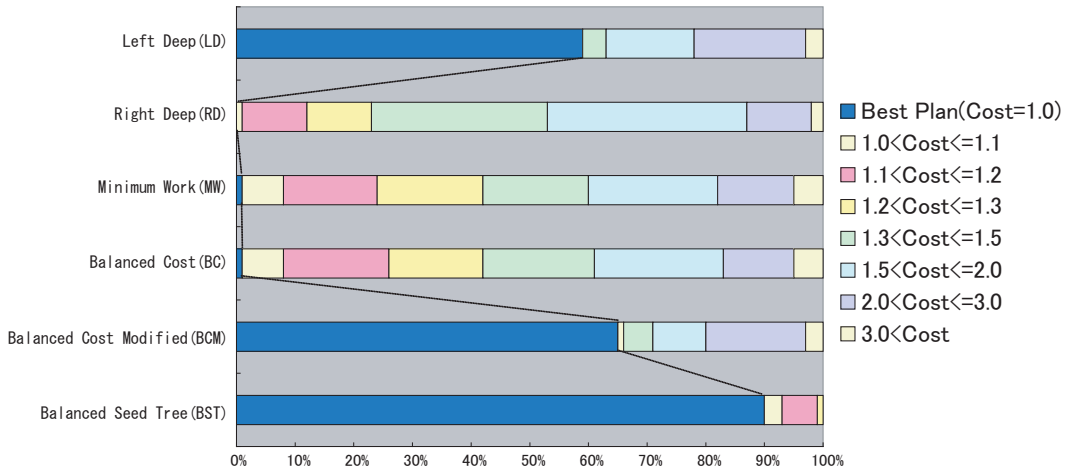

図 5.4: 生成されたプランの質 - ネットワークバンド幅に制限がある場合 -

図 5.5から, 我々が提案する手法が生成する最適解の率が図 5.4よりも低下していることが分かる。BST が最適解を生成する率は BCM とほぼ同等の率となる。しかしながら、 BCM が生成する最悪な結果が最適解の 3 倍を超えているのに対し、我々の提案する BST では最悪なプランでも最適解の二倍以内に収まっている。つまり、我々のアルゴリズムがネットワークバンド幅が制限ない場合でも有効であることを示している。これは、バランスシード木を生成する際に、ネットワークバンド幅に加え、主記憶の利用も考慮していることにより。その結果、従来の手法と比較しても、ネットワークコストと入出力コストの均衡化に加え、よい結果を得られることとなっている。

5.3.4 スケジュール木を生成するための探索空間

探索空間のサイズは多重結合演算の最適化におちえ、別の重要な要素となっている。もちろん、探索空間が大きくなれば、生成される結果木の質は良くなる。しかしながら、 [71] で指摘されているように、最適化における探索空間の爆発は結果としてアルゴリズムを非現実的なものとする。アルゴリズムを提案する主な目的はネットワークバンド幅を考慮することによる効率のよいシステム資源の利用管理であるが、本節では我々の提案する方式の探索空間が実用に耐えられるだけ十分に小さいことを示す。

表 5.5は我々の手法と全探索手法の探索空間の大きさを示す。この表の値は、シミュレーション結果の平均値を示している。表 5.5から、我々の手法の探索鵜有漢は全探索手法と比較し、十分に小さいことが確認できる。また、この表ではシミュレーションに用いた結合グラフ (図 5.2ごとの結果も示している。 type1 と type2 の結果を比較すると、 type2 の探索空間が小さくなっている。これは、 type1 のリレーションと比較して、 type2 のリレーションのサイズが大きいため、主記憶の利用という観点から生成されるバランスシード木の数が少なくなるためである。 type3 と type4 に関しては、結合グラフの縮退の方針が type1 の結合グラフよりも増えてしまうため、探索空間は大きくなる。特に結合グラフのエッジが増えると、全空間探索手法では探索区間は非常に大きくなる。しかしながら、提案

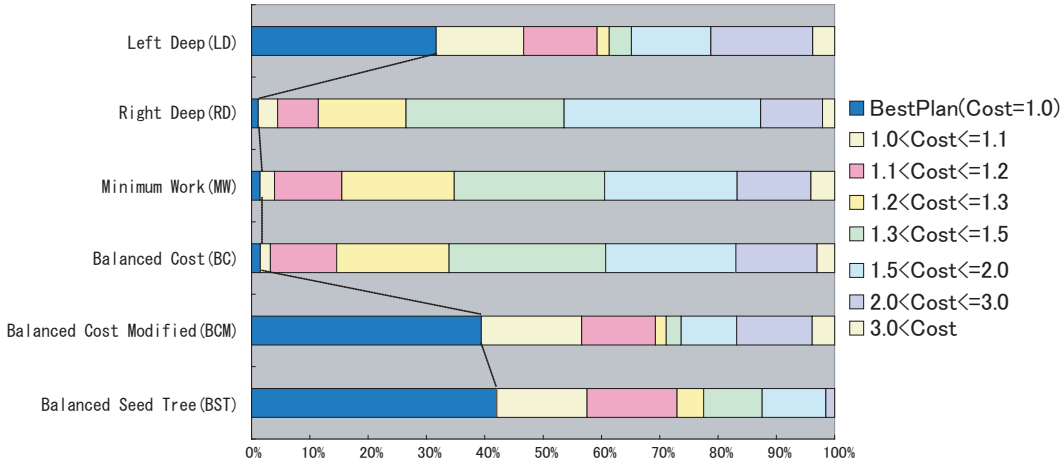

図 5.5: 生成された結果の質 - ネットワークバンド幅が制限がない場合 -

TYPE	全探索	提案する手法 (BST)	%
total	80640	235	0.29
type 1	89230	276	0.34
type 2	46080	195	0.24
type 3	92160	374	0.41
type 4	161280	377	0.23

表 5.5: 探索空間

する BST 手法では、システム資源を考慮することで候補となる木の生成を抑えることなり、結果、探索空間は全空間探索手法のように急速に大きくならない。

5.4 本章のまとめ

本章では、分散メモリ環境における多重結合演算処理について考察し、 CPU 処理コスト、ネットワーク転送、主記憶サイズ、入出力転送レートなどのシステム資源が与えられた場合に、効率良く多重結合演算のスケジューリングを決定する方式について提案した。近年の分散メモリシステムにおける大容量データベースシステム処理の観点から、多重結合演算のパイプライン処理において、大容量のデータをスムースにネットワーク転送することが重要な鍵となる。我々のアルゴリズムでは、まず、このパイプライン処理においてネットワークバンド幅を十分に利用したバランスシード木を生成する。続いて、生成されたバランスシード木の集合から互いを組み合わせることで、最適な問合せ木の生成を行う。

分散メモリシステムにおけるパイプライン処理による結合演算のコスト式を導入し、それを基に

シミュレーションにより提案した多重結合演算スケジューリング手法と従来からの手法の比較を行った。評価結果から、我々の BST 手法が従来の Left Deep 木、 Right Deep 木、セグメント right deep 木などと比較し、生成される結果木の質が良いだけでなく、劣化する率も比較的低いことを示した。特にネットワークバンド幅が限られている場合には、我々のアルゴリズムはネットワークコストを考慮しない従来の手法より優れている。さらに、ネットワークバンド幅に制限がない場合でも、最適解の生成率は従来の手法よりも良く、また、性能劣化は少なく、生成される結果の質が高いことが確認できた。我々の手法は最適解を多く得ることができる一方、探索空間も全空間探索手法と比較しても十分に現実的であることを示した。

第 6 章

分散共有メモリ計算機におけるデータベース処理に適合したバッファ管理方式

6.1 分散共有メモリ計算機と並列関係データベース処理

近年，インタネットの普及に見られるような情報基盤の急速な発展と情報化社会の成長に伴い，生成，収集されるデータは増大する一方であり，これらのデータを効率良く管理，運営するためにマルチメディアデータベース処理，意志決定支援システムなどの大規模データベースシステムが次々に開発されている．これらのシステムでは，常に肥大化するデータへの容易な拡張性，複雑化する問合せへの迅速な応答などの高性能化が求められている．例えば，意志決定支援システムのベンチマーク TPC-D の結果を見ても， 1995 年には高々 10GB 程度のデータベースサイズで計測されていたが，1997 年には 1 TB のデータベースサイズによる結果報告もあり，数年前の 100 倍以上の大きさのベンチマークが必要となっていることが伺える．

一方，高性能なマイクロプロセッサの出現，大容量メモリデバイスの普及，高速なネットワーク並びに低価格なディスクを背景に多数の商用並列計算機が開発されている．共有メモリ並列計算機は，アプリケーションの並列化，移植性の容易さから広く使用されるようになり，また，分散メモリ並列計算機も IBM 社 SP-2，富士通社 AP-3000， Intel 社 PARAGON などの多数の商用システムが稼働している．さらに，近年，共有メモリ計算機におけるプログラミングの容易性と分散メモリ計算機における台数拡張性を併せ持つ分散共有メモリ型並列計算機 [51, 2, 67, 15, 79] が着目されている．分散共有メモリ計算機は，ソフトウエアあるいはハードウエア支援によるメモリ一貫性機構を用い，各ノードに分散するメモリを一つのアクセス空間としてユーザにを提供している．そのメモリ管理機構の形態から，常にアクセスされたノードに仮想記憶空間のページがマッピングされる Cache Only Memory Architecture(COMA)，仮想記憶空間アドレスは各ノード上の物理メモリ上に割り当てられ，他ノード上に割り当てられている空間をアクセスする場合はキャッシュを通じて行い，共有されるキャッシュ空間上でメモリ一貫性を管理する Cache Coherent NonUniform Memory Architecture (CC-NUMA) に分類される．現在商用化されている代表的な分散共有メモリ計算機 (HP 社 Exemplar SPP， SEQUENT 社 NUMA-Q， SGI-Cray 社 Origin 2000， DG 社 Aviniion 等) は CC-NUMA である．

上記のような背景を基に，商用並列計算機上に多くの並列データベースシステムが実装されるようになり， TPC-D ベンチマークの結果として NCR 社 WorldMark， IBM 社 SP-2， SUN Microsystems 社 Enterprise 10000 などの並列計算機上で評価結果が報告されている．並列データベース処理の研究は共有メモリまたは分散メモリアーキテクチャにおける並列関係データベースシステムを対象とした研究は多く行われている [30, 118] が，分散共有メモリアーキテクチャ上の並列処理アルゴリズムの研究は未だ少ない [102, 10]．また，現状の分散共有メモリ並列計算機の性能評価 [14, 17] は，数値計算に見られるようなアクセス局所性の顕著なアプリケーションを対象とした解析が主であり，データベース処理のような大容量の主記憶空間をランダムにアクセスする特性を有するアプリケーションに関して詳細な解析は報告されていない． Shardal ら [102] の Shared Virtual Memory(SVM) における並列ハッシュ結合演算処理方式の検討は並列データベース処理における分散共有メモリアーキテクチャの有効性を示してはいるが，ソフトウエア支援による環境を仮定したシミュレーションであ

り，分散共有メモリに適合した並列データベース処理方式の提案，解析などは行われていない．また，その結果は現在の高速ネットワークで実現されたハードウエア支援分散共有メモリアーキテクチャには適用できない． Buganium ら [10, 11, 24] による KSR 上の並列多重結合演算処理方式の検討は，各ノード上のバッファを他ノードと共有し相互参照することで，ノード間の処理負荷のバランスを図っている．従って，分散共有メモリ計算機における共有メモリ計算機の観点を活かしたアルゴリズムの提案がされてはいるものの，実装方式に関する詳細なデータは報告されておらず，また，分散共有メモリ特有の問題であるメモリ一貫性保持のためのペナルティに対する解析はなされていない．今後巨大化する一方のデータベース処理およびユーザアプリケーションの移植性を考慮すると，比較的プログラムの実装が容易で，台数拡張性の高い分散共有メモリ並列計算機上での並列データベース処理アルゴリズムの研究，開発はこれからの重要な課題と言える．

本章では，関係データベース処理の中でも負荷の高い結合演算を対象とし，分散共有メモリ計算機上でのハッシュ結合演算処理方式について検討する．

分散共有メモリ並列計算機の主記憶空間へのアクセス特性として，一般的にローカルな空間にマッピングされているページおよび他ノード空間にマッピングされているページへの参照，書き込みのアクセス時間に大きな差がある．したがって，広大な主記憶空間全体にデータをロード，アクセスする大規模データベース処理では，単純な実装方式を採用すると，メモリ一貫性維持のためのトラヒックが増え，期待した性能が出ない．本章では，共有メモリ計算機上の実装方式をモデル化し，分散共有メモリ計算機上のハッシュ結合演算処理におけるハッシュテーブル，データバッファ，入力バッファ，出力バッファ等のデータ領域へのアクセス特性を抽出する．その結果を基に，ハッシュ結合演算処理におけるデータバッファ，ハッシュテーブル領域のノード局所性に着目した獲得方針および獲得された領域へのアクセスコストを考慮し，４種類のバッファ管理方式を提案する．新たなバッファ管理方式では，バッファの割り付けをノード局所性を意識して行うことにより，共有メモリ計算機上の処理方式をそのまま移植した場合と比較し，大幅な性能向上が期待できる．また，バッファ領域をノード毎に分割しても，実装上，これらのバッファは他ノードからの参照が可能であり，その実装方式は比較的容易である．提案された４方式を実際に分散共有計算機 HP Exemplar SPP 1600 上に実装し，性能評価を行う．性能評価の結果より，我々が提案する分散共有メモリアーキテクチャに適合したバッファ管理方式がメモリアクセスコストを大幅に削減することが可能であることを示す．さらに，メモリアクセス特性を明確化すべく，入出力を取り除いた実験に加え，ディスクの転送レートを変化させ，入出力の CPU 性能に対する相対性能を変えた場合の振舞いに関しても検討を行う．

以下，本章の構成は，第 6.2節にて分散共有メモリ並列計算機の機構およびメモリアクセス特性について述べ，第 6.3節にて並列ハッシュ結合演算アルゴリズムを簡単に紹介し，共有メモリ並列計算機における並列ハッシュ結合演算アルゴリズムのバッファ管理を比較し，第 6.2節の実測結果をもとに [86] にて提案した分散共有メモリアーキテクチャに適合したバッファ管理方式について概観する．第 6.4章にて HP Exemplar SPP 1600 上での各方式の測定結果を報告，結果の考察を行う．第?? 章では、共有分散メモリシステムにおけるキャッシュサイズ、メモリレイテンシー、ノード数等の各システム資源が変化した場合について、シミュレーションで性能評価を行い、分散共有メモリにおける

局所参照性を考慮した新たなアルゴリズムの提案を行う。第 6.6章にて本章のまとめと今後の課題について述べる.

6.2 分散共有メモリ並列計算機のメモリアクセス特性

本節では，実測に用いた分散共有メモリ型並列計算機 HP Exemplar SPP 1600 のアーキテクチャおよびそのメモリアクセス特性について述べる.

6.2.1 分散共有メモリ型並列計算機：HP Exemplar SPP 1600

本実験では， HP Exemplar SPP 1600[22](SPP 1600) を並列データベース処理方式のモデル化のプラットフォームとして用いた.

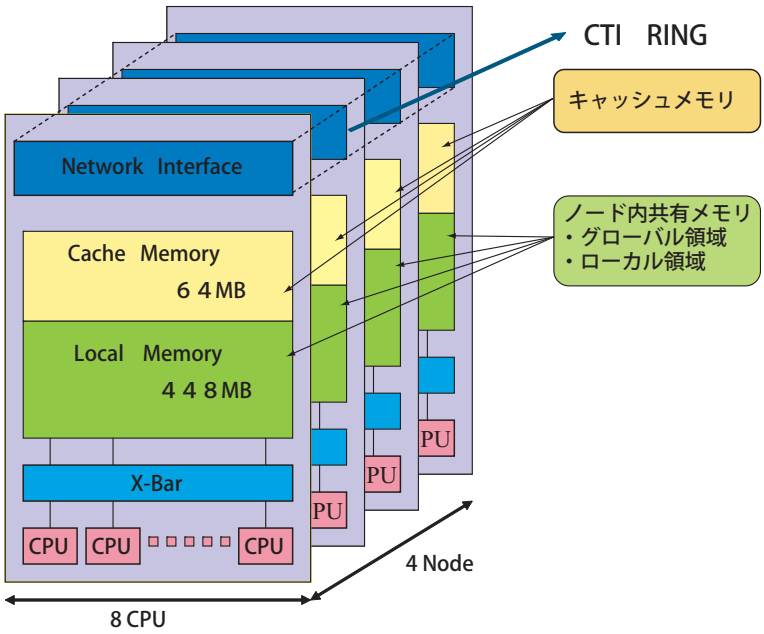

図 6.1: 本実験で用いた SPP 1600 の構成図

SPP 1600 は 8 CPU(PA-RISC 7200, 120MHz) をクロスバスイッチにより密に結合し一つのノードを構成し，各ノード間を高速ネットワークを用いて結合している． SCI プロトコル [84] を用いたメモリ一貫性処理により， 64byte のキャッシュライン単位で分散共有メモリ機構が実現されている．本実験に利用したマシンは 4 ノード構成で，各ノードに 512MB のメモリが実装されており，全体では 32CPU と 2GB のメモリからなる．このうち，各ノードごとに 64MB が下記のネットワークキャッシュとして用いられている． SPP 1600 では，ユーザのアクセスするメモリは大きく次の 3 つのクラスに分類される.

1. グローバルメモリ

全ノードから参照可能な領域であり，どのノード上の物理メモリに割り当てられているかユーザからは特定できない．各ノードの物理メモリ上に 4KB を１ページとして均等に割り当てられる．

2. ローカルメモリ
特定のノードに明示的に割り付けられたメモリ領域であるが，他ノードからも参照可能．したがって，ユーザはノード内でのローカルなアクセスを意図的に行える．

3. ネットワークキャッシュ (以下では単にキャッシュ)
他ノード上の物理メモリにマッピングされている空間をアクセスする際に利用されるキャッシュ領域であり，ユーザからは見えない．

6.2.2 SPP 1600 のメモリアクセス特性

本節では， SPP 1600 上のメモリアクセス特性として，単純なアクセスプログラムを用い，ローカルメモリ，グローバルメモリに対する参照，書き込みコスト，メモリ一貫性処理により生じる無効化処理等のペナルティコストを調べ，その結果を表 6.1に示す．メモリアクセスプログラムは， 4 バイト整数の読み書き (Read/Write) を 64 バイトのキャッシュラインおきに 64MB の連続なアドレス空間上で行うものである．

表 6.1の項の意味を以下に説明する． Mapping は，アクセス対象となる空間がアクセスするノードからみて，自ノード内メモリに割り付けられているか (Local)，他ノードのメモリに割り付けられているか (Remote) を示す． Invalidation， Fault はアクセス時のデータ領域の状態を示す． Invalidation は， read 時にはアクセス対象となるメモリが他ノードによりすでに書き込みが行われていること意味し， write 時には他ノードに参照されていることを意味する． Fault は， write 時にアクセス対象となるメモリが他ノードによりすでに書き込みが行われていることを意味する． Lock は，アクセス前に各ノード間でロック命令を用いた排他制御をとる場合の書き込み時間，つまり，ロック命令からアンロック命令までの時間を意味する． Cache hit は，他ノード上のメモリアクセス時に，すでにデータがアクセスしようとしているノード上のキャッシュにあるか否かを示す． Cost は，１キャッシュラインのアクセス時間 (μsec) を示す．

メモリ参照のコストは，ローカルメモリ内のアドレス空間あるいは，キャッシュにヒットしているときは， 0.8 μsec となる．一方，他ノード内のデータ参照は 4.0 μsec と重いが，ローカルノード内のメモリであっても，他ノードからの書き込みが行われた (つまり，メモリ一貫性処理が起きた) 場合，他ノードからの参照と同じ程度のコストがかかることがわかる．書き込みに関しては，ローカルメモリの場合には参照コストと同じコストである．しかし，キャッシュヒットしている他ノードメモリへの書き込みは，参照コストと比較すると，約２．５倍のコストがかかる．キャッシュミスの場合には，他ノード上のメモリへの書き込みは他ノードのメモリ参照とほぼ同じ程度のアクセスコストがかかる．また，ロック命令を伴うアクセス時間に関しては，同期オーバヘッドが非常に大きいことがわかる．

Mapping	Access	Invalidation	Fault	Lock	Cost(μsec)
Local	read	No	No	No	0.8
Local	read	Yes	No	No	5.1
Local	write	No	No	No	0.8
Local	write	Yes	No	No	5.8
Local	write	No	Yes	No	5.1
Local	write	No	No	Yes	6.0
Local	write	Yes	No	Yes	16.8

Mapping	Access	Invalidation	Fault	Lock	Cache hit	Cost(μsec)
Remote	read	No	No	No	Yes	0.8
Remote	read	No	No	No	No	4.0
Remote	read	Yes	No	No	No	4.6
Remote	write	No	No	No	Yes	2.0
Remote	write	No	No	No	No	4.0
Remote	write	Yes	No	No	No	5.8
Remote	write	No	Yes	No	No	4.6
Remote	write	No	No	Yes	Yes	12.2
Remote	write	No	No	Yes	No	14.5

表 6.1: SPP 1600 におけるメモリアクセス性能特性

6.3 並列ハッシュ結合演算処理

Mishira らの報告 [81] に見られるように，ハッシュに基づく結合演算アルゴリズムの研究が盛んに行われてきた．ハッシュ結合演算は，ソースリレーションを結合属性を用いてあらかじめハッシュ分割することにより結合演算の突き合わせコストを大幅に削減することができる．また，ハッシュ分割することにより，各々のパーティションは互いに異なった値を有するため，個々のパーティションごとに独立に処理でき，容易に並列実装することができる．並列ハッシュ結合演算アルゴリズムの研究も多いが，これらの研究では並列環境が共有メモリあるいは分散メモリアーキテクチャであることを想定しており，分散共有メモリアーキテクチャ上での詳細な性能評価は筆者らの知る限り，検討されていない．

本節では，共有メモリ環境における並列ハッシュ結合演算処理を基に，並列ハッシュ結合演算処理のバッファモデルとメモリアクセスについて考察する．以下では，次の二つリレーション R， S を用いたハッシュ結合演算処理を考える．

SELECT * FROM R,S WHERE R.joinkey = S.joinkey

ハッシュ結合演算として，Grace ハッシュ結合方式, ハイブリッドハッシュ結合方式，動的 Grace ハッシュ結合方式などが提案されているが，いずれの方式でも，主記憶上に保持できない大規模リレーションの処理を行う場合，二次記憶上のデータを一旦スプリット関数を適用して分割し，二次記憶上にバケットを動的に生成することで入出力コストの削減を目指している．スプリット関数適用時に主記憶を有効利用するかどうかにより，各方式の処理の流れが異なるが，本節では，Grace ハッシュ結合演算処理を基に考察する．Grace ハッシュ結合演算では，まず，スプリット関数を用いて主記憶に収まるサイズのバケットにソースリレーションを分割し，一旦，ディスクに書き戻すバケット分割フェーズを行う．さらに，生成されたバケット毎にハッシュテーブルを生成し，プローブ，結合処理を行う結合フェーズをすでに分割されているバケット数分繰り返す．バケット分割フェーズは生成するバケットの個数に対応したページバッファを用意し，書き戻しを行うので，主記憶空間全体のアクセス頻度は少なく，分散共有メモリアーキテクチャを特に意識せずとも問題がないと予想される．しかしながら，結合フェーズでは，主記憶全体にバケットのデータをロードするため，他ノード間にまたがる参照が非常に多くなり，分散共有メモリ環境ではその性能に大きな影響があると思われる．したがって，本章では，ビルドリレーション (ビルドフェーズ時にハッシュテーブルの生成対象となるリレーション，ここではリレーション R とする) が主記憶に収まる大きさ，あるいは，すでにバケットに分割されたことを前提として，結合フェーズのみを対象として処理のモデル化を行う．

以下の節では，共有メモリ環境における並列ハッシュ結合演算処理 (Shared Everything 方式) モデルを図示し，これを基に分散共有メモリ環境におけるバッファ領域の獲得，処理プロセスからのバッファ領域へのアクセスを分類する．続いて，この分類に基づき，新たなバッファ管理方式を提案する．

6.3.1 並列ハッシュ結合演算処理モデル

分散共有メモリ環境におけるハッシュ結合演算アルゴリズムの処理の流れを図 6.2に示す．

前述のように処理の流れは，リレーション R からハッシュテーブルを生成するビルドフェーズとリレーション S によりハッシュテーブルをプローブするプローブフェーズの二つのフェーズからなり，それぞれの処理をビルドプロセス，プローブプロセスが行う．また，ディスクからの入出力は入出力プロセスが行うが，図が繁雑になるため，ここでは省略する．ビルドフェーズでは，リレーション R がディスクからデータバッファへ読み込まれる (図 6.2: Build Phase(1))．続いて，結合属性にハッシュ関数を適用し (図 6.2: Build Phase(2))，ハッシュ値により該当するハッシュテーブルからタプルをリンクする (図 6.2: Build Phase(3))．ビルドリレーションのタプルはプローブ時に結合値の突き合わせのために必要であり，メモリ上のデータバッファに保持される．プローブフェーズではリレーション S がディスクから読み出され (図 6.2: Probe Phase(4)), 各タプルの結合属性に対しハッシュ関数を適用し (図 6.2: Probe Phase(5))，該当するハッシュ値のハッシュテーブルエントリにリンクされているビルドリレーションのタプルをプローブし，結合属性が一致した (図 6.2: ProbePhase(6)(7)) 場合には，結果が生成される．

図 6.2からわかるように，並列ハッシュ結合処理ではハッシュテーブル，ビルドリレーションおよ

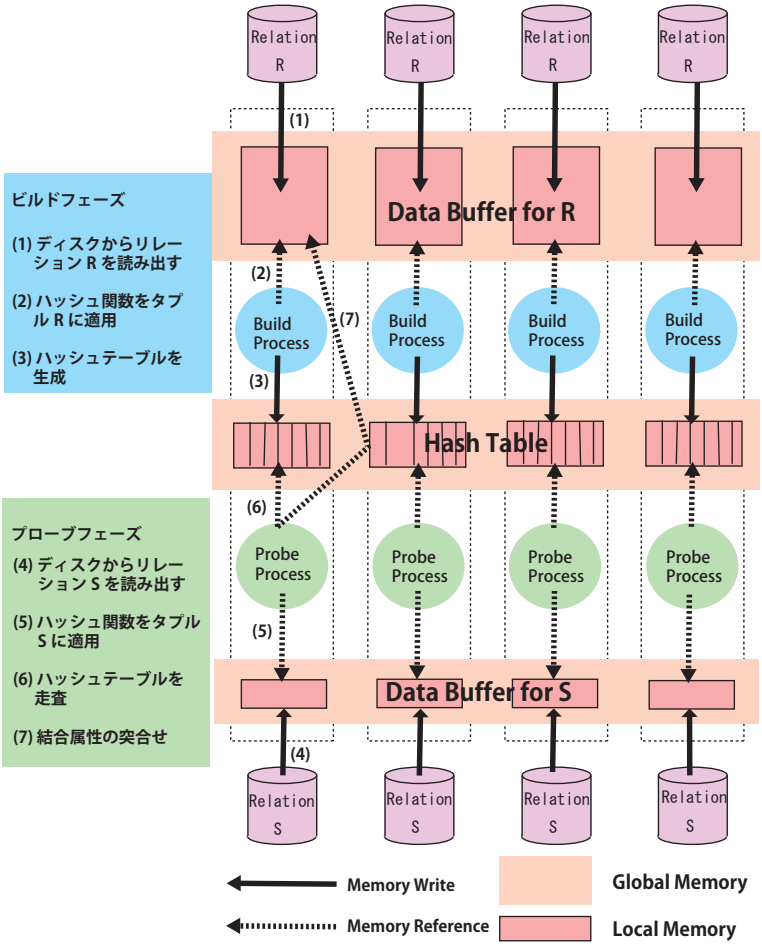

図 6.2: 並列ハッシュ結合演算処理モデル

びプローブリレーションを格納するためのバッファ領域が主としてアクセスされる領域であり，入出力，ビルド，プローブプロセスの 3 つのプロセスから，それぞれの領域に対し，アクセスが行われる．共有メモリ上であれば，各プロセスからのデータ領域へのアクセス時間は一定であるが，分散共有メモリ上では， 2.2 節で示したように，データ領域がどのように物理メモリ上にマッピングされるかにより，アクセスコストが大きく変化する．次節では，上記の 3 つのデータ領域：ハッシュテーブル，ビルドリレーションデータバッファ，プローブリレーションデータバッファと， 3 つのプロセス：入出力プロセス，ビルドプロセス，プローブプロセスから構成される並列ハッシュ結合演算処理モデルを基にメモリアクセスコストを，分散共有メモリの観点から考察する．

6.3.2　バッファ獲得方針とアクセス方式の分類

分散共有メモリ計算機 SPP 1600 では，ローカルメモリを用いることで，ノード局所性を意識し，各ノード毎に個別にデータ領域を獲得することが可能である．例えば，実際に読み出すリレーションは各ノードのディスクに格納されていることから，リレーションのためのデータバッファ領域を個々

	Data Area	SE	SHT	LHT	LHT–R
	Data Buffer	Global	Local	Local	Local
	Hash Table	Global	Global	Local	Local
Process	behavior	SE	SHT	LHT	LHT–R
I/O Proces	Build Phase(1) Data Read	inter node	intra node	intra node	intra node
Build Process	Build Phase(2) Data Reference	inter node	intra node	inter node	inter node
Build Process	Build Phase(3) Hash Table Generation	inter node	inter node	intra node	intra node
I/O Process	Probe Phase(4) Data Read	inter node	intra node	intra node	intra node
Probe Process	Probe Phase(5) Data Reference	inter node	intra node	inter node	intra node
Probe Process	Probe Phase(6) Hash Table Reference	inter node	inter node	intra node	inter node
Probe Process	Probe Phase(7) Build Data Reference	inter node	inter node	intra node	inter node

表 6.2: 4つのバッファ管理方式

のノード毎に獲得することは自然である．同様に，ハッシュテーブル領域を各ノード上に個々に生成することも可能である．また，このバッファ割り当てを処理プロセス側から見た場合，ノード内のみのローカル領域へのメモリアクセス (intra node) か，他ノードへのメモリアクセス (inter node) となるかに分類できる．

図 6.2におけるデータ処理の流れを基に，ハッシュテーブルをグローバルまたはローカルに獲得した場合およびデータバッファをグローバルおよびローカルに獲得した場合の各プロセスのメモリアクセス方式を考える．

データバッファ領域へのアクセスは，主として図 6.2内の Build Phase(1) と Probe Phase(4) において，入出力プロセスによるディスクからのリレーションの読み込み時における書き込みとその後のビルドプロセス，プローブプロセスによるデータ参照がである．ここでは，各ノード毎にディスクが接続されている環境を想定する．入出力プロセスは基本的にデータバッファへの書き込みのみである．したがって，リレーションを読み込むバッファは，グローバルに領域を獲得するより各ノード毎にローカルに獲得した方が効率が良い．そこで，最も単純な方式 (SE 方式) ではデータバッファはグローバルに獲得するものとするが，他の方式では各ノードのローカル領域に獲得するものとする．ハッシュテーブル領域も分散メモリ環境の並列ハッシュ結合演算の実装で採用されているようにビルドフェー

ズ時に各ノードごとにローカルに生成することが可能である．以上のことから，組合せとしては，ハッシュテーブルをグローバルメモリ上に獲得しデータバッファをローカルメモリ上に獲得するか，あるいは，ハッシュテーブル，データバッファ共にローカルメモリ上に獲得するかの２つの場合が考えられる．

領域の割り付けが決まると，各プロセスからのアクセス手法が決定される．例えば，データバッファ，ハッシュテーブルをともにローカルに獲得する場合を考える．ビルドリレーションを取り込む際にノード内のバッファに書き込み，他ノードから参照させる場合と，他ノードのバッファに書き込み，それを当該ノードからローカルに参照する２通りのアクセス方法が考えられるが，表 6.1からわかるようにいったん他ノードで書き込まれた領域を自ノードで参照するのは，明らかにコストが高い．

上記の検討を基に，ノードを意識したバッファ領域確保とコストを考慮したアクセス方式の組合せとして，表 6.2に示すように， Shared Everything(SE) 方式， Shared Hash Table(SHT) 方式，Local Hash Table(LHT) 方式および Local Hash Table with Remote reference(LHT-R) 方式の４方式を提案する．以下で，各バッファ管理方式の詳細を述べる．

(1) SE 方式

前節にて概略を説明した SE 方式ではハッシュテーブル，データバッファ共にグローバルメモリ上に確保されている．従って，ハッシュテーブルへのアクセス，データバッファの管理とも一元化されており，従来の共有メモリ計算機上での実装をそのまま流用できる．

図 6.3に， SE 方式の処理の流れを示す．

ビルドフェーズ　ディスクから読み出されたデータは，入出力プロセスにより，バッファプールから獲得されたページに書き込まれる (図 6.3: Build Phase(1))． SE 方式では，ビルドリレーション保持用のデータバッファをグローバル領域に獲得しているため，ノード間アクセスが頻繁に生じる．つぎに，書き込まれたデータバッファはビルドプロセスにより参照され，ハッシュ関数が適用される．このとき，あるノードの入出力プロセスにより書き込まれたデータバッファを同じノード上のビルドプロセスが参照するとは限らないため，やはり，他ノードへの参照が生じる (図 6.3: Build Phase(2))．各ビルドプロセスはタプルのハッシュ値により，該当するハッシュテーブルにタプルを登録するため，書き込みを行う (図 6.3: Build Phase(3))．この際，ハッシュテーブルの参照，書き込みでも他ノードへのアクセスを生じる．また，バッファプールからデータバッファの獲得，データキューへの登録およびハッシュテーブルへの書き込みではノード間での排他制御が必要である．

プローブフェーズ　ディスクから読み出されたデータは，ビルドフェーズ同様，グローバルメモリ上のデータバッファに書き込まれる (図 6.3: Probe Phase(4))．プローブリレーションは結合演算処理後は不要なため，獲得されたデータバッファはページ内の全タプルの処理が終ると，バッファプールに返される．

プローブプロセスはデータバッファを取り込み，タプルごとにハッシュ関数を適用し，該当する

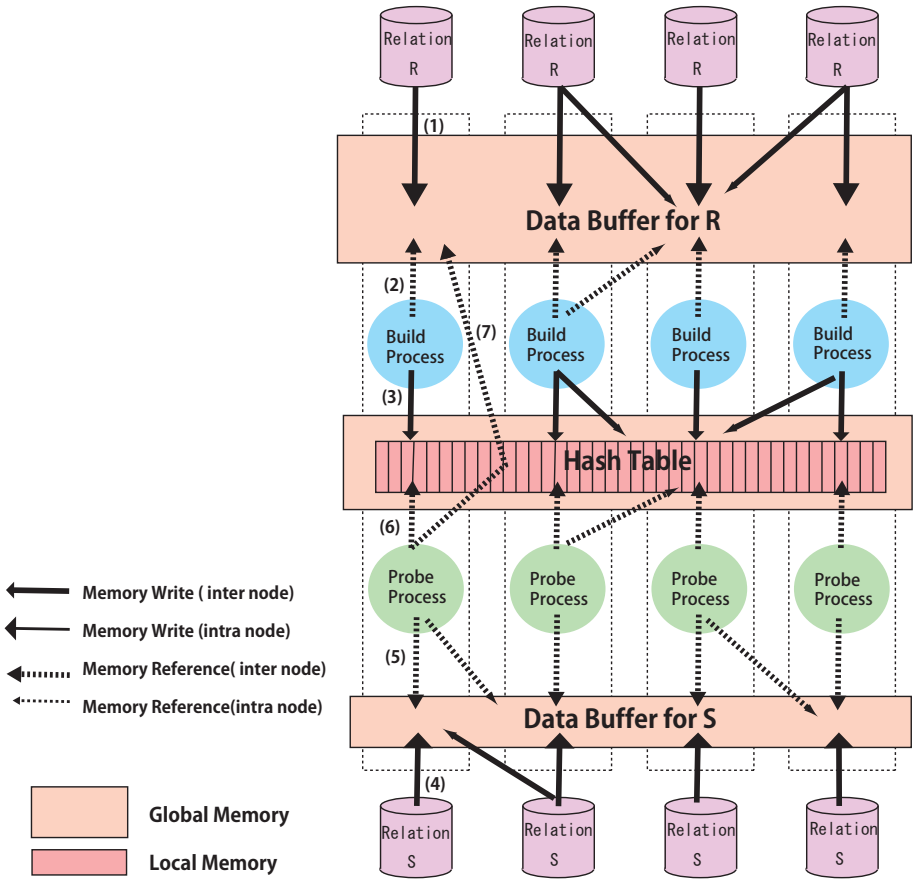

図 6.3: Shared Everything (SE) 方式

ハッシュテーブルをプローブする．ビルドリレーションのタプルと結合属性値が一致した場合には結合演算処理を行う．したがって，プローブリレーションのデータバッファ参照，ハッシュテーブルの参照，ハッシュテーブルからリンクされるビルドリレーションのデータバッファの参照によって，他ノードへのアクセスが生じる．この際，データバッファの獲得，返却などの管理にはノード間の排他制御が必要である．

(2) SHT 方式

SE 方式では，共有メモリ環境における実装方式をそのまま分散共有メモリに移植しているため，ノード間並列処理が容易に実現されるが，同時に，すべての領域をグローバルメモリに獲得しているため，他ノード上のメモリ参照，書き込み，ノード間での排他制御が多くなり，メモリアクセスコストが非常に重くなる．

実際，各ノードごとのディスクに均等にデータが格納されている場合，各ノードが読み出すデータ量およびビルドプロセスが処理するコストは各ノードで均等であるため，データバッファをグローバル領域に獲得する必要はない．そこで，ハッシュテーブルはグローバル領域に獲得し，データバッ

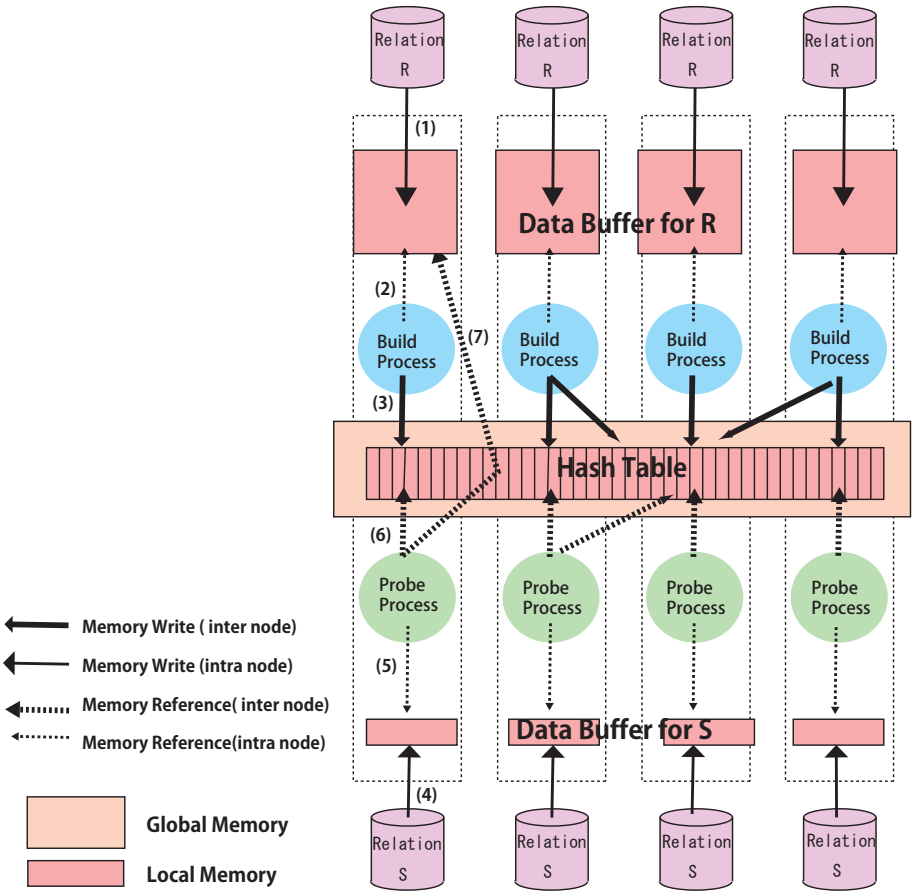

図 6.4: Shared Hash Table (SHT) 方式

ファをノードのローカルメモリに獲得することにより，データ読み出し時のノード間アクセスを削減することができる．この方式では，ハッシュテーブルへのアクセスは各ノード間で共有できるため，ノード間の処理負荷のバランスを容易に実現できるという共有メモリ環境の特徴をそのまま受け継いでいる． Shardal[102] らが提案しているソフトウエア SVM 環境での方式も同様の方式である．ハッシュテーブルのみをノード間で共有して参照，書き込みすることから， Shared Hash Table(SHT) 方式と呼ぶ．

本方式の実装では，データバッファを個々のノードで管理するため，データバッファへのアクセスに関してノードを意識した実装が必要となり， SE 方式より複雑になる．つまり，ハッシュテーブルの排他制御は一元化されているが，データバッファの管理は，個々のノードごとに行わなくてはならない．

ビルドフェーズ　ビルドリレーションのためのデータバッファはノード内のローカルメモリ上に獲得される．入出力プロセスは各々のノードのディスクからタプルを読み出しデータバッファに書き込む．同じノード上のビルドプロセスが，書き込まれたデータバッファを参照し，ハッシュ関数を適用する．

さらに，ビルドプロセスはグローバルメモリ上のハッシュテーブルにそれぞれのタプルをリンクする．したがって， SE 方式とは異なり，データバッファの書き込み，参照時には他ノードへのアクセスは起きない．一方，ハッシュテーブル作成はノード間をまたがって行なわれる．また，ハッシュテーブルのアクセスには，ノード間での排他制御が必要となる．

プローブフェーズ　データバッファがローカルメモリ上に確保され，結合演算処理後はプローブリレーションを保持しておく必要が無いためデータバッファはバッファプールへ返却される．入出力プロセスはディスクから読み出したページをローカルメモリ上のデータバッファに書き込む．バッファが一杯になるとそのバッファを自ノード上のプローブプロセスが管理するデータキューへリンクする．

プローブプロセスはデータキューからバッファを取り込み，グローバルなハッシュテーブルをプローブし，結合演算処理を行う．この際，ハッシュテーブルの検索，ハッシュテーブルからリンクされるビルドリレーションへの参照は他ノードへのアクセスを生じる．処理を終えたデータバッファはバッファプールへ返却される．

(3) LHT 方式

グローバルメモリ上に獲得されたハッシュテーブルを各ノード上のビルドプロセスがアクセス場合，ロックを用いた排他制御を伴う他ノード参照となるため，メモリアクセスコストは高い．そこで，ハッシュテーブルを各ノードにローカルな領域に振り分け，それぞれのノードごとに分散して処理を行う．この方式では，分散メモリ環境と同様に，それぞれのノード間で各ノードのハッシュテーブルに対応するデータバッファを転送することで実現される．すなわち，ハッシュテーブル領域，データバッファ領域は共にローカルメモリ上に確保される．つまり，ハッシュテーブル自体もハッシュ分割されて，各ノード毎にローカルなテーブルとして割り振られ，それぞれのテーブルのエントリは重ならないものとする．

本方式の実装では，ローカルに獲得したデータバッファに対し他ノードからのアクセスも可能とするため，ノード数だけのキューをそれぞれのノードが管理しなくてはならない．そのため， SHT 方式よりも，さらにデータバッファ管理は複雑になる．また，ハッシュテーブルは他ノードからアクセスされることはなく，ノード内のプロセスのみがアクセスするため，排他制御を個々のノード内で実現するだけでよい．

また、分散メモリ環境上では、他ノード上の物理メモリはいずれも異なるアドレス空間となるため、他ノード上のハッシュテーブルなどは通信ライブラリィを介してアクセスすることになる。しかしながら、分散共有メモリ環境では、他ノード上のハッシュテーブル等も同じアドレス空間にあるため、プログラミング上は単なるメモリ参照として表現できるため、分散メモリ環境で通信制御を行う場合と比較し、非常に容易に実装できる。

ビルドフェーズ　入出力プロセスによって読み出されたタプルはハッシュ (ノード間スプリット) 関数によってどのノードのハッシュテーブルに属するかを決定され，該当するデータバッファに書き込

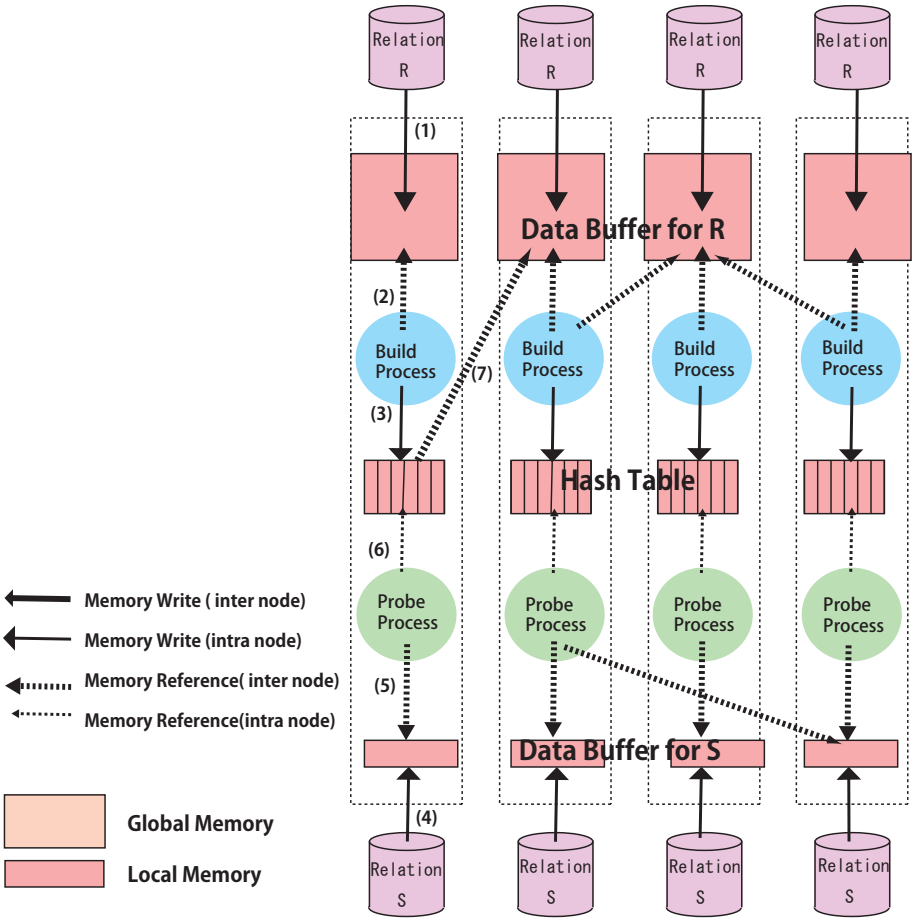

図 6.5: Local Hash Table (LHT) 方式

まれる．つまり，データバッファはノード数分に区分されている．書き込み自体はノード内ローカル領域へのアクセスとなる．

ビルドプロセスは，各ノード上のデータバッファのうち，自ノードのハッシュテーブルに該当するバッファの内容を取り込み，タプルごとにハッシュ関数を適用し，自ノード上のローカルなテーブルに登録する．つまり，タプル参照時には，他ノード上のデータバッファを参照するが，ハッシュテーブルへの書き込みはノード内にローカルに行われる．また，ハッシュテーブルへの書き込みは，それぞれのノードごとに行われるため，ノード間における排他制御の同期オーバヘッドは生じない．

プローブフェーズ　入出力プロセスはビルドプロセスと同様に動作する．プローブプロセスは該当するデータバッファからデータを取り込んだ後，ノード内のテーブルをプローブし，プローブ処理を行う．ハッシュテーブルの参照自体はノード内アクセスとなり，また，ビルドリレーションの参照自体もビルドフェーズ時にキャッシュされているため，キャッシュの大きさが十分あればほぼローカルアクセスと同じコストになることが期待される．

プローブ処理が終わったデータバッファは，書き込みが行われたノード，つまり，物理的に割り

当てられているノードのバッファプールに返却され，また，新たにデータを書き込まれる．したがって，ビルドフェーズでは，リレーションの読み込みによるデータ書き込みはローカルに行われる．一方，プローブフェーズでは他ノードが参照したページへ入出力プロセスが書き込みを行うため，頻繁にメモリ一貫性処理が生じることになり，2.2 節の結果から分かるようにアクセスコストは，リモートアクセスコストと同じになる．

(4) LHT-R 方式

前述の LHT 方式では，ハッシュテーブルをノード内にローカルに生成し，プローブフェーズ時にもローカルに参照していた．しかしながら，LHT 方式では，プローブフェーズ時に他ノードに参照されたデータバッファに新たに書き込みを行うため，メモリ一貫性処理が起こり，ローカルにデータバッファを獲得したことが活かせない．そこで，本方式では，プローブ時にはハッシュテーブルへの書き込みは行わず，参照のみであることに着目し，プローブ時のデータバッファの書き込み，参照をローカルノード内で行い，他ノードのハッシュテーブルを参照することにする．

本方式の実装では，LHT 方式と異なり，フェーズごとのデータバッファの管理に異なる実装が必要である．しかしながら，プローブフェーズでのデータバッファ管理はノード内参照のみであるため，SHT 方式同様に単純である．一方，他ノードからのハッシュテーブルの参照を許している為，全ノードのハッシュテーブルのアドレスを各ノードが保持しなくてはならない．

本方式では，ビルドフェーズは，LHT 方式と同じであり，プローブフェーズはほぼ SHT 方式と同様の流れとなる．つまり，まず，入出力プロセスがデータの読み出し時に各ノードごとにスプリットせず，ローカルなデータバッファに書き込む．プローブプロセスはそのデータバッファからタプルを読み出し，キーの値を基に該当するハッシュテーブルをプローブ，結合演算処理をする．

6.4 バッファ管理方式の性能解析

我々が提案した方式の有効性を検討するために，SPP 1600 上で４種類のバッファ管理方式を実装し，その処理時間を計測，評価した．以下では，測定環境を簡単に説明し，ビルドフェーズ，プローブフェーズおよび全体の測定結果を示す．

6.4.1 測定環境

今回の測定では，それぞれのノード毎に入出力プロセス，ビルドプロセス，プローブプロセスを１ CPU ずつに割り当て，４ノードを用いて測定した．リレーション R と S のサイズは，各々 8MB ～ 640MB まで変化させ，各ノードごとについているディスクに均等に格納し，データ分布は均一とした．ページサイズは８ KB とし，タプル長は２５６バイトとした．また，メモリアクセス時に各ノードからのアクセスが公平になるよう，それぞれのノードのキャッシュはあらかじめグローバル領域をランダムにアクセスすることで初期化されている．以下の結果では，それぞれの方式のメモリアクセ

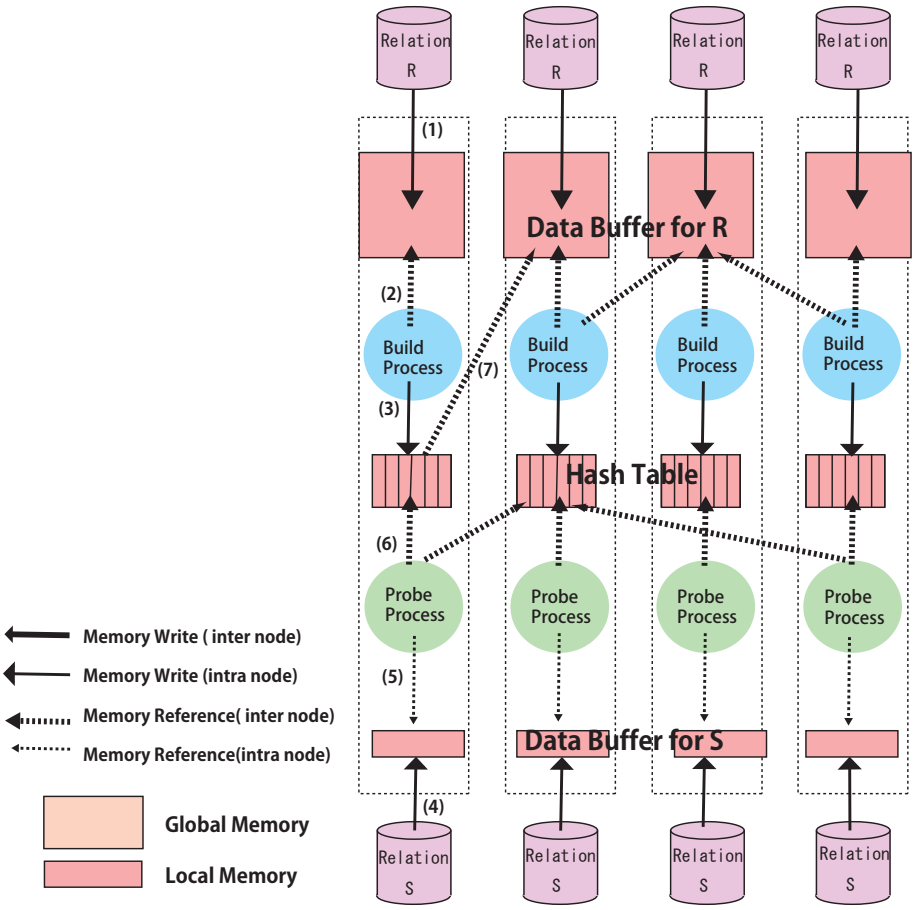

図 6.6: Local Hash Table with Remote Reference (LHT-R) 方式

ス時間の差を明確にするために，まず，各方式毎のディスクアクセス時間を除いた CPU 処理コストの結果を示し，方式ごとの特徴について検討を行う．続いて，入出力コストも含めた性能解析を行う．

6.4.2 CPU 処理コスト

CPU 処理コストの結果に関し，まず，各方式の全体の処理時間を，続いてそれぞれのフェーズごとの処理時間を示し，フェーズごとの分散共有メモリのメモリアクセス特性による処理性能への影響について検討する．

(1) 全体の処理時間

図 6.7に，全体の処理時間を示す．縦軸は処理時間を，横軸は 1 リレーションのサイズを表す．図 6.7からわかるように，採用するバッファ割り当て方式およびアクセス方式により大きく性能が異なる．まず，単純な SE 方式では，十分な処理性能が得られないことが確認できる．さらに， LHT-R 方式が最も性能がよく， SE 方式と比較して，約 58% 以上の性能向上がみられることが確認できる．また，ハッシュテーブルをグローバル領域に確保する SE， SHT 方式よりも，ハッシュテーブル

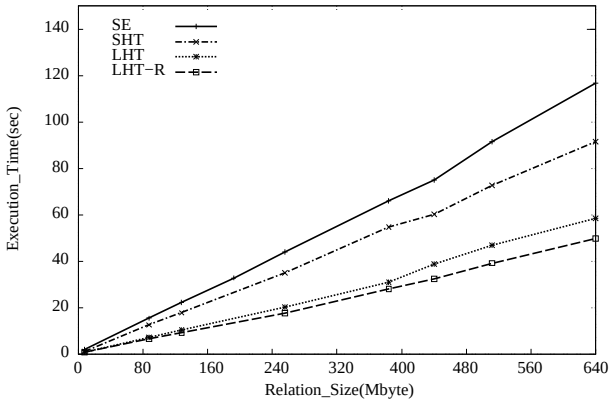

図 6.7: CPU 処理コスト：全処理時間

を各ノードごとに分割して割り付ける LHT， LHT-R 方式が性能が良いことがわかる．以下，ビルドフェーズおよびプローブフェーズごとの処理時間を示し，各方式の性能差について検討を行う．

(2) ビルドフェーズ

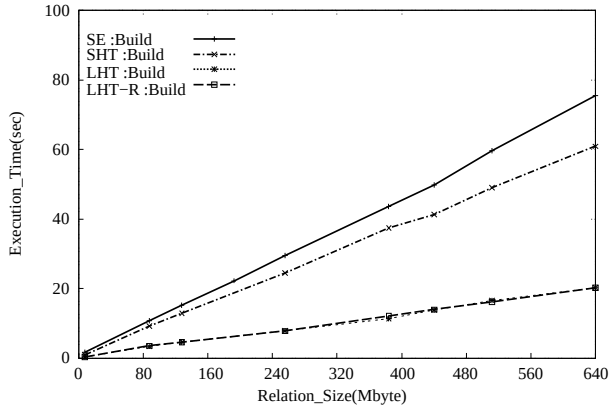

図 6.8: CPU 処理コスト：ビルドフェーズ処理時間

ビルドフェーズのみの処理時間を図 6.8に示す．図 6.8から分かるように，ハッシュテーブルをローカルに生成する LHT， LHT-R 方式が SE， SHT 方式と比較して大きく性能向上がみられる．これは，ハッシュテーブルを分割することで各ノードからの書き込み時の排他制御が不要なこと，および，ノード内のローカルな領域への書き込みになることによる． LHT 方式および LHT-R 方式のビルド処理のコストは，同じ処理を行っているため，重なっている．

SE 方式および SHT 方式の差は， SE 方式ではデータバッファもグローバルに確保されているため，入出力プロセスが他ノードメモリ上への書き込みをを行うことになり，書き込みコストが高くなることによる．さらに，ビルドプロセスで獲得するデータバッファがそのノード上で書き込まれたものとは限らないため，データ参照も他ノード参照のコストになるため，さらにコストがかかる．

SHT 方式と LHT, LHT-R 方式の差は，ビルドプロセスにおけるハッシュテーブル生成時に他ノード上への書き込みが生じること，ハッシュテーブルへの書き込みの同期をとることによる．特に，キャッシュラインにはハッシュテーブルエントリが 8 つ格納されているため，各ノードから均等にアクセスされるため，メモリ一貫性処理が頻繁に起きる．

通常の書き込みであると，自ノードへの書き込みは約 $0.8\mu sec$，他ノードへの書き込みは約 $4\mu sec$ であるが，メモリ一貫性処理が生じる場合でも約 $5.8\mu sec$ にしかならない．しかし，同期をとるためにロックを用いると，自ノードへの書き込みが約 $6.0\mu sec$，他ノードへの書き込みが約 $14.5\mu sec$ となる．このためテーブルへの書き込みコストが増大し処理時間が著しく増大する．またハッシュテーブルへの書き込みは，タプルごとに排他制御が必要なため，非常に大きなオーバヘッドとなる．

(3) プローブフェーズ

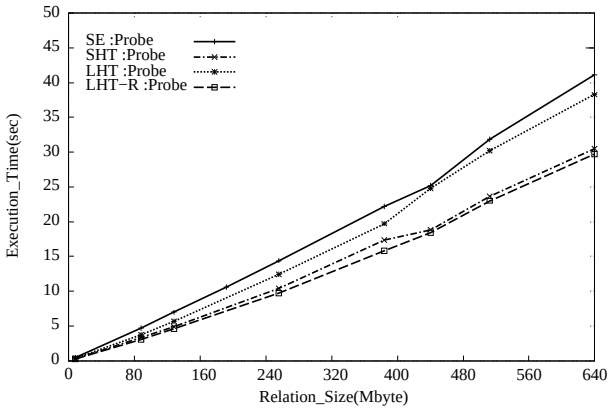

図 6.9: CPU 処理コスト：プローブフェーズ処理時間

プローブフェーズの実行結果を図 6.9に示す．ビルドフェーズと比較して，いずれの方式の処理時間も速い．これは，ビルドフェーズでは，ハッシュテーブルの生成を行うための書き込みコストが大きいが，プローブフェーズでは，ハッシュテーブルへのアクセスは参照のみであることによる．図 6.9からわかるように， SHT および LHT-R 方式が SE， LHT 方式と比較して性能が良い．

ハッシュ結合演算ではビルドフェーズリレーションはすべてのプローブリレーションのマッチングが終わるまでメモリ上に保持しなくてはならないが，プローブリレーションは一旦マッチングが終われば必要ないため，プローブリレーションに割り当てられるバッファ領域は，書き込み，参照が繰り返される．つまり， SE， LHT 方式ではプローブリレーションを各ノード上のローカル領域に獲得しているとはいえ，他ノードのプローブプロセスにより，参照されており，いったん，プローブのデータバッファは他ノード上のキャッシュに取り込まれている．従って，入出力プロセスが次にそのデータバッファへ書き込みを行う時点で，メモリ一貫性処理が生じ，表 6.1にある通り，単純なローカルな書き込みと比較して約 7 倍程度のコストがかかることになる．

一方， SHT， LHT-R 方式ではプローブデータをノード内のみで書き込み，参照しているため，

他ノード参照後の書き込みによるメモリ一貫性処理は起きない．プローブプロセスは他ノード上のハッシュテーブルを参照するが，ハッシュテーブル自体はビルドフェーズですでに生成されているため，書き込みは起きず，処理時間を短くなる．

6.4.3 入出力コストの影響

入出力をも含めた処理時間の測定に関しては，計測に用いた SPP 1600 の入出力性能が極めて低いため，いずれの方式でも入出力コストが CPU 処理コストよりも重くなり，方式ごとの違いが明確に出ない．しかしながら，最近のディスク転送レートの向上は非常に大きく，本稿では，現在使用されている一般の SCSI ディスクの転送速度に基づきディスク転送速度が変化した場合の予測性能も実測結果と併せて，以下に示す．

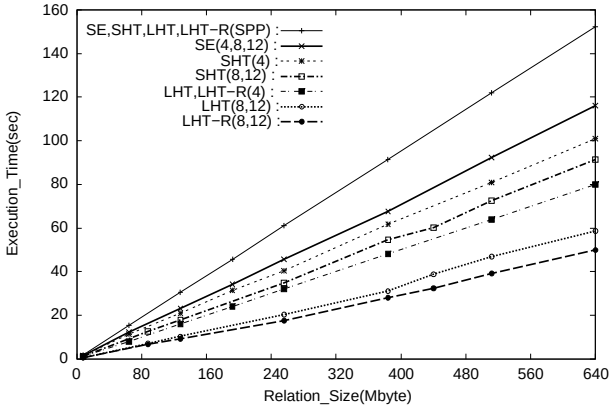

図 6.10: 全処理時間

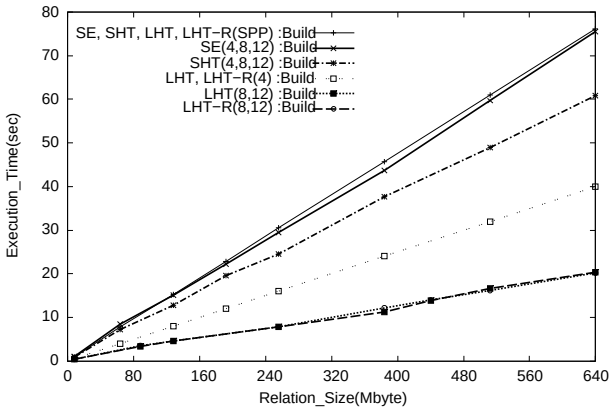

図 6.11: ビルドフェーズ処理時間

図 6.10,6.11,6.12に，入出力コストも含めた全体の処理時間，ビルドフェーズ処理時間，プローブ

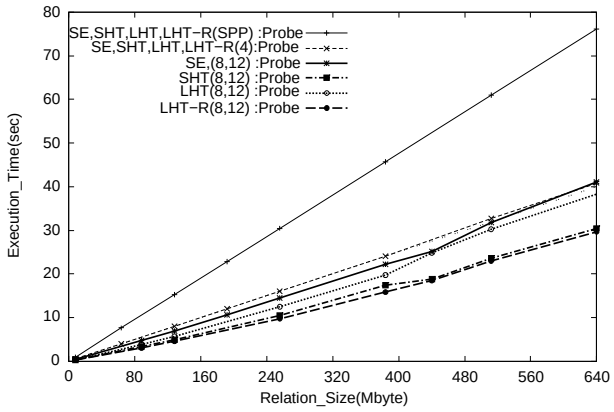

図 6.12: プローブフェーズ処理時間

フェーズ処理時間を示す．縦軸が処理時間，横軸が 1 リレーションのサイズを表す．図 6.10,6.11,6.12から分かるように， SPP 1600 上の結果 (図中で方式名の後ろに (SPP) で表した結果) は全ての方式において，いずれのフェーズ処理時間も，全処理時間でも同じ結果となる．これは， SPP 1600 では，2.1MB/sec と入出力性能が非常に重いためである．例えば， 640MB のリレーションを 2 つ読み出す時間は，およそ 152 秒 ((640MB * 2 relation) /(2.1MB/sec * 4 nodes)) となる．しかしながら，ディスク転送速度は年々に向上しており， Seagate 社の cheetah シリーズでは転送速度が 11.3MB/sec ∼ 16.8MB/sec となっている．そこで，転送速度を 4MB/sec， 8MB/sec， 12MB/sec と変化させ，その結果も一緒に図示する．図 6.10,6.11,6.12内の方式名の後ろの () 内の値が転送速度を示す．図から分かるように， SE 方式では，入出力コストが 4MB/sec 以上になると，メモリアクセスコストが支配的である． SHT 方式では，入出力コストが 8MB/sec 以上になると CPU 処理コストが支配的なり，全節の CPU コストのみの結果と同じになる．しかしながら，図 6.11から分かるように入出力コストが 4MB/sec の場合にはビルドフェーズではハッシュテーブルをグローバルに生成するために CPU 処理コストが入出力コストよりも重いくなる．一方，図 6.11に示されるように，プローブフェーズでは入出力コストが CPU 処理コストよりも重くなる．従って，全処理時間は， SPP による測定結果よりは速くなるが，プローブフェーズの入出力コストの影響が出ているため， CPU 処理コストのみの結果よりは遅くなることが図 6.10よりわかる． LHT 方式， LHT-R 方式では，入出力コストが 4MB/sec の場合には，入出力コストがビルド，プローブの両フェーズで CPU 処理コストよりも重いため，双方は，全処理時間，ビルド・プローブフェーズ処理時間は同じ結果となる．しかしながら， 8MB/sec 以上の場合には， CPU 処理コストのみの結果と同じになる．

このように， CPU 性能に対し，相対的に入出力性能が高くなるにつれ，当然のことながら，提案方式の有効性が明らかとなることが判る．

6.5 分散共有メモリシステムの性能評価

本節では、シミュレーションをもとにより詳細に分散共有メモリ計算機上の並列データベースシステムの性能ついて、前節で延べた４つの結合演算処理アルゴリズムを基に検討を行う。一般に解析的モデルはアクセスコストモデルなどを基に構築するが、分散共有メモリシステムのようにキャッシュを介してのメモリアクセスコストを定式化するのは非常に難しい。そこで、本節では、実システムの挙動に基づき、メモリアクセス数およびキャッシュヒット／キャッシュミス比／インバリデーション比などを実際の資源パラメタ (メモリサイズ、データベースサイズ等) から計算する。

すでに、 HP SPP1600 上にて我々が提案した方式の性能評価については [86] にて方向した。しかし、今までの計測においては実マシンを用いていたため、分散共有メモリ計算機におけるキャッシュサイズ、ノード数、リモートメモリアクセスコストなどは計算機に固有の値であり、また、均一なデータ分布のみの計測った。そこで、本節の性能評価においては、シミュレーションを用いることで、キャッシュサイズ、ノード数、リモートメモリアクセスストなど、分散共有メモリ計算機を特徴つける資源パラメタを変化させた場合について検討を行う。また、分散共有メモリ計算機上での並列ハッシュ結合演算処理に対するデータスキューの影響について調べる。

以下では、まず簡単にシミュレーション環境について述べ、続いて、そのシミュレーションの結果を実マシンの結果と比較することでシミュレーションの正当性について検証する。次に、上記のパラメタおよびデータ分布を変化させた場合に結果について述べる。

6.5.1 シミュレーション環境とパラメタ

本シミュレーションでは、図 6.1に示す CC-NUMA 計算機を扱うものとする。また、そのコストは表 6.1を基本メモリコストおよびパラメタ値として用いる。リレーションサイズは 8MB から 640MB まで変化させ、リレーションなディスクの上に同じサイズに分割して格納されているものとする。ページサイズは 4KB、タプル長は６４バイトとする。データ分布については、のちほど、詳細に述べる。

本計測では、すべてのキャッシュラインのディレクトリマップはすでに生成されているものとし、グローバルおよびローカルメモリはそれぞれのノードからのアクセスコストが同じになるようにランダムにアクセスされたものとする。ある特定のノードにデータがキャッシュされていると、そのアクセスコストはキャッシュしているノードとそれ以外のノードでは大幅に個となる。さらに、分散共有メモリ計算機上にデータベースサーバが稼働していると仮定した場合、十分な時間が経過した後には殆どすべてのメモリ空間はランダムにアクセスされていると考えられためである。また、以下の結果では、それぞれの方式のメモリアクセスコストの違いを明確にするために、ディスクアクセスコストは除いてある。

6.5.2 HP Exemplar SPP 1600 の結果とコスト式の比較

パラメタを変化させたシミュレーション結果を提示する前に、均一データ分布において実マシン (HP SPP1600) 上で計測した結果とシミュレーション結果の比較を行い、我々が用いた分散共有メモ

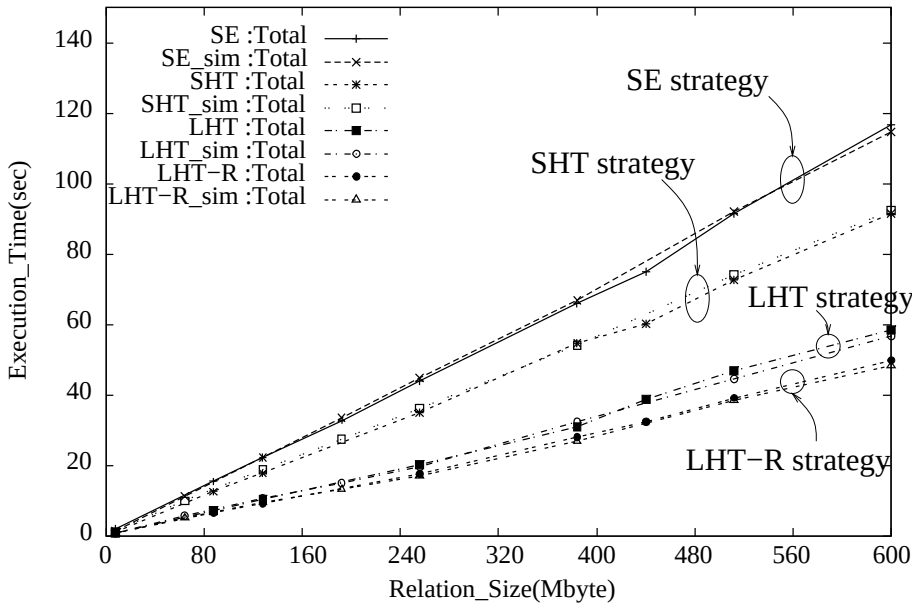

図 6.13: 実測値とシミュレーションの結果

リアクセスコスト・シミュレーションの検証を行う。 SPP 1600 の構成は 4 ノードとし、各ノードはクロスバースイッチで接続された 8 CPUs(PA-RISC 7200, 120 MHz) および 512MB local memory から構成される。また、各ノードは分散共有メモリ機構を支援する SCI プロトコルを採用した高速ネットワークで結合され、６４バイト単位のキャッシュラインごとにメモリコヒーレンスが実現されている。システム全体では、３２プロセッサ、２ GB の主記憶構成となる。

図 6.13に４方式 (SE, SHT, LHT, LHT-R) の実行時間について、 SPP1600 およびシミュレーションの結果を示す。この図からわかるように、シミュレーションと実マシン上の結果の違いは非常にわずかであり、シミュレーション結果は十分に実際の分散共有メモリ計算機の挙動として用いることが可能である。

図 6.13に示されるように, 全てのバッファをグローバルメモリに割付ける SE 方式が他の方式に比べると性能が低い。一方、 LHT-R 方式は図 6.13から分かるように最も性能がよく SE 方式と比較すると 58% もの改善が見られる。また、 LHT と LHT-R 方式の結果と SE,SHT 方式の結果を比べることにより、ハッシュテーブルをグローバルメモリに割り付けるよいローカルメモリに割り付けることで性能が改善されていることが分かる。

以下の節では、結果はすべてシミュレーションの結果である。

6.5.3 キャッシュサイズの効果

キャッシュされることにより、メモリアクセスコストは格段と小さくなるため、キャッシュサイズを大きくすることにより、並列結合演算処理コストも改善することが期待できる。図 6.14, 図 6.15および図 6.16にキャッシュサイズを変化させた場合に各方式の結果を示す。本計測では、全リレーショ

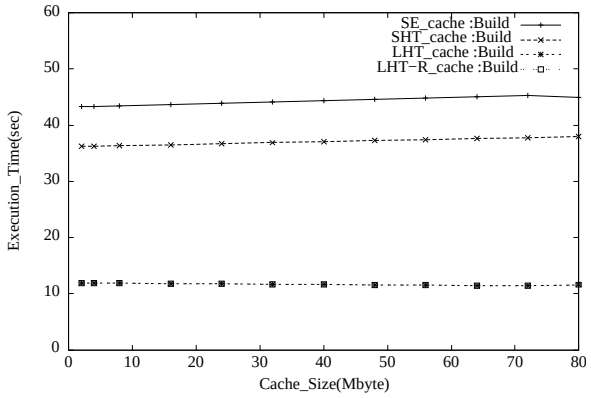

図 6.14: キャッシュサイズの効果 : ビルドフェーズ

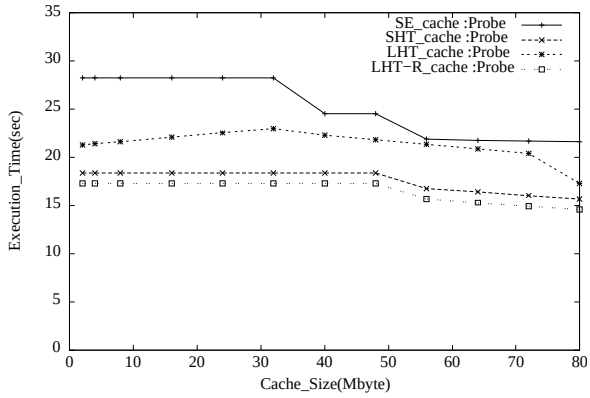

図 6.15: キャッシュサイズの効果 : プローブフェーズ

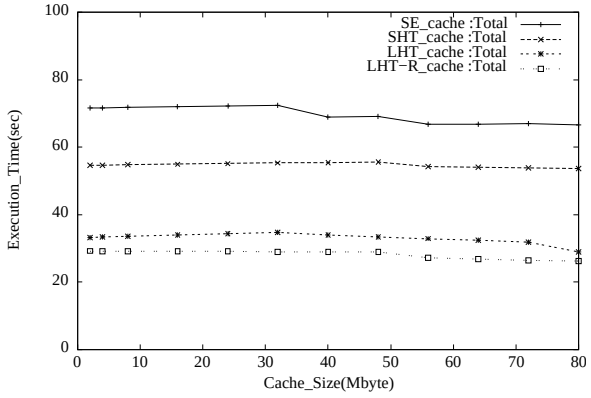

図 6.16: キャッシュサイズの効果 : 全体の処理時間

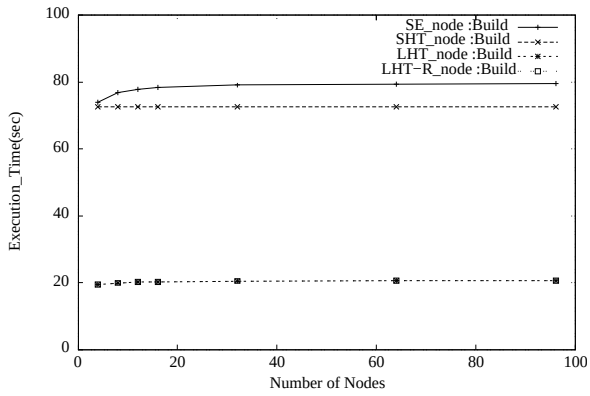

図 6.17: ノード数の効果 : ビルドフェーズ

ンサイズを 384 MB、すなわち、各ノードごとに 96 MB のリレーションを処理し、データ分布は均一とした。この図からわかるように、キャッシュサイズが大きくなるにも関らず、いずれの方式のビルドフェーズ処理コストが一定であることが確認できる。これは、ビルドフェーズでは主にハッシュテーブルへの書き込みを中心に、リモートメモリへの書き込みが支配的なためである。ビルドフェーズとは対照的に、 プローブフェーズの実行時間はキャッシュサイズが増加すると改善される。 SE および SHT 方式では、グローバルメモリあるハッシュテーブルがプローブフェーズでは頻繁に参照されるため、ほぼハッシュテーブル全体がキャッシュにヒットするようになるためと考えられる。さらに、ハッシュテーブル走査時にビルドリレーションもキャッシュされる効果もあると考えられる。 LHT および LHT-R 方式に関しては、 SE および SHT 方式と比べると、プローブフェーズにおいてもキャッシュによる効果は小さい。 LHT-R 方式では、ハッシュテーブル自体はローカルメモリに保持されているが、プローブプロセスでは他ノード上のハッシュテーブルを参照にいく。そこで、キャッシュサ

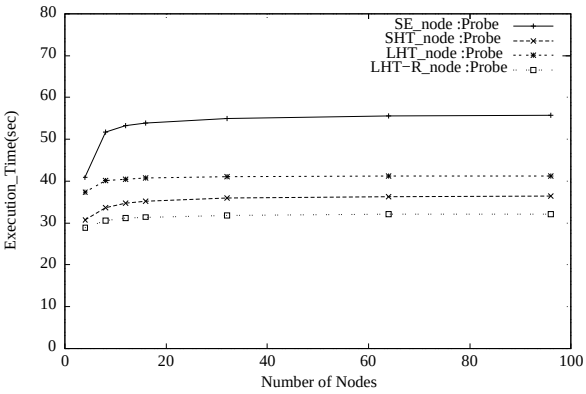

図 6.18: ノード数の効果：プローブフェーズ

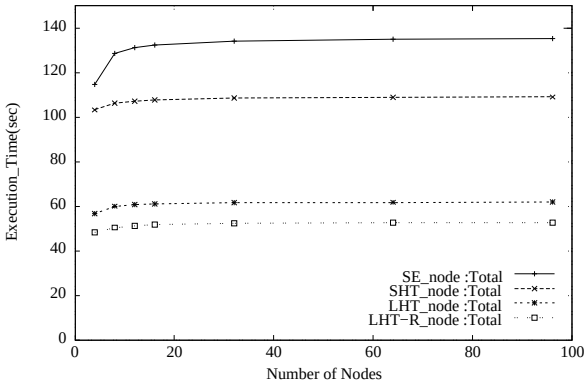

図 6.19: ノード数の効果：全体の処理時間

イズが大きくなると、ハッシュテーブル参照時のキャッシュヒット率は高くなる。しかし、ビルドフェーズにて、すでにビルドリレーションは各ノードごとにクラスタリングされているため、必要なビルドリレーションのキャッシュヒット率は比較的小さい。 LHT 方式では、プローブプロセスがハッシュテーブルの参照時にリモートメモリアクセスを行わないため、キャッシュサイズを大きくすることによる効果が得られない。

6.5.4 台数効果

ノード数を４台から９６台に増やした場合の各方式の実行時間を図 6.17、図 6.18および図 6.19に示す。本計測では、キャッシュサイズは 64MB 固定とし、リレーションサイズは各ノードあたり 160 MB とした。したがって、ノード数が増えると処理される全リレーションサイズも大きくなる。これらの図から、ノード数が８台以上の場合には実行時間が一定であることがわかる。これは、分散共有メモリ計算機ではシステムスケーラビリティが容易に得られることを示している。しかしながら、図 6.17、図 6.18および図 6.19からわかるようにノード数が４台から８台へ増やした場合には、各ノードごとに処理するリレーションサイズが同じであるにも関らず、実行時間が増加している。前の節で述べたように、キャッシュサイズは処理時間に影響を与える。この実験ではキャッシュサイズを一定としているため、相対的ノードが８ノードよりもキャッシュサイズが大きくなり、４ノードと８ノードの性能差はキャッシュサイズによるものと考えられる。また、ノード数が十分大きくなると、キャッシュの影響は相対的に小さくなり消える。以上の結果から、システムスケーラビリティは得られるが、分散共有メモリ計算機はノード数が増えるとリモートメモリアクセスコストが支配的になると考えられる。

6.5.5 リモートメモリレイテンシィの影響

今までの観察から、リモートメモリアクセスコストが分散共有メモリ計算機の性能において支配的と考えられる。キャッシュサイズもシステム性能に影響を 与えるが、それはリレーションサイズが

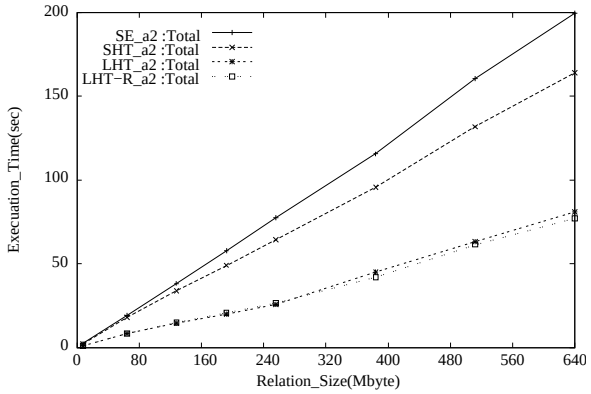

図 6.20: リモートメモリ (レイテンシィ：２倍) の場合

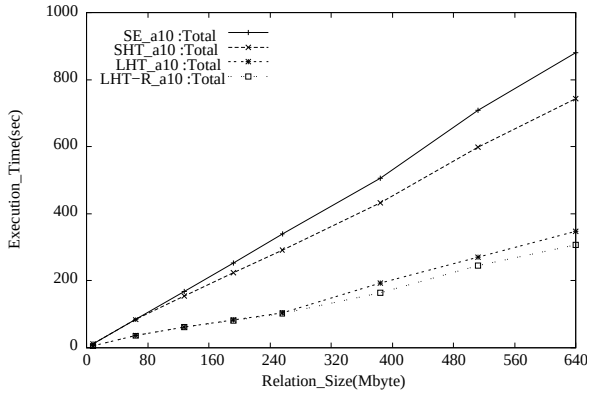

図 6.21: リモートメモリ (レイテンシィ：１０倍) の場合

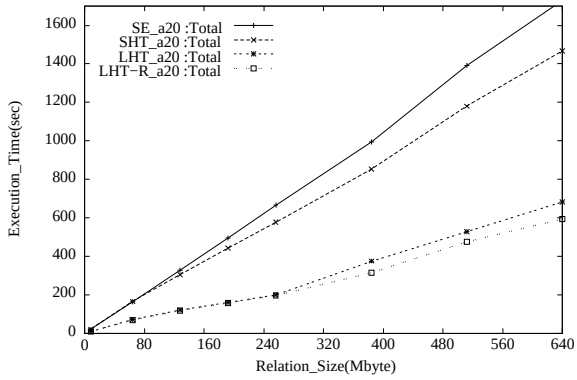

図 6.22: リモートメモリ (レイテンシィ：２０倍) の場合

キャッシュサイズと比較して十分におさまる程度の小さい場合飲みであり、ほとんどの実行時間はリモートメモリアクセスコストである。そこで、ここでは、ローカルメモリアクセスコストとリモートメモリアクセスコストの比について検討する。今まで、メモリアクセスコストに関しては、表 6.1に示す SPP 1600 の実測値をローカルメモリおよびリモートメモリのアクセスコストとして用いてきた。そこで、ここではこのメモリアクセスコスト比 (リモートメモリアクセスコスト／ローカルメモリアクセスコスト) を２倍、１０倍、２０倍を変化させる。メモリアクセスコスト比が２０倍は、ほぼソフトウェアによって実現された分散共有メモリシステムと同じといえる [2]。

図 6.20、図 6.21および図 6.22にそれぞれの方式の実行時間に関し、をメモリアクセスコスト比を２倍、１０倍、２０倍と変化させた場合の結果を示す。リレーションサイズは 16MB から 640 MB まで変化させ、キャッシュサイズは 64 MB 固定とした。データ分布は均一である。これらの結果から、メモリアクセスコスト比が大きくなるにつれて、グローバルメモリのアクセスが中心となる SE 方式の性能がローカルメモリにハッシュテーブルを保持する LHT(or LHT-R) 方式の性能差が大きくなることがわかる。

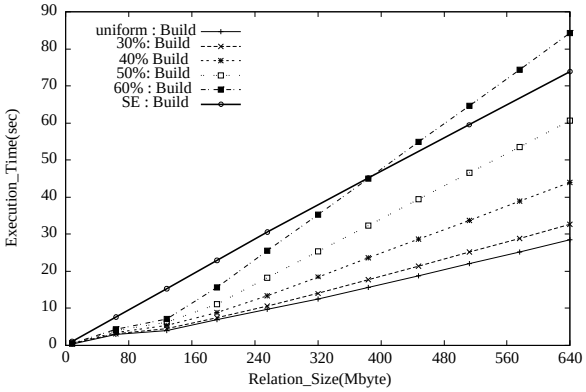

図 6.23: データ偏りの影響 (LHT 方式：ビルドフェーズ)

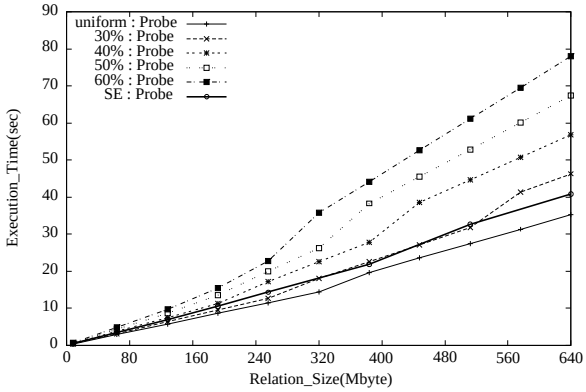

図 6.24: データ偏りの影響 (LHT 方式：プローブフェーズ)

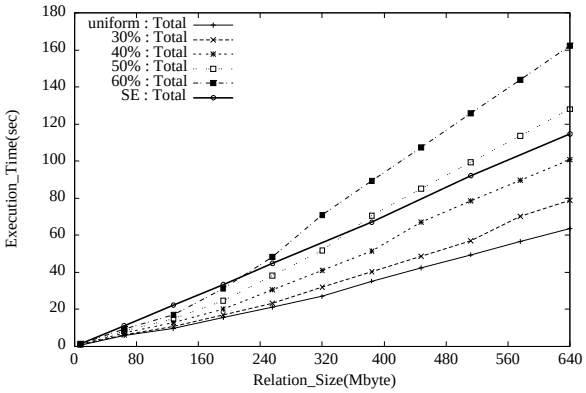

図 6.25: データ偏りの影響 (LHT 方式：全体の実行時間)

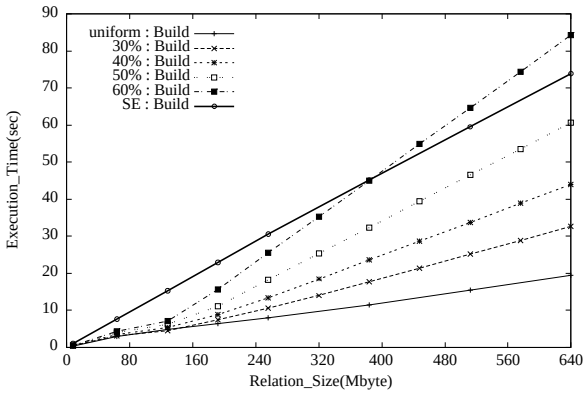

図 6.26: データ偏りの影響 (LHT-R 方式：ビルドフェーズ)

6.5.6 データの偏りによる影響

本節では、データスキューがある場合のシミュレーション結果について示す。以下の計測では、リレーション R と S は同じデータ分布とし、分散共有メモリ計算機がデータスキューにどの程度強いかを明らかにすることが目的であるため、負荷分散機構については考慮しない。

ここで、データスキューは全リレーション中、一つのノードに集中したデータサイズ比 $load\ N\%$ として表す。残りのリレーションは、そのノード以外のすべてのノードに均等に分散するものとする。これは、分散共有メモリ計算機の性能評価をおこなっている文献 [92] にしたがうものである。

(1) LHT 方式

図 6.23、図 6.24および図 6.25にデータスキューとして uniform(load 25%) から最大負荷が６０%偏った場合 ($load$ 60%) までの LHT 方式の結果を示す。リレーションサイズは 16MB から 640MB

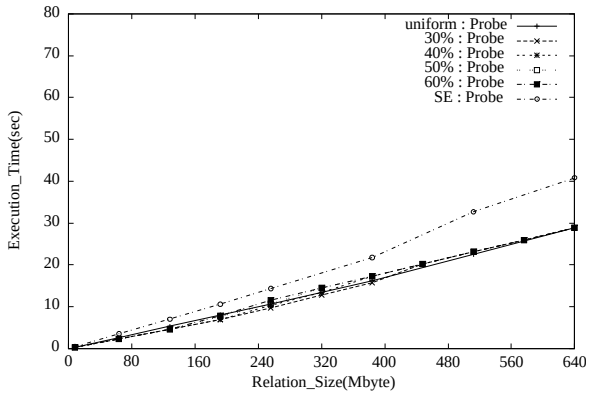

図 6.27: データ偏りの影響 (LHT-R 方式：プローブフェーズ)

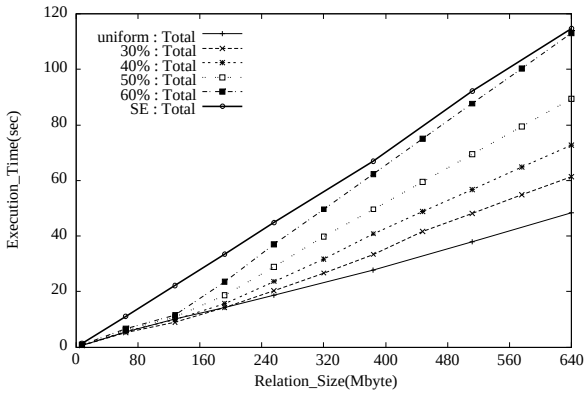

図 6.28: データ偏りの影響 (LHT-R 方式：全体の実行時間)

まで変化させ、キャッシュサイズは 64 MB 固定とする。参考のために、 SE 方式の結果もあわせて載せている。

SE および SHT 方式では、特定のノードではなく、ハッシュテーブルおよびデータバッファ全体を均等にアクセスするため、データスキューはその性能に大きな影響は与えない。一方、 LHT 方式では、ハッシュ値が同じデータは単一のノードにわりつけられ、そのノード上で処理をする。そのため、分散メモリアーキテクチャ上の並列ハッシュ結合演算処理と同様に、データスキューはその性能に大きな影響を与える。

図 6.23、 6.24および 6.25に見られるように、データスキューが大きくなると LHT 方式の実行時間は大きくなる。図 6.23のビルドフェーズの結果から、データが偏っていたとしてもキャッシュおよびローカルメモリ上に保持できる間は SE 方式よりも処理性能がよいことがわかる。しかしながら、処理すべきデータがローカルメモリおよびキャッシュサイズから溢れると、リモートメモリアクセスコストが増えるため、処理時間はかなり大きくなる。プローブフェーズでは、リモートアクセス、特に他ノードのキャッシュ上にのっているため invalidation を伴うローカルメモリへの書き込みコストが増えるため、 LHT 方式の処理時間は SE 方式よりも大きい。これらん結果、データスキューが増えると、 LHT 方式の処理時間は非常に大きくなる。

(2) LHT-R 方式

LHT-R 方式の実行時間について、データスキューを均一から (load 25%) 一つのノードの負荷が ($load$ 60%) まで変化させた場合の結果を図 6.23、図 6.24および図 6.25に示す。リレーションサイズは 16MB から 640MB まで変化させ、キャッシュサイズは 64 MB 固定とする。参考のために、 SE 方式の結果をあわせて示す。

図 6.26、 図 6.27および図 6.28からわかるように, データスキューが大きくなると LHT-R 方式の実行時間は LHT 方式と同様に大きくなる。前述しているように、ビルドフェーズ自体は LHT-R 方

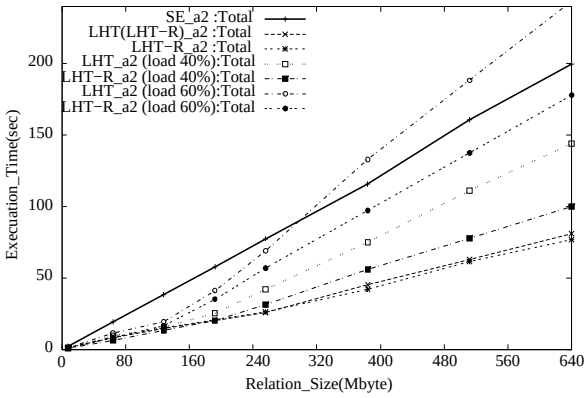

図 6.29: リモートメモリ (レイテンシィ：２倍) のデータ偏りの影響

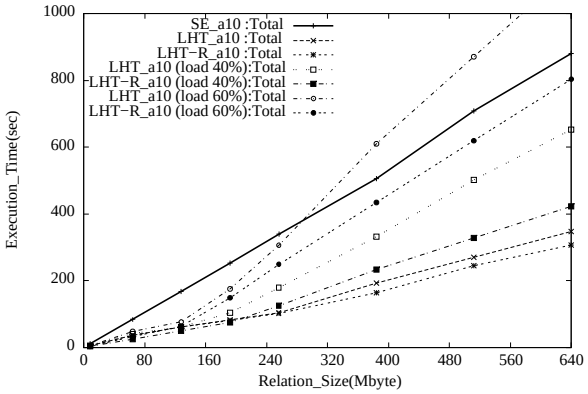

図 6.30: リモートメモリ (レイテンシィ：１０倍) のデータ偏りの影響

式と LHT 方式ではまったく同じである。したがって、図 6.26のビルドフェーズの処理時間はデータスキューにより、もっともロードの大きなノードのデータがローカルメモリおよびキャッシュに保持できなくなると、増加する。ビルドフェーズとは個となり、図 6.27から LHT-R 方式のプローブフェーズ処理時間は LHT 方式ほどデータスキューの影響を受けない。これは、 LHT-R 方式では、 Invalidation を伴うリモートおよびローカルメモリアクセスが生じないことによる。

(3) リモートメモリアクセスレイテンシィとデータの偏りの関係

今までの結果から、リモートメモリアクセスコストおよびデータスキューの双方が分散共有メモリ計算機上の並列結合演算処理性能に影響を与えることがわかる。データスキューが変化すると、 LHT 方式および LHT-R 方式の全リモートメモリアクセスコストも変化する。そこで、データスキューおよびローカルメモリおよびリモートメモリのメモリアクセスコスト比双方が変化した場合の LHT 方式および LHT-R 方式の性能について検討する。

図 6.29および 6.30はそれぞれ、メモリコスト比を２倍、１０倍、２０倍にした場合の二つの方式の処理時間を示す。参考のために、 SE 方式の処理時間も併せて示す。これらの図から、図 6.25と図 6.28と比較して、 SE 方式がリモートメモリアクセスコストが大きくなるにつれて急激に処理時間が増えるのに対し、 LHT および LHT-R 方式双方の処理時間がゆるやかに増加していることがわかる。特に、 SE 方式と比べると、 LHT-R 方式は負荷 ($load60\%$) が大きくなっても処理性能が良いことがわかる。負荷 ($load60\%$) のデータスキューにおちて、 LHT-R 方式は SE 方式と比較しておよそ 10% ほど性能がよい。これは、リモートメモリアクセスコスト比が高い場合には、データスキューが存在しても、ノード内ローカリティを考慮した方式を用いることが重要であることがわかる。

6.5.7 Copy Hash Table 方式

前節の結果が示すように、データの偏りが起きた場合には、 LHT-R 方式においても、偏りが集

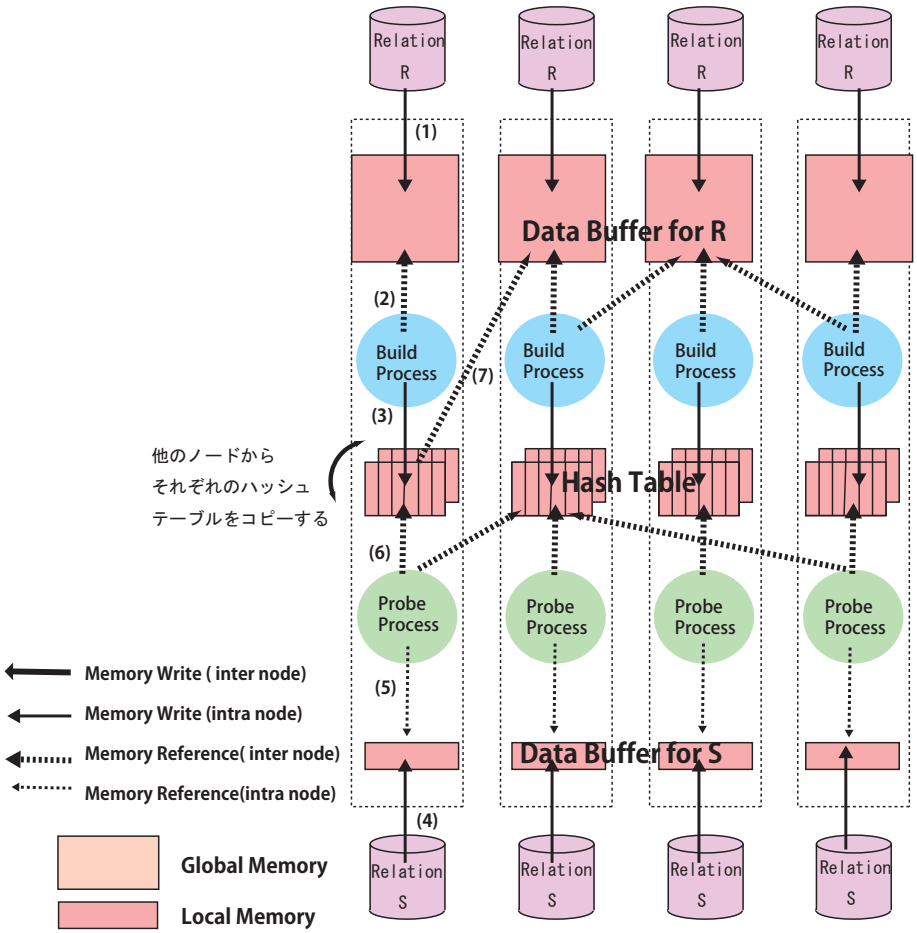

図 6.31: Copy Hash Table(CHT) 方式

中しているハッシュテーブルを局所的にアクセスしても、キャッシュにおさまらないため、性能が劣化する。そこで、あらかじめハッシュテーブルをコピーすることにより、ローカルメモリ上に置くことで、キャッシュサイズより大きなハッシュテーブルの頻繁な参照によるキャッシュミスを防ぎ、性能の低下をおさえられることが期待できる。

本方式の実装では、 LHTー R 方式のビルドフェーズ終了時に、作成されたハッシュテーブルの複製 (あくまでもハッシュテーブルのみで、データ自体は対応ノード上のみにあるとする) をそれぞれのノード上に作成することにより、頻繁にアクセスするハッシュテーブルはそれぞれのノード上のローカルデータを参照し、必要なデータボディのみ他ノード上の内容を参照することとする。

図 6.31に Copy Hash Table 方式の処理の流れを示す。

本方式では，ビルドフェーズは， LHT 方式と同じであり，プローブフェーズはほぼ SHT 方式と同様の流れとなる．ビルドフェーズでは、 (1) 入出力プロセスがデータの読み出し時に各ノードごとにスプリットされたローカルなデータバッファに書き込む． (2) それを、該当するノードのビルドプロセスが参照し、ローカルメモリ上にハッシュテーブルを生成する。 (3) 全てのリレーション R が読み終わった後、ビルドプロセスは他ノード上に生成され t ハッシュテーブル (データのボディはコピー

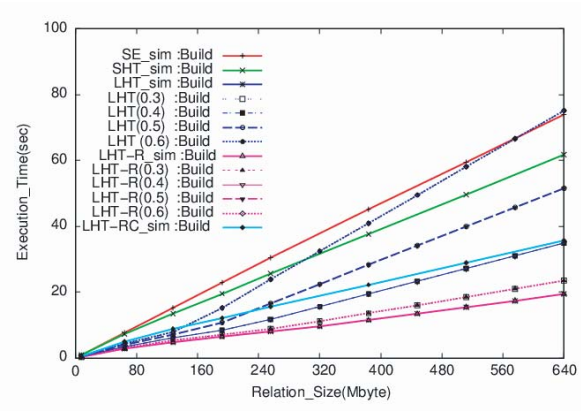

図 6.32: リモートメモリアクセスコストを変化させた場合のデータ偏りの影響

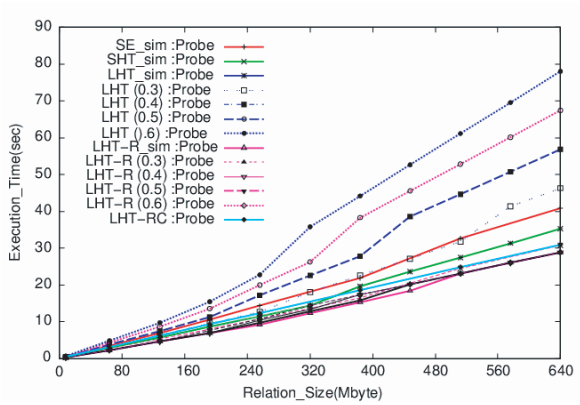

図 6.33: リモートメモリアクセスコストを変化させた場合のデータ偏りの影響

しない) をローカルメモリ上にコピーする。プローブプロセスはそのデータバッファからタプルを読み出し，キーの値を基に該当するハッシュテーブルをプローブ，結合演算処理をする．このとき、他ノード上のハッシュテーブルがすでにノード内にあるため、他ノードへの参照コストが抑えられる。

図 6.32および 6.33はそれぞれ、各ノードの負荷の偏りを変化させた場合の、 SE, SHT, LHT, LHT-R, CHT 方式を LHT-R に応用した場合 (LHT-RC) の結果を示したものである。 4 台のプロセスに負荷が均等に割り振られたときを 25% ととし、最も負荷の高いノードの負荷を 30%, 40%, 50%, 60% と変化させている。 LHT 方式および LHT-R 方式では負荷の偏りが生じるため、個別に結果を示しているが、 SE、 SHT 方式は偏りの影響を受けないため、一つの結果のみを示し、 CHT 方式では偏りにより差異がほとんどないため、最も高い偏り (60%) の結果のみを図中に示している。

これらの図から、 CHT 方式を採用することで、ビルドフェーズの処理時間がハッシュテーブルのコピーコストがあるため、 LHT-R 方式より大きくなっていることが分かる。しかしながら、 LHT 方式が負荷が大きくなるとビルドコストが大きくなるのと比較し、 CHT を採用した方式ではハッシュテーブルのサイズは一定であり、またさほど大きくないため、ほぼ負荷が一様の LHT 方式と同じコストであることが確認できる。

プローブフェーズの処理時間は負荷の偏りをほとんど受けず、偏りのない LHT 方式および LHT-R 方式の処理時間とほぼ同じであり、 SE 方式や SHT 方式の処理時間よりも小さい。一方、 LHT 方式と LHT-R 方式では、負荷の偏りがあると、リモートメモリ参照のためにコストが増えてしまい、一番負荷の重いノードが 60% の場合、 CHT 方式と比べ、 2 倍ほど処理時間が増加している。

この結果から、ノード内ローカリティを考慮した方式を用いることが負荷の偏りの対処にも有効であり、処理性能をあげることが可能であることが確認できた。

6.6 本章のまとめ

本章では，分散共有メモリ計算機上の並列データベース処理の実装方式の検討を行うため，並列ハッシュ結合演算を取り上げ，分散共有メモリアーキテクチャに適合したバッファ管理方式を提案し，実際に商用分散共有メモリ計算機 (HP Exemplar SPP 1600) 上に実装することで，我々の提案したバッファ管理方式の有効性を確認した.

データベース処理は本質的に広大なアドレス空間をランダムに検索する必要があるため，分散共有メモリ計算機を用いた場合，単純な実装では共有メモリへのランダムなアクセスによりメモリ一貫性処理負荷が高くなり，処理性能が下がるという問題点がある．本研究では，分散共有メモリ計算機上での並列ハッシュ結合演算処理モデルを提案し，バッファ領域をハッシュテーブル領域，データバッファ領域と区分し，参照，書き込みのアクセス特性を考慮することで， SE， SHT， LHT， LHT-R 方式の 4 方式を提案し，これらの方式を SPP 1600 上に実装し，性能評価を行った．その結果，CPU 処理コストに関しては，単純な SE 方式の結果と比較して，最も性能のよい LHT-R 方式では，50% 以上の性能向上が得られることを確認した．また，入出力コストも含めた処理時間の結果を示し，現在のディスク転送速度を前提とした場合，分散共有メモリ計算機上で性能向上を図るためには，分散共有メモリアーキテクチャに適合したメモリ管理方式を用い，メモリアクセスコストを低減する必要があることを示した.

続いて、分散共有メモリ計算機のデータベース処理に対するスケーラビリティ、拡張可能性などを検討するために、シミュレーションを用いて分散共有メモリ計算機の性能に大きな影響を与えるリソース (キャッシュサイズ、ノード数、メモリアクセスコスト比、データスキュー) を変化させて性能評価を行い、分散共有メモリ計算機のスケーラビリティを確認した。

本章で提案した４方式へのデータスキューの影響を検討し，大容量主記憶では参照局所性を考慮し、演算処理の特性を利用することで局所性を利用して、さらなる性能向上が可能であることを示し、新たな CHT 方式を提案した。本章で示した分散共有メモリアーキテクチャ上の並列データベース処理実装に関する指針は、数百 GB の主記憶を擁する最新の共有メモリエンタプライズサーバにおいても適用可能であり、複雑な問合せ処理にも有効であると考えられる。

第 7 章

GN ハッシュ結合演算処理

7.1 ハッシュ結合演算処理

関係データベースは高度なデータ独立性，簡易なユーザインタフェース，強固な論理的基盤等の優れた特徴を有し，近年，データベースの中心となりつつある。しかし，従来からのネットワーク型モデルや階層型モデル等における手続き型処理に比べ，関係データベースにおける非手続き的な問い合わせの処理は負荷が高くなる傾向にある。特に，リレーション同士のリンクを動的に行う結合演算は問い合わせの記述性を高める反面，ファイル間相互の関連付けが静的なリンクで実現されているネットワークモデル等に比べ、動的に双方のリレーションを検索，比較するため単純な処理方式ではその処理負荷は非常に重くなる。この様な背景から，関係データベースシステムの高速化を目指し、特に結合演算を中心とした効率の良い関係代数演算処理方式の研究が数多く行われている。

現在までに結合演算処理方式として，ネストループ結合方式，ソートマージ結合方式，ハッシュ結合方式が提案されてきた。ネストループ結合方式は最も単純な方式であり，初期の関係データベースシステムでは実装上の容易さ等から採用するものが多かった [110][3]。ネストループ結合方式は，二つのリレーションをR, Sとするとリレーション R の１タプルを取り出してはリレーション S の全タプルと比較，結合処理を施す。この繰り返し処理の負荷は $O(R \times S)$ （R, S はリレーションR, S のタプル数）となり，負荷が両リレーションの積に比例するため大容量リレーション処理に適しているとは言えない。一方, 文献 [7] では，インデクスの無いリレーション同士の結合を行う場合，両リレーションをあらかじめソートし，ソート済のリレーションを突き合わせることにより結合処理を施すソートマージ結合方式の性能がネストループ結合方式を用いるより効率が良いという報告がなされた。ソートマージ結合方式の処理時間は $O(R \log R + S \log S)$ となり、ネストループ結合方式よりも高速化が期待できるため，現行の関係データベースシステムの多くはソートマージ結合方式により結合演算を実現している場合が多い。

近年、更なる高速化を目指し, ハッシュ結合方式が提案されている [61, 12, 129, 26]。ハッシュ結合方式では，まずソースリレーションをハッシュ関数を用いることにより互いに値の重ならない独立なクラスタに分割する。実際のタプルの突き合わせと結合処理は，クラスタ間に重複がないので，同じハッシュ値をもつクラスタ毎に独立に行えばよい。従って、単純なネストループ結合方式が $O(R \times S)$ 処理時間かかるのに対してハッシュ結合方式では $O(\sum_i R_i \times S_i)$ の処理時間 (R_i,S_i はリレーション R,S のハッシュにより分割された各々のクラスタのタプル数を示す) となり，シミュレーションによる性能評価結果からも従来のネストループ結合方式やソートマージ結合方式と比較してはるかに高い性能が得られることが確認されている [12, 129, 26, 96, 103, 25]。また，射影演算における重複除去や集計演算などの関係代数演算処理にも容易に適応できる。しかしながら、ネストループ結合方式の入出力コストがデータ分布の偏りによらず一定であるのに対し、ハッシュ結合方式では、ハッシュ関数による分割後のクラスタのサイズを予測することは難しく、データ分布の偏りにその性能が大きく影響を受けると考えられる。従来提案されてきたハッシュ結合方式の性能比較においてはいかに主記憶を効率良く使用するかに主眼がおかれ、データ分布は一様分布（つまりクラスタのサイズは主記憶にちょうど入るサイズに均等に分割される）を仮定した性能比較が主であり [96, 103]、データ分布

が不均一な場合の性能評価では、主記憶を越えない範囲でのクラスタサイズのばらつきに対する性能評価しか行なわれていない [97]。主記憶を越えたクラスタが生成された場合を考慮に入れた性能比較はほとんどなされておらず [58]、いずれの方式においても主記憶を越えたクラスタにはそれ々の方式を再帰的に適応すると述べるに留まっており [97, 103]、十分な解析が行なわれているとは言い難い。また、アルゴリズムをより高度化する研究もあるが [135]、バッファ管理が極めて複雑になると同時にそのオーバヘッドも無視できなくなることが報告されている。

本章は，まず，第 7.2節において従来提案されている各種ハッシュ結合方式の処理方式について簡潔に説明するとともにその問題点について考察する。また，処理コスト解析式を導入し，各方式の処理コストについて検討する。第 7.3節において Grace ハッシュ結合方式とハッシュを用いたネストループ結合方式を組み合わせることでデータの分布が偏っている場合にも性能が大きく劣化することのない GN ハッシュ結合方式を提案する。本方式では，実行時にネストループ結合方式と Grace ハッシュ結合方式の二つから入出力コストの小さい方式を選択することで、従来のハッシュ結合方式に比べ、データ分布の偏った場合の性能を改善すること目指している。また、その処理手法も単純であり実装も比較的容易と予想される。第 7.4節では、第 7.2節で導入した処理コスト式を用いて、不均一なデータ分布の場合の GN ハッシュ結合方式と従来の各種ハッシュ結合方式の性能比較を行う。本章で提案する GN ハッシュ結合方式がクラスタが均等な大きさに分割できる場合において従来最も性能のよいハイブリッドハッシュ結合方式と同等の性能であるのみならず、 Zipf-like 頻度分布に従ったクラスタサイズを有する場合の性能比較においても、他のハッシュ結合方式と比べ高い性能を示すことを確認する。第 7.5節では、 GN ハッシュ結合演算において、実行時にリレーションサイズを基にして Grace ハッシュ結合方式とハッシュを用いたネストループ結合方式を選択する。その実行時の選択ポイントにおいて、選択した時点での推定値と実際のリレーションのデータの偏り、クラスタのサイズのばらつきに対して、アルゴリズムが頑健であるかどうか、あるいは、ネストループ方式および GN ハッシュ方式の I/O コスト評価式における値として最も適当な値について検討を行う。第 7.6節では、シミュレーションを用いて、 I/O コスト評価式における選択値として適当な値の範囲をデータの偏りを変化させて調べ、 I/O コスト評価式のアルゴリズム選択値としてかなり広い範囲にわたるものが利用可能であること、均等に分散したデータ時の評価選択方式を用いることで非常に傾きが高い場合にも問題ないことを示し、 GN ハッシュ方式がバッファ管理などをデータの偏りによって詳細に行う必要がある Hybrid Hash 方式などと比較して、単純なアルゴリズム選択方式を用いて十分にデータの偏りに対して頑健なアルゴリズムであることを示す。第 7.7節にて本章のまとめを行う。

7.2 従来のハッシュ結合法とその問題点

主記憶容量を越えるリレーションに対するハッシュを用いた結合演算方式に関しては、その基本形である Grace ハッシュ結合方式 [61, 12, 129]、単純ハッシュ（Simple Hash）結合方式と Grace ハッシュ結合方式を融合したハイブリッドハッシュ結合方式 [26]、更にハイブリッドハッシュ結合方式の性能改善を試みた動的 Grace ハッシュ結合方式 [135, 87] などが従来提案されてきた。とりわけ、

ソートマージ結合方式に対する優位性が明らかにされた後 [96]、活発な実装と並列化への努力がなされている [30]。本節では従来からのネストループ結合方式にハッシュ関数を適用したネストループハッシュ結合方式，単純ハッシュ結合方式、 Grace ハッシュ結合方式、ハイブリッドハッシュ結合方式、及び動的 Grace ハッシュ方式についてそれぞれの処理方式を簡単に解説し、処理コスト式を導入する。さらに、この処理コスト式に基づき、入出力コストによる各処理方式の性能比較を行ない、問題点について検討する。

7.2.1 従来のハッシュ結合方式とその処理コスト

本節以降、二つのリレーション R、 S に対し、結合属性 joinkey に関する等結合演算を想定する。対応する SQL 文を以下に示す。

```
SELECT * FROM R,S WHERE R.joinkey = S.joinkey
         AND R.a1 < Constant1 AND S.b1 > Constant2....
```

where 節第一条件は二つのリレーションの結合を表し、第 2 条件以降は R,S 各々に対する選択条件とする。本節で導出される処理コスト解析式に用いるパラメタ及び条件を表 7.1に示す。表にある選択率とはソースリレーションのタプル数に対する where 節第 2 項目以降の条件にの適用後のタプル数の割合を指す。いずれの方式においても結果リレーションの書き戻しコストは同じであるため，このコストは省略する。また、以下の説明において、混乱を避けるために、主記憶上でタプルの突き合わせのためにハッシュ分割されるクラスタをバケット、分割に用いられる関数をハッシュ関数と呼び、二次記憶に書き戻されるクラスタを入出力クラスタ、分割に用いられる関数をスプリット関数と呼ぶ。

(1) ネストループハッシュ結合方式

ネストループハッシュ結合方式（以下、本論文では簡単にネストループ結合方式とする）では、主記憶上のタプル突き合わせ処理として、ハッシュ関数を用いてハッシュテーブルを生成するが、入出力方式に関しては従来のネストループ結合方式と同じく、入出力クラスタは生成しない。ここではリレーション R,S のサイズをリレーション R が小さい（$R \leq S$）とする。本方式ではまず、リレーション R を外側のループとし、主記憶上に収まるだけのタプルをハッシュ関数を用いてハッシュテーブルに展開する。続いて内側のループとしてリレーション S を残りの主記憶にロードし，各タプル毎にリレーション R のハッシュテーブルを探査し結合演算処理を行う。リレーション S の全てのタプルに関してこの内側のループ処理が行われる。続いて，改めて外側のリレーション R の未処理のタプルが主記憶に収まるだけハッシュテーブルとして展開され，再びリレーション S が全部読み出され、結合演算処理を行なう。リレーション R の全てのタプルに対する突き合わせ処理が終わるまで上記の処理が繰り返し行われる。

本方式の処理コスト式は以下のように表される。小さいリレーション R を外側のループで分割して読み込むため，外側ループ回数は $n = \lceil \frac{f_r R}{N} \rceil$ となる。

$$Cost_{NL1} \quad = \quad Init + |R|_{io} + n|S|_{io} + |r|_{comp} + |s|_{comp} +$$

パラメタ	説明
$R,\ S$	リレーション R,S のサイズ (単位：ページ)
$r,\ s$	リレーション R,S のサイズ (単位：タプル)
$M = N + 2$	主記憶サイズ (単位：ページ) ここで N はリレーション R のための領域、他にリレーション S 用に 1 ページ ,結果リレーション用に 1 ページ用意されている。
$\|X\|_{io}$	X ページの読み出し, 書き戻し時間
h	主記憶上のクラスタ数 (バケット数)
H	二次記憶上のクラスタ数 (入出力クラスタ数)
$f_s,\ f_r$	選択率
$Init$	ハッシュテーブル初期化時間
$\|x\|_{comp}$	x タプルの選択条件比較時間
$\|x\|_{move}$	x タプルのデータ転送時間
$\|x\|_{hash}$	x タプルのハッシュ関数処理時間
$\|y * x\|_{probe}$	y タプルから成るハッシュテーブルに対する x タプル分の探索時間

表 7.1: 処理コスト式で用いるパラメタ

$$
\begin{aligned}
&|f_r r|_{hash} + |f_r r|_{move} + n|f_s s|_{hash} + n|f_s s|_{move} + \\
&n\sum_{i=1}^{h} |\left(\frac{f_r r}{n}\right)_i * (f_s s)_i|_{probe}
\end{aligned}
\tag{7.1}
$$

式 (7.1) からわかるように内側のリレーション S は外側のループ数 n 回だけ読み出される。そこで選択率 f_s が小さい場合には選択後のリレーション S を一旦書き戻し，２度目の読み出しからこの書き戻したデータを読み込む方式が考えられる。この場合のコストは、以下のようになる。

$$
\begin{aligned}
Cost_{NL2} \quad = \quad & Init + |R|_{io} + |S|_{io} + n|f_s S|_{io} + |r|_{comp} + \\
& |s|_{comp} + |f_r r|_{hash} + |f_r r|_{move} + n|f_s s|_{hash} + \\
& n|f_s s|_{move} + n\sum_{i=1}^{h} |\left(\frac{f_r r}{n}\right)_i * (f_s s)_i|_{probe}
\end{aligned}
\tag{7.2}
$$

本ネストループ結合方式では，従来のハッシュを用いない方式と比べ、主記憶上の突き合わせ処理に関してはハッシュを施すことで性能向上が図れるが、入出力コストに関しては変わらず、リレーションが大きくなると内側のリレーションを何回も読み出すため、入出力コストが急激に増大する。

(2) 単純ハッシュ結合方式

単純ハッシュ結合方式では、リレーション R の読み出し時にスプリット関数を適用して主記憶にロードするクラスタを決定するが、書き戻し時は入出力クラスタを生成しない。すなわち、リレーション R を読み出す際に選択的に一つの入出力クラスタのみを主記憶上にロードし，主記憶から溢れたデータは全てディスクに書き戻す。まず、リレーション R のロードされるデータに対してハッシュテーブルが生成される。続いて，リレーション S が読み出され，当該クラスタに関するタプルについては，結合処理が施され，残りのデータはディスクに書き戻される。上記と同様の処理がディスクに書き戻されたデータに対して施され、すべてのデータが処理されるまで繰り返される。

本方式の処理コストを以下に示す。まず，リレーション R を全て検索するための外側ループ回数はネストループ方式と同様に $n = \lceil \frac{f_r R}{N} \rceil$ となる。

$$
\begin{aligned}
&Cost_{simple} = \\
&\quad Init + |R|_{io} + 2|(n-1)f_r R - \frac{n(n-1)}{2}N|_{io} + \\
&\quad |r|_{comp} + |nf_r r - \frac{n(n-1)}{2}N|_{hash} + |f_r r|_{move} + \\
&\quad |S|_{io} + 2|(n-1)f_s S - \frac{n(n-1)}{2}N\frac{f_s S}{f_r R}|_{io} + |s|_{comp} + \\
&\quad |nf_s S - \frac{n(n-1)}{2}N\frac{f_s S}{f_r R}|_{hash} + |f_s s|_{move} + \\
&\quad n\sum_{i=1}^{h}|\left(\frac{f_r r}{n}\right)_i * \left(\frac{f_s s}{n}\right)_i|_{probe}
\end{aligned}
\tag{7.3}
$$

ネストループ結合方式が毎回リレーション S を全て検索しているのに対し、本方式は、主記憶に特定のクラスタのみをロードしリレーション R と S の当該クラスタを除いた部分をディスクに書き戻すことにより、主記憶上での検索コストを減らしている。従って主記憶上での比較処理コストに関してはネストループ方式より効率が良い。しかし，処理コストの内, 最も大きな比重を占める入出力コストに関してはネストループ結合方式では外側のリレーションは一回しか読み出されないのに対し，本方式では主記憶から溢れたデータは少なくとも書き戻し及び読み出しの 2 回のアクセスが必要となり，リレーションの大きさに対する処理コストの増加はネストループ結合方式より大きい 。

(3) Grace ハッシュ結合方式

前述の二つの方式では，主記憶上での結合処理にハッシュ関数を適用しデータ分割することで処理時間の低減を図っており、いずれかのリレーションが主記憶に収まる場合にその性能は良いが、二次記憶上のデータに対しては基本的にループ処理であり、リレーションが大きくなると入出力コストの増加は著しい。 Grace ハッシュ結合方式, 及び後で述べるハイブリッドハッシュ結合方式, 動的 Grace ハッシュ結合方式では，二次記憶上のデータを一旦スプリット関数を適用して分割し，二次記憶上に入出力クラスタを動的に生成することで入出力コストの削減を目指しており，大規模なリレーション

を処理する上で最も大きな割合を占めている入出力コストを前述の二つのループ方式の処理と比較して大幅に小さくすることができる。

Grace ハッシュ結合方式は, スプリットフェイズと結合フェイズから構成される。この二つのフェイズは完全に分かれており，まず，主記憶上に収まるように両リレーションをいくつかの入出力クラスタに分割 (スプリット) し，それぞれのクラスタ毎に結合処理が行われる。本方式の流れは以下のようになる。

スプリットフェイズ　リレーション R をディスクから読み出し、適当なスプリット関数を選びそれぞれのクラスタが主記憶に格納できるような大きさになるよう分割し、クラスタ毎にディスクに書き戻し、入出力クラスタを生成する。リレーション S も同じスプリット関数を用いて分割し、ディスクに書き戻す。

以上のスプリットフェーズを終了すると，分割された入出力クラスタ数回だけ次の結合フェイズを繰り返す。

結合フェイズ　リレーション R の入出力クラスタ R_i を主記憶上にロードする。 Grace ハッシュ結合方式では，主記憶上のデータの結合演算処理方式として，単純なネストループ方式, ソートマージ方式，ハッシュ方式等が考えられるが，本論文では，ハッシュ方式を採用し，リレーション R の各入出力クラスタ毎にハッシュテーブルを生成することとする。リレーション S の該当する入出力クラスタ S_i のタプルを主記憶上にロードし，ハッシュテーブルを走査し結合演算処理を施す。

本方式では，一旦ソースリレーションを入出力クラスタとしてディスクに書き戻しているため、均等に分割できるとすると入出力クラスタ数は， $H = \lceil \frac{f_r R}{N} \rceil$ となる。従って処理コストは次式になる。

$$
\begin{aligned}
Cost_{Grace} = Init+ \\
& |R|_{io} + |r|_{comp} + |f_r r|_{hash(=split)} + |f_r r|_{move} + \\
& |S|_{io} + |s|_{comp} + |f_s s|_{hash(=split)} + |f_s s|_{move} + \\
& 2\sum_{i=1}^{H} |((f_r r)_i + (f_s s))_i|_{io} + \sum_{i=1}^{H} |(f_r r_i + f_s s_i)|_{hash} + \\
& \sum_{i=1}^{H}(\sum_{j=1}^{h} |(f_r r_{ij} * f_s s_{ij}))|_{probe}
\end{aligned}
\tag{7.4}
$$

本方式は，リレーションを入出力クラスタとして二次記憶上に分割し，結合処理を各クラスタ毎に行うことにより，リレーションの大きさに係わらず，扱うリレーションサイズに比例したコストで処理ができる。しかし，スプリットフェイズと結合フェイズを完全に分離しているため，スプリットフェイズではスプリット分割数に等しい主記憶ページが必要となるが，主記憶がこれ以上あっても有効利用されない。また，僅かでも主記憶より大きなリレーションに対し、必ずスプリットフェイズによる読み出しと書き戻しが必要となる。

(4) ハイブリッドハッシュ結合方式

ハイブリッドハッシュ結合方式は， Grace ハッシュ結合方式と単純ハッシュ結合方式を融合（ハイブリッド化）することで，主記憶の有効利用を図った方式である。すなわち， Grace ハッシュ結合方式のスプリットフェイズで有効利用されていない領域に第一番目の入出力クラスタ（これを第一クラスタと呼ぶ）のハッシュテーブルを生成, データをロードすることで，スプリットフェイズと結合フェイズの一部をオーバラップさせ, 高速化を図っている。

本方式では，主記憶に保持する第一クラスタを R_1 とすると，入出力クラスタの個数は，
$H = \lceil \frac{R-R_1}{N-1} \rceil + 1$ となる。

$$Cost_{Hybrid} = \text{式（7.4）} - |2(f_r R_1 + f_s S_1)|_{io} \tag{7.5}$$

本方式では、第一クラスタを二次記憶に書き戻さないため、リレーションが主記憶の数倍程度の場合には Grace ハッシュ結合方式よりも性能が良いが, リレーションが大きくなると相対的に第一クラスタをロードする領域が小さくなり、主記憶上に保持する効果が小さくなり，二つの方式はほぼ同じ性能となる。また，データの分布が不均一な場合，主記憶に丁度収まるように第一クラスタを選ぶことが困難である。

(5) 動的 Grace ハッシュ結合方式

動的 Grace ハッシュ結合方式では，ハイブリッドハッシュ結合方式が主記憶に保持する第一クラスタを固定的に決めているに対し, 実行時に動的に主記憶に保持する入出力クラスタを決定することにより一層の性能向上を図っており, 不均一なデータに対してハイブリッドハッシュ結合方式より性能がよいことが確認されている [87]。本方式の処理コストは, データの分布が一様であるとするとハイブリッドハッシュ結合方式と同じであり、処理コスト式（7.5）を用いる。しかし，実行時に動的にハッシュテーブルとスプリットを実現するため，比較, データ移動等に対するオーバヘッドが大きくなる。

7.2.2 従来のハッシュ結合方式の性能とその問題点

ここでは、各ハッシュ結合方式の性能比較を前述の処理コスト式を用いて行なう。ハッシュを用いた結合方式では、主記憶上の突き合わせ処理をハッシュを用いて行なうため、従来のネストループ方式、ソートマージ方式と比較して cpu 処理コストは大幅に低減し、その処理コストは入出力コストが主体となる。実際、我々は機能ディスクシステム上にハッシュ結合方式の実装を行ない、ウィスコンシンベンチマーク程度のタプル長では完全に入出力バウンドであることを確認している [137]。従って、本節の性能比較では 処理コスト式の中から、入出力コストのみを取り出し、総ページ入出力回数で各方式の比較を行う。

簡単のため、二つのリレーションの大きさは等しい $(R = S)$ とし，選択率は共に $1.0(= f_r = f_s$) とする。また，主記憶は１００２ページで構成され、１ページはリレーション S のためのバッファ

領域,1 ページは出力用バッファで残りはリレーション R の領域として使用する。リレーションは主記憶と同じ大きさから１００倍まで変化させる。

(1) 入出力コストによる性能比較結果

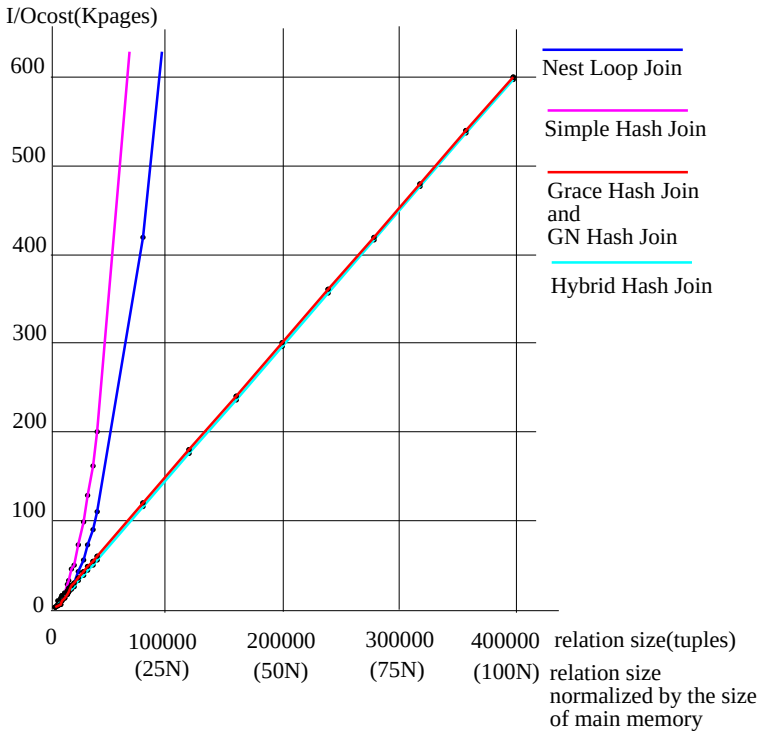

図 7.1: 均一分布の場合の入出力コスト

図 7.1に均一なデータ分布の場合の各ハッシュ結合方式の総ページ入出力回数を示す。縦軸にページ入出力回数を、横軸にリレーション R のサイズをタプル数と主記憶サイズ (N) を１とした場合の相対値で示す。また、図 7.2にリレーションサイズが主記憶の５倍までの拡大図を示す。図 7.1からわかるように，入出力クラスタを一旦生成する Grace ハッシュ結合方式とハイブリッドハッシュ結合方式は，リレーションのサイズに比例してページ入出力回数が増加するが，単純ハッシュ結合方式とネストループ結合方式は，ページアクセス回数が急激に増大し、性能劣化の大きいことが確認できる。また、リレーションが十分主記憶に対して大きい場合には、 Grace ハッシュ結合方式とハイブリッドハッシュ結合方式の性能はほとんど同じことが確認できる。一方，図 7.2からわかるように主記憶の４倍までの小さなリレーションを対象とする場合は，ネストループ結合方式の入出力コストがハイブリッドハッシュ結合方式とほぼ同じであることが確認できる。特にリレーションが主記憶の３倍より小さい場合、ネストループ結合方式がハイブリッドハッシュ結合方式より性能がよい。また、リレーションが主記憶の４倍より大きい場合にはハイブリッドハッシュ結合方式の入出力コストが最小となるが、リレーションが主記憶の１０倍程度より大きい場合には図 7.1からわかるとおり、第一クラスタを主記憶に保持することの効果はわずかである。

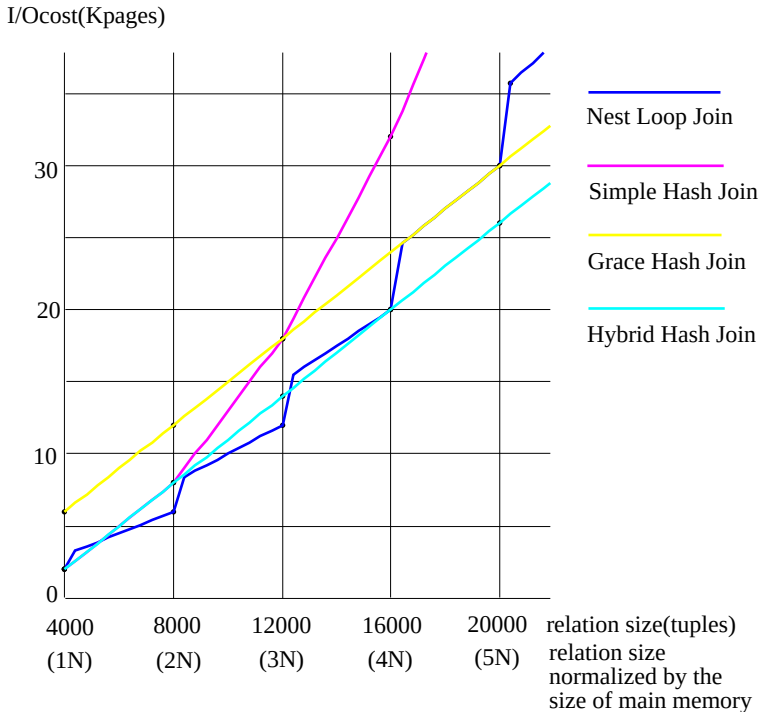

図 7.2: 均一分布の場合の入出力コスト – 拡大図 –

(2) 従来のハッシュ結合方式に対する考察

従来のハッシュ結合方式は，入出力コストを如何に低減するか，また，主記憶を如何に有効に使用するかという観点からのみ論議されており，データの分布に対する対応、実装面での考慮が十分ではない。

データの分布が不均一である場合、スプリットされた入出力クラスタのサイズは必ずしも均等、あるいはハイブリッドハッシュ結合方式や動的 Grace ハッシュ結合方式で期待されるように主記憶容量と等しい大きさに分割できるとは限らない。入出力クラスタを生成しないネストループ結合方式はデータ分布の偏りによる影響を受けないが、入出力クラスタを単位として処理を行なう単純ハッシュ結合方式、 Grace ハッシュ結合方式、ハイブリッドハッシュ結合方式、および動的 Grace ハッシュ方式は以下の如く性能が低下する。例えば，固定的に第一クラスタを決定するハイブリッドハッシュ結合方式および単純ハッシュ結合方式では，第一クラスタが主記憶からわずかでも溢れるとディスクに書き戻す必要があり、主記憶に第一クラスタを保持する効果はなくなる。逆に，第一クラスタが主記憶上の領域よりはるか小さい場合にも、処理コスト全体に対する性能改善が小さく主記憶上に保持する効果はない。また、主記憶から溢れた入出力クラスタの再分割を行なう場合、そのクラスタのデータの分布が均一であることは保証されず、単純に同じ方式を再帰的に適用していては入出力コストが急激に増加する。このように、主記憶利用の効率化手法はデータ分布の予測がはずれた場合、必ずしも期待される性能が得られない。動的 Grace ハッシュ結合方式ではこの点の改良を試みている

が、以下に述べるようにバッファ管理が複雑であり、その効率の良い実装は容易でない。

実装面から考察すると Grace ハッシュ結合方式が入出力クラスタを生成するための単純なバッファ機構で十分なのに対し、ハイブリッドハッシュ結合方式及び動的 Grace ハッシュ結合方式では、第一クラスタを主記憶上にロードするためにスプリット時に複雑なバッファ管理を必要とする。特に，動的 Grace ハッシュ結合方式は実行時にデータの分布によって主記憶にロードする第一クラスタを選択，変更するため，文献 [135] では，スプリットフェイズにおけるオーバヘッドは大きくなってしまい、主記憶の 10 倍程度のリレーションに対し Grace ハッシュ結合方式と比較して 4% 程度の性能向上であると報告している。すなわち、第一クラスタをディスクに書き戻さないで得られる性能向上と実装時のオーバヘッドコストが相殺するため、 Grace ハッシュ結合方式とハイブリッドハッシュ結合方式との性能差は入出力コストのみで比較している図 7.2に示されるほど大きいとはいえず、実装面における簡易性も性能向上の上で重要な要因といえる。

7.3 GNハッシュ結合方式

前節で検討したように、 Grace ハッシュ結合方式、ハイブリッドハッシュ結合方式、動的 Grace ハッシュ結合方式は大容量リレーションの結合処理に関して高い性能を有しているが、いずれもデータ分布が不均一な場合に於ける主記憶から溢れた入出力クラスタ（オーバフロー入出力クラスタ）に対する考慮が不充分である。本節では、データの分布が不均一でオーバフロー入出力クラスタが生成される場合にも性能の低下を小さく押えることを目的とするGNハッシュ結合方式（Grace-Nested loop Hash-Based Join Algorithm) を新たに提案する。本方式は，ハイブリッドハッシュ結合方式と異なり，二つの処理方式 (ネストループ結合方式と Grace ハッシュ結合方式) を融合（ハイブリッド化）するのではなく，動的にいずれか一つを選択, 実行する。本結合方式の性能は、データが均一の場合には、リレーションの大きさが主記憶の数倍程度まではネストループ結合方式を適用することにより、又、それより大きいリレーションでは Grace ハッシュ結合方式を適用することでほぼハイブリッドハッシュ結合方式の性能と同等になる。さらにデータの分布が偏っている場合、オーバフロー入出力クラスタの処理において、ネストループ結合方式または Grace ハッシュ結合方式の入出力コストの小さい方を再帰的に選択するため、主記憶の数倍程度のクラスタではネストループ結合方式が適用することにより再分割コストが不必要となり、従来のハッシュ結合方式と比較して入出力コストを低減することが可能となり、全体性能の向上が期待できる。また, アルゴリズム選択評価式自身は以下に述べるように単純であり、ハイブリッドハッシュ方式や動的 Grace ハッシュ方式と異なり、特に複雑なバッファ管理をすることなく実装できるという利点がある。すなわち、ハイブリッドハッシュ結合方式およびに動的 Grace ハッシュ結合方式では主記憶の有効利用のためにスプリットテーブルと結合処理のためのハッシュテーブルの二つを同時に生成するため、粒度の異なる二つのテーブルのための領域割り当ておよびオーバフロー管理、あるいはオーバフロー時の主記憶領域のハッシュテーブルからスプリットテーブルへの切替え等が必要となる。一方、 GN ハッシュ結合方式では実行時には二つの処理方式の一方のみ実行するため、ネストループ結合方式が選択された場合は結合処理のため

のハッシュテーブル処理のみ、 Grace ハッシュ結合方式の場合はスプリットテーブル処理とハッシュテーブル処理が順に行なわれるため、そのバッファ管理方式ははるかに簡便になる。

```
GN ハッシュ結合方式 (data =source relation)
    アルゴリズム選択フェイズ (data)
    Begin
        ソースリレーションなら選択率を推定し、
        data の情報よりアルゴリズムの選択を行う
        if ネストループ結合方式の場合
            Begin /* f_r R ≤ (1 + 4f_r)N であれば */
                ネストループの回数を計算
                内側リレーションの書き戻し条件を確認
            End
        else Grace ハッシュ結合方式が選択された場合には
            Begin /* f_r R > (1 + 4f_r)N であれば */
                入出力クラスタ個数を決定
                data を入出力クラスタに分割
                for each オーバフロー入出力クラスタ
                    アルゴリズム選択フェイズ (オーバフロー
                        入出力クラスタ)
            End
    End
    結合フェイズ (data)
    Begin
        for data で指定される外側のデータ
            for data で指定される内側のデータ
                外側の data のハッシュテーブルを生成
                内側の data を読み出し、結合処理を行なう
                内側 data 書き戻し指定があれば書き戻し処理
    End
Begin
    アルゴリズム選択フェイズ (data)
    for each data
        結合フェイズ (data)
End
```

表 7.2: GN ハッシュ結合方式

本方式を疑似コードを用いて表 7.2に示す。表内に二つのアルゴリズム選択の条件式を $f_r = f_s$ としてコメントで示している。本方式は，実行時の処理方式の選択決定とソースリレーションの分割、入出力クラスタの生成を行うアルゴリズム選択フェイズとその結果に従いリレーションあるいは入出力クラスタをディスクから読み出し、主記憶上で処理する結合フェイズに分けられる。以下に、それぞれのフェイズの処理の流れおよびアルゴリズム選択評価式と入出力クラスタの生成（データの分割）について、詳細に述べる。

7.3.1 アルゴリズム選択フェイズ

本フェイズでは，ネストループ結合方式と Grace ハッシュ結合方式の 2 つから実行時の処理方式を選択し、選択された処理方式に従って必要であればデータの分割を行なう。すなわち、与えられたデータがソースリレーションであれば、選択率を実際一部のデータを読み込むことにより推定し，オーバフロー入出力クラスタであれば選択率を 1.0 として次節 7.3.1で述べる条件式に従い、アルゴリズムの選択を行う。

Grace ハッシュ方式が選択された場合には，生成すべき入出力クラスタの個数などを決定し、 7.3.1に

詳細に述べるように入出力クラスタの生成を行なう。この際、オーバフロー入出力クラスタが生成されると、再帰的に本フェイズが呼び出され、改めて選択評価式を用いて、当該オーバフロー入出力クラスタに対する処理方式が決定される。

ネストループ結合方式が選択された場合には，データの分割は行われず，結合フェーズ時にステージングする際の外側ループのデータと内側ループのデータを決定する。また、内側ループのリレーションの書き戻しが必要かどうかを決定する。

(1) アルゴリズム選択評価式

本 GN ハッシュ結合方式では、アルゴリズム選択評価式として 7.2節で述べたネストループ結合方式, Grace ハッシュ結合方式の処理コスト式から cpu による処理コストを除き入出力コストのみ用いたコスト式を利用する。ネストループ結合方式と Grace ハッシュ結合方式を比較したとき，ネストループ結合方式の入出力コストが良い条件は、 式(7.1) $\leq$ 式(7.4) より，

$$n \leq 1 + 2f_s + 2f_r \quad (\leq 5) \tag{7.6}$$

となる。選択率を $f_r = f_s$ とすると、 $n = \lceil \frac{f_r R}{N} \rceil$ より、

$$f_r R \leq (1 + 4f_r)N$$

が成り立つ場合には，ネストループ結合方式の入出力コストが小さいと言える。図 7.2からも、 $f_r = 1.0$ の場合には、 $R \leq 5N$ のとき、ネストループ結合方式の方が Grace ハッシュ結合方式よりも性能がよいことがわかる。

次にネストループ結合方式で選択率を考慮し，内側のループにおけるリレーションの書き戻しを行うための条件は， 式(7.2) $\leq$ 式(7.1) より

$$f_s < 1 - \frac{1}{n} \quad (2 \leq n) \tag{7.7}$$

このとき、ネストループ方式の方が入出力コストが小さい条件を求めると、 式(7.2) $\leq$ 式(7.4) より，

$$n \leq 2 + 2\frac{f_r|R|}{f_s|S|} \leq 4 \tag{7.8}$$

が求められる。

本論文では，結合演算の評価式について示したが，重複除去, 集計演算等の関係代数演算についても同様の議論ができる [137]。

(2) 入出力クラスタの生成（データの分割）

Grace ハッシュ結合方式が選択されるとリレーションの分割が行なわれる。その際に重要な点は生成する入出力クラスタが主記憶から溢れないようにすることである。一方，結合フェイズで主記憶を有効に利用するためには生成する入出力クラスタは，主記憶と同じ程度の大きさであることが望ましい。

本方式でも入出力クラスタの生成方式に関しては、従来からのバケットチューニング方式 [87] を想定している。すなわち，リレーションの大きさから主記憶上の入出力クラスタの数としてすでに計算されている理想の入出力クラスタ数よりも数倍程度多くなるように決定し，主記憶上に読み出し，主記憶上の小さいクラスタをいくつか結合して入出力クラスタとする。

データの分布によっては均等に入出力クラスタに分割できず，オーバフロー入出力クラスタが生成される場合がある。このオーバフロー入出力クラスタに対しては表 7.2で示しているように再帰的にアルゴリズム選択フェイズを呼びだすことにより，改めてオーバフロー入出力クラスタの大きさに適した処理方式が選ばれる。これにより、主記憶の５倍程度までのオーバフロー入出力クラスタであれば、ネストループ結合方式を適用することで、従来の Grace ハッシュ結合方式やハイブリッドハッシュ結合方式と異なり、その入出力クラスタの再分割が必要なくなり、データ分布が均一な場合と同じ入出力コストで処理が可能となる。また、偏りが激しく何回も再分割が必要な場合、それが主記憶の５倍程度までの大きさの入出力クラスタまで分割されれば、ネストループ結合方式で処理可能となる。このように、オーバフロー入出力クラスタの処理に関して、本方式は従来の方式に比べて再帰的処理の収束が速く、入出力コストを小さく押えられると言える。

7.3.2 結合フェイズ

全ての入出力クラスタの生成が終ると、本フェイズが入出力クラスタ回だけ呼び出され、それぞれの入出力クラスタに対してアルゴリズム選択フェイズで決定された処理方式に従って結合演算処理を行う。すなわち，生成された各入出力クラスタ或いはソースリレーションに対してネストループ結合方式が指定されている場合には主記憶上に外側リレーション（または入出力クラスタ）の一部をハッシュテーブルに展開し，ネストループ結合方式と同じ処理を行う。その際、式（7.7）により内側リレーション S の書き戻しが指定されている場合には、一回目のループで選択後のリレーション S の書き戻しを行なう。ネストループ結合方式の指定がない場合には，その入出力クラスタ或いはソースリレーションは主記憶に収まるため， Grace ハッシュ結合方式の結合フェイズの処理が行われる。

7.4 入出力コストによる GN ハッシュ結合方式の性能評価

本節では， 7.2節に述べた各ハッシュ結合方式及び新たに提案した GN ハッシュ結合方式に対してデータの分布を変化させて入出力コストによる性能比較を行い,GN ハッシュ結合方式の有効性について示す。

7.4.1 性能評価方法

本来、各結合方式の処理においては、解析式からわかるように CPU の処理コストが含まれているが、実装の観点からは文献 [?] に於いても確認されているように入出力コストが主体でありそれ以外は無視できる。ここでは入出力コストのみで比較を行なう。

7.2.2節と同じパラメタ及び条件を用い, 総ページ入出力回数による性能比較を行う。従って、主記憶は１００２ページ、それぞれのリレーションは１ページに４タプル格納されている。また、結果リレーションの書き込みコストについては、いずれの方式でも同じため省略する。二つのリレーションのサイズは等しいとし、選択率は 1.0($= f_r = f_s$) としている。選択率に対する入出力コストの変化は最初のリレーションの読み出しのみに効いており、最初のリレーションの読み出しコスト自体はいずれの方式でも同じため、選択率の変化はオーバフロー入出力クラスタの処理に対する各結合方式の性能に影響を与えない。また、入出力クラスタの個数は均等に分割できると期待されるときの個数 $H = \lceil \frac{R}{N} \rceil$ とする。

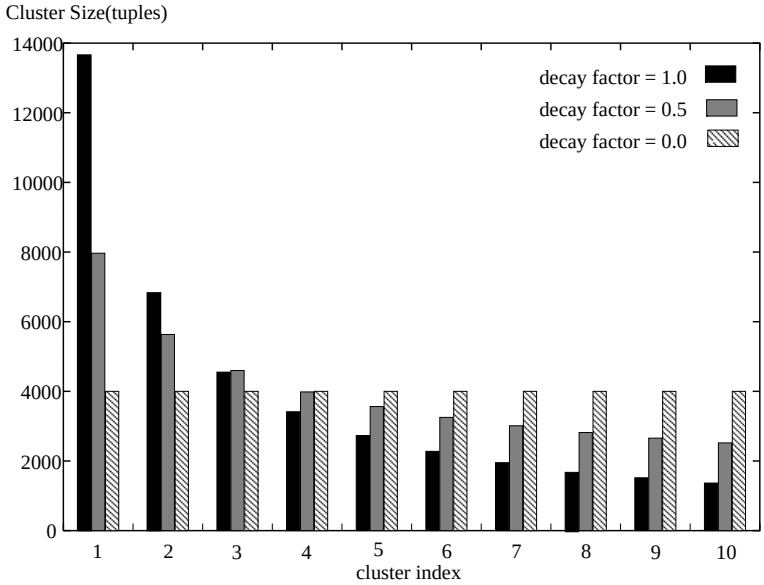

図 7.3: Zipf-like 分布に於ける decay factor とクラスタサイズ （リレーションサイズ＝４０、０００タプル）

今回の解析では、近年データベースシステムの性能評価において不均一なデータ分布としてしばしば用いられている Zipf-like 頻度分布を用いた [20, 37, 124, 127, 49, 76]。各クラスタサイズ（タプル数） c_i は、クラスタの個数を H、リレーションのタプル数を r とすると、

$$c_i = \frac{r}{i^{decayfactor} \times \sum_{j=1}^{H} \frac{1}{j^{decayfactor}}}$$

図 7.3にリレーションの大きさが主記憶の１０倍の場合に、 decay factor を変化させた場合のクラスタサイズを示す。オーバフロー入出力クラスタのデータ分布は、ソースリレーションと同様に Zipf-like 分布に従うものとする。従って、リレーションが主記憶の１００倍程度の大規模リレーションでは、最大のオーバフロー入出力クラスタは主記憶サイズよりはるかに大きく、複数回に渡って再帰的にスプリットしなくてはならない。図 7.4にリレーションサイズが主記憶の１００倍とし、 decay factor が 1.0 の場合におけるクラスタの分割状況を、それぞれのスプリットの時点でクラスタインデク

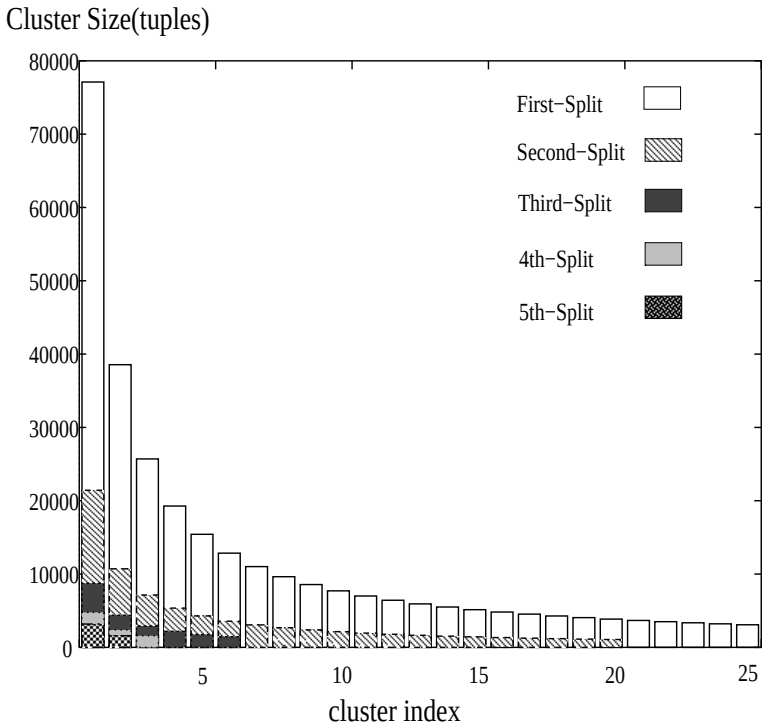

図 7.4: オーバフロー入出力クラスタの再分割状況

スが１のクラスタ（すなわち、最大のクラスタ）を再帰的にスプリットした場合を示す。最初のスプリットでは１００個のクラスタが生成されるが、２５までのクラスタのみ図示する。ハイブリッドハッシュ結合方式では、いずれの入出力クラスタが第一クラスタとなるかによって、その性能が変わるため、ここではそれぞれの入出力クラスタが第一クラスタになった場合の性能を測定し、それを平均してハイブリッドハッシュ結合方式の性能としている。

今回の性能評価では、データの偏りが激しい場合として decay factor が 1.0 の場合及びデータの偏りが中程度として decay factor が 0.5 の場合の各ハッシュ結合方式の性能をクラスタサイズの偏りは一定とし、リレーションサイズを変化させて詳細に解析した。さらに、主記憶の１００倍のリレーションについてリレーションサイズを一定とし decay factor を 0.0 から 1.0 まで変化させて性能評価を行なった。

7.4.2 リレーションサイズを変化させた場合の性能比較

ここでは、 decay factor を一定（0.5,1.0）とし、リレーションサイズを変化させた場合の性能比較を行なう。リレーションサイズは、主記憶サイズと同じ大きさから主記憶の１００倍まで変化させた。

(1) decay factor が 0.5 の場合

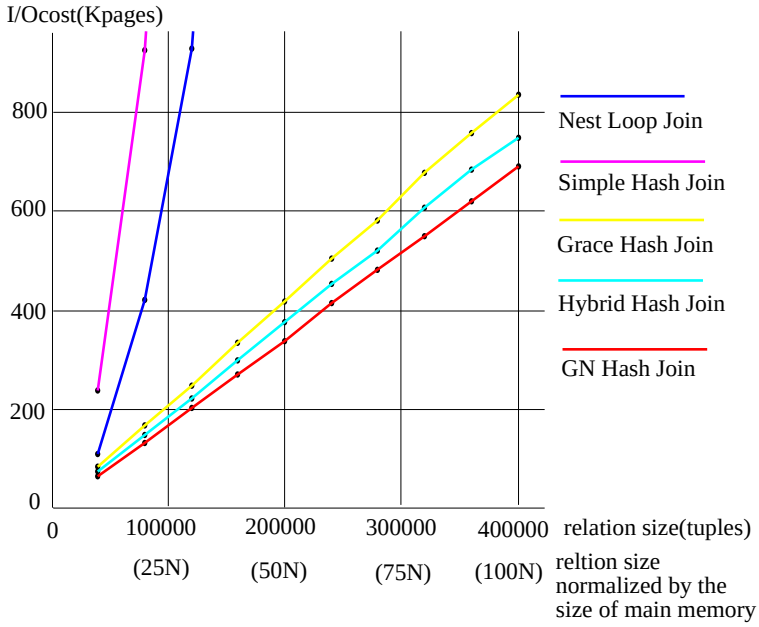

図 7.5: 不均一分布（decay factor = 0.5）の場合の入出力コスト

図 7.5に Zipf-like 分布（decay factor = 0.5）の場合の各ハッシュ結合方式の総ページアクセス回数を示す。また、図 7.6に、図 7.5では図示していないリレーションが主記憶の１倍から１０倍までの結果を示している。

図 7.5からわかる通り、いずれのハッシュ結合方式でもオーバフロー入出力クラスタに対してスプリットを再帰的に繰り返しているため、均一な分布の結果と比較すると入出力コストが増加している。しかし、 GN ハッシュ結合方式は、 Grace ハッシュ結合方式やハイブリッドハッシュ結合方式と比べ、その入出力コストはかなり小さくなっている。例えば、リレーションサイズが主記憶の１００倍の場合、１００個生成された入出力クラスタの内、２８個が Grace ハッシュ結合方式やハイブリッドハッシュ結合方式では再分割を行なわねばならないオーバフロー入出力クラスタとなる。また、最大のオーバフロー入出力クラスタは主記憶の６倍の大きさであり、主記憶に収まるまで再帰的に３回のスプリットを行うことになる。しかし、 GN ハッシュ結合方式ではネストループ結合方式を適用することにより、最大のオーバフロー入出力クラスタに対し１回の分割で済み、又、他のオーバフロー入出力クラスタはスプリットせずに処理できる。主記憶の１００倍の大きさのリレーションの処理では、均一の場合に比較して GN ハッシュ結合方式の入出力コストは、１３パーセント程度しか増加していないが、ハイブリッドハッシュ結合方式で２５パーセント、 Grace ハッシュ結合方式では４０パーセント程度入出力コストが増加している。

一方、図 7.6から GN ハッシュ結合方式はリレーションが小さい場合にもその入出力コストが最も小さく、ネストループ結合方式を用いることによりデータ分布が偏っていても、それほど性能に影響を受けないことを示している。また、この主記憶の数倍程度の大きさでの GN ハッシュ結合方式とハイブリッドハッシュ結合方式の性能差が、多数のオーバフロー入出力クラスタができる大容量リ

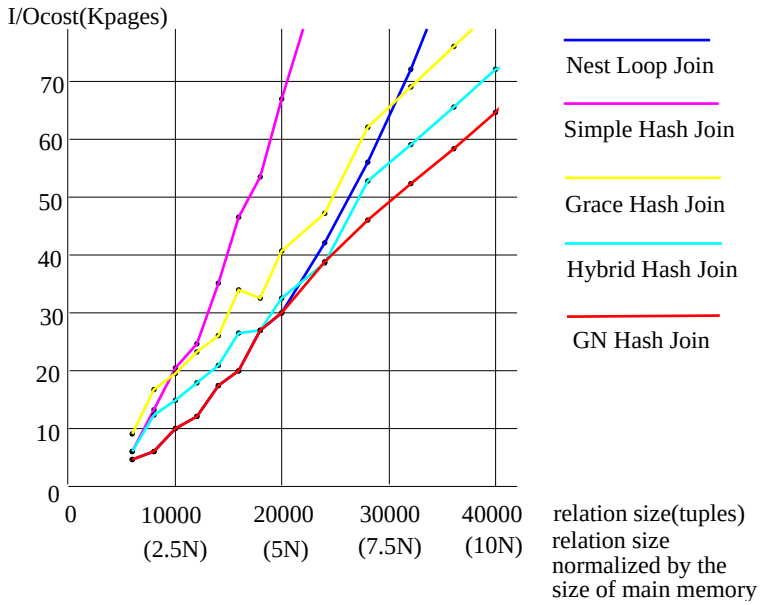

図 7.6: 不均一分布 (decay factor =0.5) の場合の入出力コスト – リレーションが主記憶の１倍から１０倍まで –

レーションでの処理において、累積効果として現れているといえる。

(2) decay factor が 1.0 の場合の性能比較

図 7.7に Zipf 分布（decay factor = 1.0）の場合の各ハッシュ結合方式の総ページ入出力回数を示す。また、図 7.7に示されていない部分、すなわちリレーションが主記憶の１倍から１０倍までの結果を図 7.8に示している。図 7.8に於いて、最大のリレーション（主記憶の１００倍の大きさ）の処理を行なう場合、従来のハッシュ結合方式では最大のオーバフロー入出力クラスタが主記憶に収まるまでに５回のスプリットを再帰的に行なう必要があるが、 GN ハッシュ結合方式ではネストループ方式を適用することにより２回スプリットするだけで処理が行なえる。主記憶の１００倍程度のリレーションの場合、均一な分布の場合の結果と比較し、ハイブリッドハッシュ結合方式で７０パーセント、Grace ハッシュ結合方式で８０パーセント程度入出力コストが増加しているのに対し、 GN ハッシュ結合方式では４０パーセントと入出力コストの増加が半分に押えられている。

一方、図 7.8からわかるように、 GN ハッシュ結合方式は、ネストループ結合方式を用いることによりデータ分布が偏っていても、性能に影響を受けないことが確認できる。また、 decay factor が 0.5 の場合に比較して、リレーションが主記憶の５倍程度の大きさまでの場合、 GN ハッシュ結合方式は他のハッシュ結合方式と比較し、さらに優位であることも確認できる。

7.4.3 decay factor を変化させた場合の性能比較

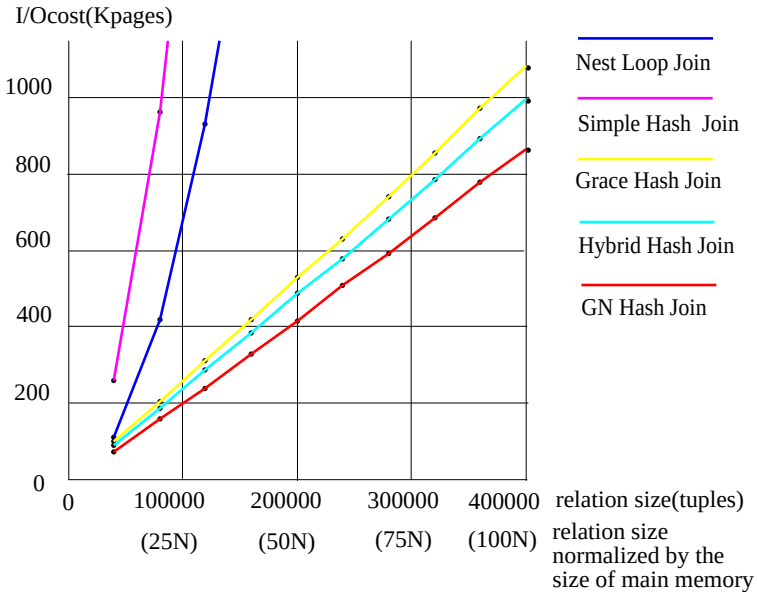

図 7.7: 不均一分布（decay factor =1.0）の場合の入出力コスト

ここでは、 decay factor を 0.0 から 1.0 まで変化させた場合の Grace ハッシュ結合方式、ハイブリッドハッシュ結合方式および GN ハッシュ結合方式の性能比較を行なう。図 7.3からわかるように decay factor が大きくなるにつれて、クラスタサイズは均等分割（decay factor=0.0) から偏りが大きく (decay factor=1.0) なる。リレーションサイズは主記憶の１００倍の 400,000 件とした。図 7.9に示す通り、データの偏りがなく、リレーションが均等に入出力クラスタに分割出来る場合にはいずれの方式でもほぼ同じ性能である。しかし、 decay factor=0.01 の結果からわかるようにわずかでもデータの偏りがあると、オーバフロー入出力クラスタが生成され、入出力コストが増える。データの偏りに係わらず、他の二つの方式と比較して GN ハッシュ結合方式の性能が高いことがわかる。特に、データの偏りが激しくなるとハイブリッドハッシュ結合方式では、第一クラスタが主記憶を越えてしまうか、主記憶を有効利用できないような小さなクラスタになってしまうため、性能の劣化が他の二つの方式よりも大きい。

7.5 GN ハッシュアルゴリズムの実行時選択評価式

本節では、 GN ハッシュアルゴリズムにおける実行時選択条件について検討を行う。本節では以下に示す等結合演算処理を用い、結合演算コストに利用するパラメタを続いて示す。

```
SELECT * FROM R,S WHERE R.joinkey = S.joinkey
          AND R.a1 < C1 AND S.b1 > C2....
```

- $|R|$, $|S|$ はそれぞれリレーション R および S のサイズ (単位 ページ) を示す

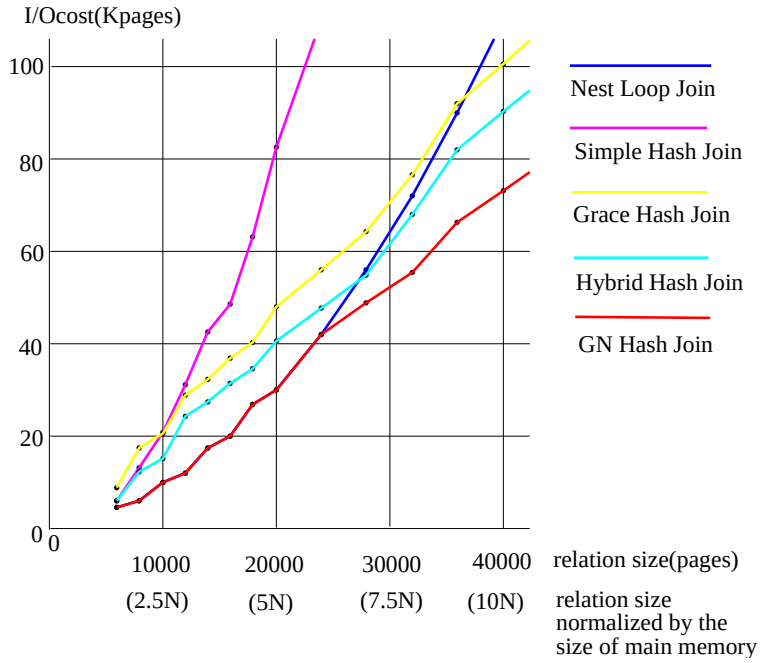

図 7.8: 不均一分布（decay factor =1.0）の場合の入出力コスト – リレーションが主記憶の１倍から１０倍まで –

- $M = N + 2$ は主記憶サイズ (単位 ページ) を示す。 N はリレーション R をロードする領域であり、１ページをリレーション S の読込領域、最後の１ページを結果リレーションの書込領域として用いる。
- $|X|_{io}$ は $|X|$ ページのディスクに対する読込あるいは書込コストを示
- h は主記憶上のバケット数を示す。
- H はディスク内のクラスタ数 (以降、入出力クラスタと呼ぶ) を示す。
- f_s, f_r はそれぞれリレーション R および S の選択率を示す。

入出力コストが結合演算の性能において支配的要素であるため、実行時にネストループアルゴリズムと Grace ハッシュアルゴリズムのいずれかを選ぶために入出力コストを用いる。ここで $|R| \leq |S|$ とする。ネストループアルゴリズムでは、リレーション R はインナーループでディスクから一回のみ読み込まれ、その際のアウターループ数は $n = \lceil \frac{f_r R}{N} \rceil$ で表される。ネストループハッシュアルゴリズムの入出力コスト式は、以下と表される。

$$cost_{NL} = |R|_{io} + n|S|_{io} \tag{7.9}$$

Grace ハッシュアルゴリズムでは、入出力クラスタ数は $H = \lceil \frac{f_r R}{N} \rceil$ となる。 Grace ハッシュアルゴリズムの入出力コスト式は以下で表される。

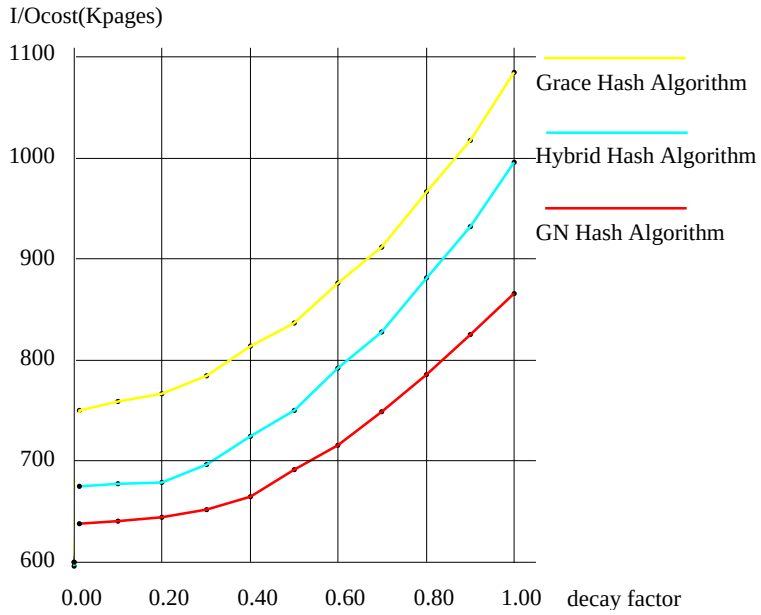

図 7.9: decay factor を変化させた場合の入出力コスト

$$cost_{Grace} = |R|_{io} + |S|_{io} + 2\sum_{k=1}^{H} |((f_r r)_k + (f_s s)_k)|_{io} \tag{7.10}$$

From eq(7.9) $<$ eq(7.10) and $|R|_{io} \leq |S|_{io}$ and ,

$$n|S|_{io} < |S|_{io} + 2\sum_{k=1}^{H} |((f_r r)_k + (f_s s)_k)|_{io}$$

$$n|S|_{io} < |S|_{io} + 2f_r|R|_{io} + 2f_s|S|_{io} \tag{7.11}$$

From eq(7.11) and $|R|_{io} = |S|_{io}$,

$$n < 1 + 2f_r + 2f_s < 5 \tag{7.12}$$

以上より、 eq(7.12) と $n = \lceil \frac{f_r R}{N} \rceil$ および $f_r = f_s = 1$ から、リレーション R のサイズが主記憶の 5 倍より小さい場合には、容易にネストループアルゴリズムの入出力コストが Grace ハッシュアルゴリズムよりよいことがわかる。しかしながら、この評価条件はデータが均一に分布していることを仮定した場合のものである。

7.6 GN ハッシュアルゴリズムの実行時選択評価式の検証

本節では、上述の選択評価式について検討する。まず、データ分布を変化させた場合のネストループアルゴリズムと Grace ハッシュアルゴリズムの入出力コストの交差ポイントがどのように変化するかについて調べる。ついで、 GN ハッシュアルゴリズムの性能を解析するために、シミュレーション

により、 Zipf- 分布データを用いた場合に選択評価条件を変化させると GN ハッシュアルゴリズムの性能がどのように変化するかについて調べる。

7.6.1 評価環境と前提

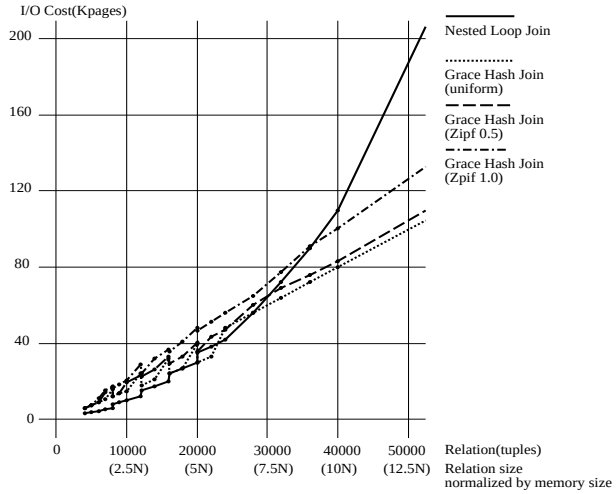

図 7.10 ネストループアルゴリズムと Grace ハッシュアルゴリズムのシミュレーション結果

以下の実験では、次の結合演算問合せ処理を行うものとする。

```
SELECT * FROM R,S WHERE R.joinkey = S.joinkey
```

シミュレーション環境は以下のとおりである。

- メインメモリ 1002 ページ
  (ステージングバッファ (1000 ページ)+ リレーション S 読込バッファ (1 page) +
  結果リレーション 書込バッファ (1 page))
- フィルタリングファクタ 1.0.
- 1024 バイト / ページ
- 256 バイト / タプル
- リレーション R とリレーション S のサイズは等しい

我々の目的は異なるサイズの入出力クラスタに対する処理コストの比較を行うため、ここでは均一分布と共に Zipf 分布 [65, 20, 37, 124] を用いた。

均一分布の場足、一つのリレーションの入出力クラスタのサイズは４０００タプルである。しかし、均一分布とはいえども、全ての入出力クラスタのサイズは完全に等しいわけではない。それは、i 番目の入出力クラスタに属するタプル x は $P(x)_i = \frac{1}{H}$ によって算出されることによる。ここで H はクラスタの数であり、i は入出力クラスタのランク $(0 < \mathrm{i} \leq \mathrm{H})$ とする。

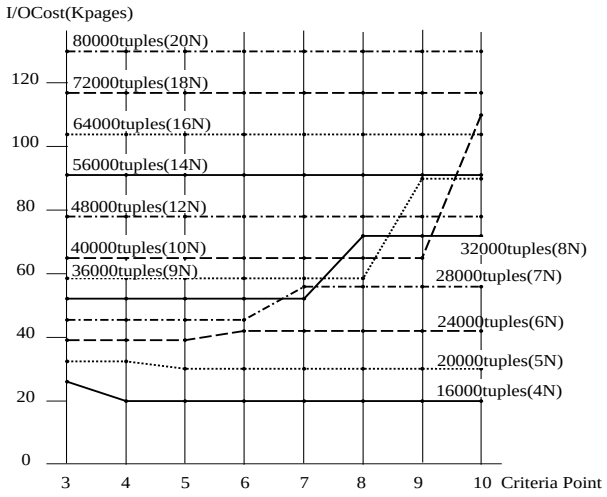

図 7.11 均一分布におけるシミュレーション結果

Zipf 分布の場合、 decay factor が大きくなるほど、データの偏りも大きくなる。ここで、 decay factor が 1.0 の場合が一般に Zipf 分布と呼ばれ、 decay factor が 0.0 の場合、均一分布となる。 i 番目の入出力クラスタにタプル x が属する確率は $P(x)_i = \frac{1}{i^{decayfactor} \times \sum_{j=1}^{H} \frac{1}{j^{decayfactor}}}$ によって、算出される。ここで、バッファから溢れる入出力クラスタのデータ分布はソースリレーションのデータ分布と同じと仮定する。引続き、溢れた入出力クラスタは再帰的に分割される。

評価に用いるデータは、それぞれのデータ分布の確率式を用いて人工的生成した。また、異なるランダム数列によって生成されたデータで３００回、各アルゴリズムのシミュレーションを走らせ、その平均値を結果として用いた。

7.6.2 データの偏りがある場合の二つの方式のクロスポイントの変化

データの偏りが生じると、バッファから溢れた入出力クラスタはさらにバッファに入るまで繰り返し分割するため、余分な入出力コストがかかり、 Grace ハッシュアルゴリズムの入出力コストは増加する。一方、ネストループアルゴリズムの入出力コストはデータの偏りで変化しない。このため、データの偏りを変化させると、以下に示すように、二つのアルゴリズムのコストが入れ違うクロスポイントは変化する。

図 7.10は異なる decay factors を用いた場合のネストループアルゴリズムと Grace ハッシュアルゴリズムのシミュレーション結果を示す。リレーションサイズはメモリと同じサイズから１０倍まで変化させている。 decay factor はそれぞれ 0,0.5, 1.0 の場合のシミュレーションを行った。 decay factor が変化してもネストループあり後リズムの結果は変化がないため、図中、ネストループの結果を示す線は一つのみである。一方、 Grace ハッシュの性能はデータ分布の影響を受ける。 decay factor が大きくなると、 Grace ハッシュアルゴリズムの入出力コストも増加する。その結果、二つのアルゴリズムの入出力コストが交差する点は変わる。均一分布の場合、以前の議論および図 7.10から分

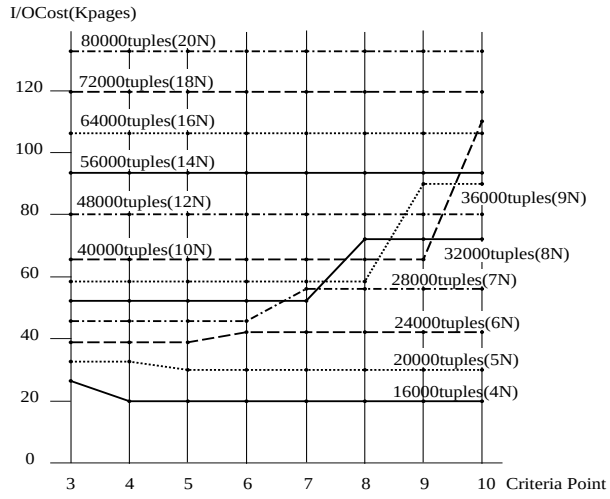

図 7.12 Zipf 分布 (decay factor=0.5) におけるシミュレーション結果

かるように 5N(メモリサイズの五倍) がクロスポイントとなる。また、 decay factor が 0.5 のとき、クロスポイントは 7.5N となり、 decay factor が 1.0 のとき、クロスポイントは 9N となる。このように、二つのアルゴリズムが交差する点はデータの偏りが大きくなるにつれ、大きくなる。しかしながら、データ分布が偏る場合、ここで求められたクロスポイントが GN ハッシュアルゴリズムの評価式として最適とは限らない。

例えば、 Zipf 分布での評価式として、 decay factor が 1.0 の場合のクロスポイントを採用した場合について考察する。この場合、評価式は 9N となるため、 GN ハッシュアルゴリズムの入出力コストはメモリサイズの９倍よりもリレーションが小さい場合には、ネストループアルゴリズムの入出力コストと同じになる。しかしながら、９ N は最適ではない。９ N のクロスポイントでは、 Grace ハッシュアルゴリズムは全ての溢れた入出力クラスタを分割する入出力コストも含んでいる。したがって、メモリサイズの９倍よりも小さいリレーションに対し、一旦 Grace ハッシュアルゴリズムを適用した後のオーバフロー入出力クラスタに対し、ネストループアルゴリズムを適用する機会を失っていると考えられる。続く節において、均一分布における評価式が不均一分布においても有効かどうかについて、評価を行う。

7.6.3 性能評価

均一分布の場合

均一分布において、全てのクラスタサイズは同じことが期待される。しかしながら、この状況は非常に理想的な場合であり、ハッシュ関数を適用した結果、あるタプルが i 番目のクラスタとなる確率の一様性が、クラスタサイズを等しくすることと意味しないことは容易に推測できる。今まで、ハッシュアルゴリズムの性能評価のほとんど全てはこの理想的な場合についてなされてきた。だが、たとえクラスタがタプル一つ分でもステージングバッファを溢れれば、ハッシュ結合アルゴリズムはそのクラスタをディスクへと書き戻し、再度分割しなくてはならない。したがって、データ分布が一様で

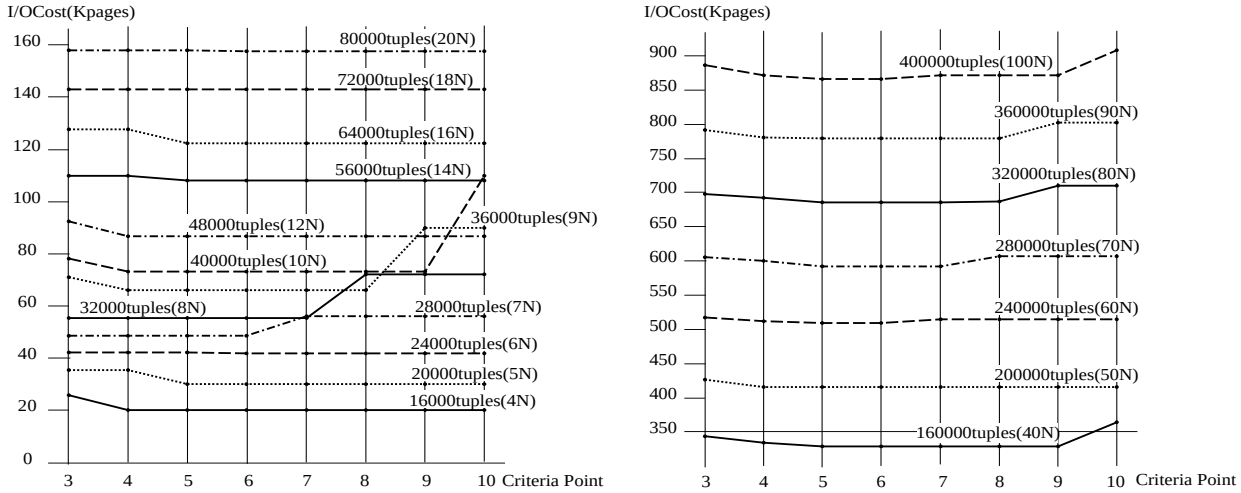

図 7.13 Zipf 分布 (decay factor = 1.0) におけるシミュレーション結果

あったとしても、クラスタが溢れる問題は生じる。

図 7.11は、評価式の値を３から１０まで変化させた場合の均一分布におけるシミュレーション結果を示す。例えば、評価式の値が４であるとは、リレーションサイズがメモリの４倍よりも小さい場合にはネストループアルゴリズムを、それ以外の場合は Grace ハッシュアルゴリズムが利用されることを意味する。この図では、ステージングバッファと同じサイズから２０倍までリレーションサイズを変化させ、その値を図中の線の右または左に記している。主記憶の２０倍ようり大きなリレーションのシミュレーション結果は、この評価式の値には影響を与えないため、結果を省略する。

図 7.11から 7.6.2 で述べた最適評価式が５であることが確認できる。また、この図から、メモリサイズの７倍よりもリレーションのサイズが小さい場合には、 GN ハッシュアルゴリズムの性能が評価式にさほど影響を受けないことが分かる。リレーションのサイズがメモリサイズの４倍または５倍の場合、評価式 (N=3) の結果が他の場合よりも大きくなっている。これは、 GN ハッシュアルゴリズム開始時に Grace ハッシュアルゴリズムを選択し、その結果主記憶の三倍より大きなソースリレーションが分割されてしまうため、ネストループアルゴリズムよりも入出力コストが増えてしまうためである。

リレーションサイズが主記憶の五倍よりも大きい場合 (6N,7N,.... 等) の場合、評価式として 5N よりも大きな値をとると、入出力コストは増加する。これは、均一分布の場合、およそ半分のクラスタがわずかだけ主記憶を溢れ、 GN ハッシュではこれらの溢れたクラスタにネストループアルゴリズムが適用されることにおる。もし、評価式が 5N より大きくなると、ソースリレーションにネストループアルゴリズムを適用する範囲が増え、そのコストは、 Grace ハッシュアルゴリズムを最初のソースリレーションの分割に適用するよりもコストが高くなってしまうためである。

一方、大きなリレーション (図 7.11では 80000 タプルのライン) では、 GN ハッシュアルゴリズムは評価式の変化による影響がない。これは、溢れたクラスタのサイズがほとんど主記憶と同じため、

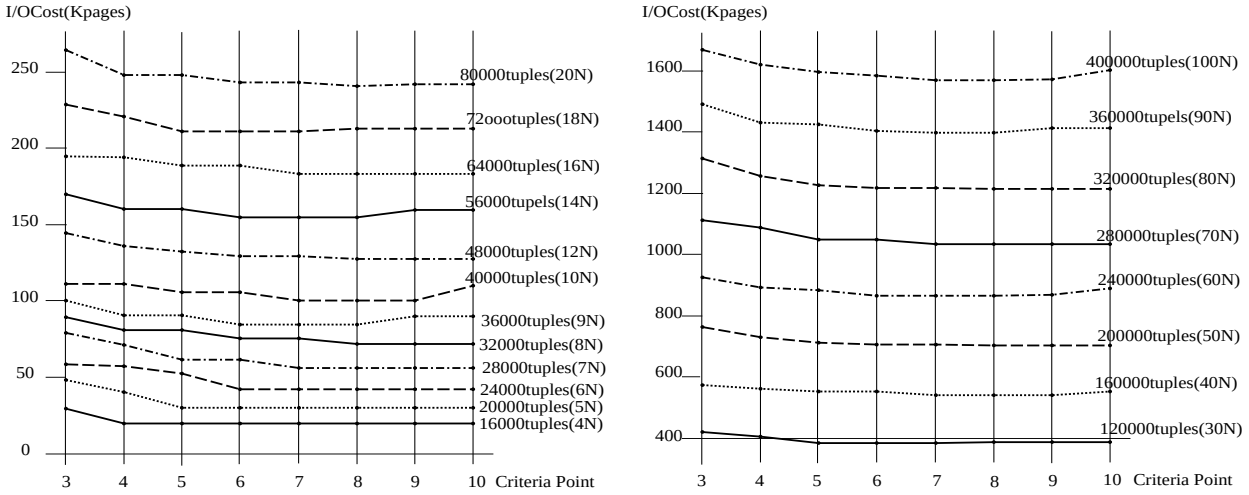

図 7.14 Zipf 分布 (decay factor = 2.0) のシミュレーション結果

つまり、主記憶の二倍より小さいため、いかなる評価式でも、ネストループアルゴリズムが採用されることによる。

Zipf 分布の場合

以下のシミュレーションでは、 decay factors が 0.5, 1.0, 2.0 の場合の Zipf 分布を用いる。図 7.12は decay factors が 0.5 の場合のシミュレーション結果を示す。リレーションサイズはステージングバッファと同じ大きさから、ステージングバッファの２０倍まで変化させている。図 refthdz100graph は Zipf 分布 (つまり、 decay factor が 1) の GN ハッシュアルゴリズムの結果を示す。リレーションサイズはステージングバッファと同じ大きさから、その百倍の大きさまで変化させている。

図 7.12から、最適な評価式がステージングバッファの５倍であることが分かる。また、図 7.12の結果が均一分布の結果に大変類似していることが分かる。これは、 decay factor が 0.5 とさほど大きくないため、リレーションサイズがステージングバッファの５から６倍より小さい場合に、溢れたクラスタの分割が繰り返し行う必要がないことによる。サイズの大きなリレーションに対しては、評価式を変化させても、 GN ハッシュアルゴリズムの全入出力コストは影響を受けない。

図 7.13から、最適な評価式がここでもステージングバッファの五倍であることが分かる。リレーションサイズがステージングバッファの５倍より小さい場合には、図 7.13の結果は図 refthdz50graph の結果と同じである。つまり、５倍よりも小さなリレーションでは、溢れたクラスタのサイズがいずれもメモリの２倍よりも小さいことを意味する。リレーションサイズがメモリの１０倍よりも大きい場合には、図 7.13の結果は図 refthdz50graph の結果と少し異なっている。これは、 decay factor が 1.0 のため、データの偏りが生じ、溢れたクラスタの幾つかは一回以上分割されることによる。大きなサイズのリレーションの場合、評価式の値を変更することにより GN ハッシュアルゴリズムの結果は大きな影響は受けないが、 decay factor が 0.5 の結果と比較すると、５ N が最適な評価式となる範

囲が狭まっている。

図 7.14は decay factor が 2.0 の Zipf 分布における GN ハッシュアルゴリズムの結果を示す。リレーションサイズはステージングバッファサイズからその１００倍の大きさ間で変化させた。この場合、データは大変偏っているため、ステージングバッファから溢れた入出力クラスタは何回も再度分割される。例えば、リレーションサイズがステージングバッファの１００倍 (図中では 400,000 タプル) の場合、最も大きな溢れた入出力クラスタは主記憶に収まるまでに２６回再分割が繰り返される。実際のシステムではこのような状況は起きないかもしれないが、評価式が性能にどの程度の影響を与えるかを確認するために、この結果を示した。

図 7.14はリレーションサイズがステージングバッファの５から６倍より小さい場合、すでに示した図とは異なっている。これは、リレーションサイズが小さい場合であっても、溢れた入出力クラスタの分割が何回も行われることによる。この結果、評価式が３ N の場合、再分割コストが積算されるため、全ての場合の結果の中で最も入出力コストが大きくなる。また、図から最適評価式の範囲は６から８と広くなっていることが確認できる。

7.6.4 Decay Factor を変化させた場合

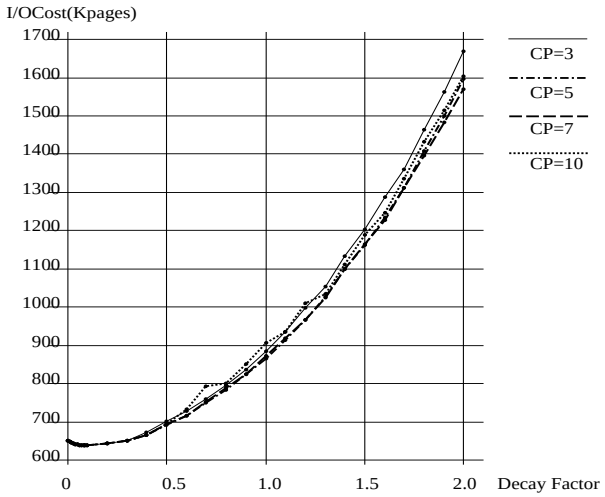

図 7.15 decay factor を変化させた場合のシミュレーション結果

図 7.15は decay factors を変化させた場合の GN ハッシュアルゴリズムの入出力コストを示す。この計測では、リレーションサイズは 400,000 タプル (ステージングバッファの１００倍) とし、 decay factor を 0.0 から 2.0 まで変化させた。図 7.15中の四本の線はそれぞれ評価値を 3,5,7,10 と変化させた場合の結果を示す。 decay factor が 0.0 の場合、入出力コストは均一分布と同じである。図 7.15から分かるように、データが偏ると、入出力コストは大きくなる。 decay factor が 1.0 よりも大きくなると、入出力コストは大きく増加する。これは、溢れた入出力クラスタが何度も読み込まれ、細分かつされる結果、それらのコストが積算してオーバヘッドとなることによる。

前述のとおり、図 7.15からも、データの偏りが小さい場合には、評価値を変化させても GN ハッシュアルゴリズムの入出力コストは影響を受けない。 decay factor が 0.5 よりも大きいと、評価値により入出力コストも変化する。特に、評価式が３の場合、ネストループアルゴリズムが適用されまでに、何度も溢れた入出力クラスタが分割されるため、データの偏りが大きくなるほど、入出力コストが増加する。一方、評価値が１０の場合、溢れた入出力クラスタはさほど再分割されないため、データの偏りが大きくなるとその性能は他の評価値の場合もよりも良くなる。しかしながら、ほとんどの場合、均一分布を基にして算出された評価値５の場合が、他の評価値の結果よりも良い性能であることが確認できる。

7.6.5 選択率を変化させた場合

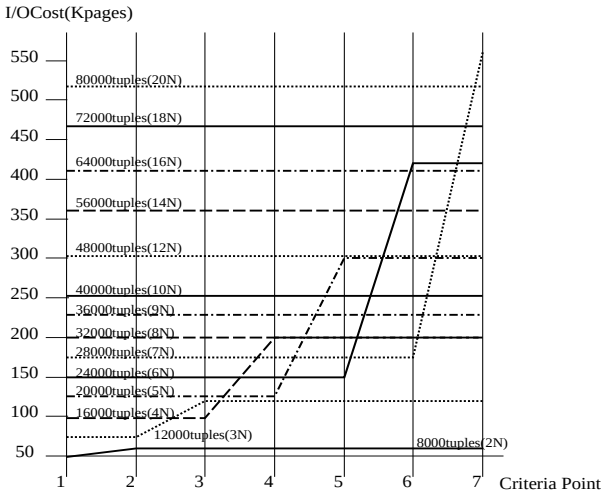

図 7.16 シミュレーション結果

フィルタリング・ファクタが 0.1

上述のシミュレーションで、フィルタリングファクタは 1.0 に固定、二つのリレーションサイズは等しいと仮定してきた。しかしながら、節 7.5の eq(7.12) から分かるように、フィルタリングファクタが減少すると、評価値も変わる。この節では、フィルタリングファクタが変化した倍の eq(7.12) に基づく評価式について検討する。ここで、二つのリレーションのフィルタリング・ファクタおよびサイズは等しいものと仮定する。

図 7.16と図 7.17は、 Zipf 分布 (decay factor が 1.0) のとき、フィルタリング・ファクタを 0.1 と 0.5 にした結果をそれぞれ示す。最初のランタイム方式を選ぶために、これらの図では評価値は１から７まで変化させている。溢れた入出力クラスタに関しては、フィルタリング・ファクタは必ず 1.0 となるため、評価式は５に固定している。フィルタリングされた結果のサイズをそれぞれの線上に示している。フィルタリングファクタが 0,1 の場合、均一分布の評価式は eq(7.12) より１である。これは、 GN ハッシュアルゴリズムの実行時方式として、常に Grace ハッシュアルゴリズムが選択されることを意味している。フィルタリング・ファクタが 0.5 の場合、評価式は３となる。これらの図か

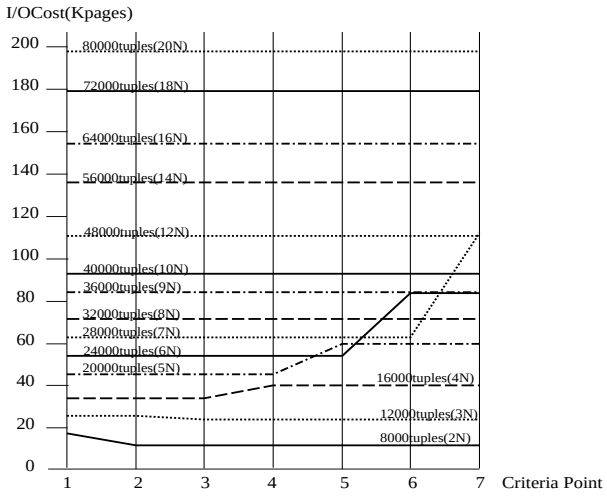

図 7.17 シミュレーション結果

フィルタリング・ファクタが 0.5

ら、最適な評価式は均一分布の評価式となることが分かる。この結果、フィルタリング・ファクタが１でない場合も、我々のコスト式が実行時方式の選択に有効であることを示している。

7.6.6 二つのリレーションサイズが異なる場合の評価結果

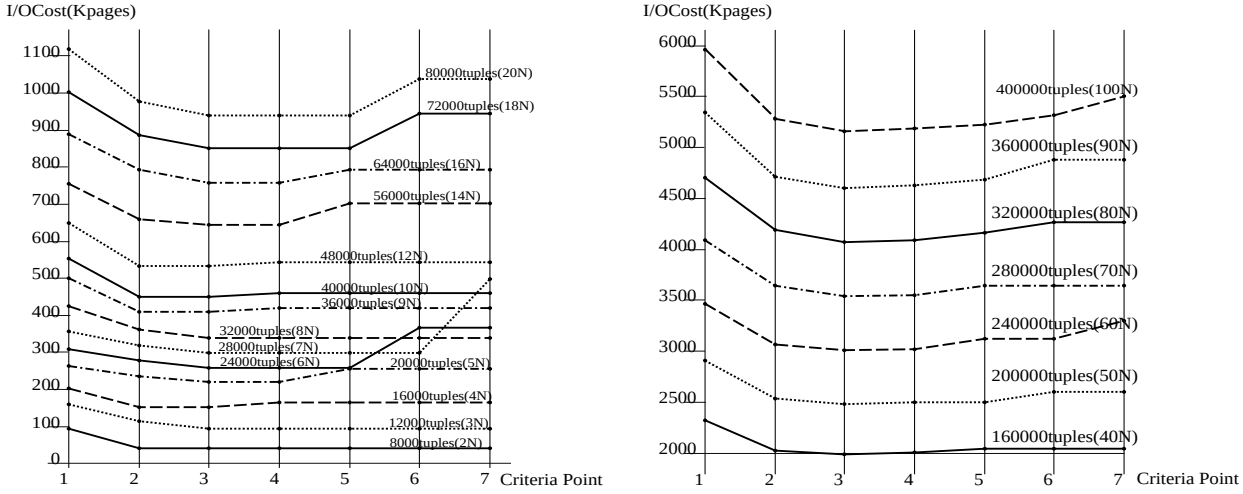

図 7.18 シミュレーション結果 (リレーションサイズが S： R が 10：1 の場合

この節では、リレーション R と S のサイズが異なる場合の評価式について検討する。ここで、節 7.5の eq(7.11) を評価式を算出するのに用いる。

図 7.18はリレーション S がリレーション R より１０倍大きい場合の GN ハッシュアルゴリズムの入出力コストについて示す。シミュレーションは Zipf 分布とし、フィルタリング・ファクタは 1.0 と

する。リレーション R のサイズを図中の線に示している。この場合、 eq(7.11) から算出される評価値は 3 である。この図から、最適な評価式は均一分布の評価式と同じことが確認できた。ここでは一つの例しか提示していないが、リレーション R と S の比が異なっている場合、その線の形状はこの図とほぼ等しい。なぜなら、リレーション S が溢れた入出力クラスタのために呼び込まれる回数はリレーション R のデータ分布で決定され、リレーション S のサイズが変わったとしても、リレーション R のデータ分布は固定されているためである。

この結果から、評価式を算出する eq(7.11) はリレーション R と S のサイズが異なっていた場合でも、 GN ハッシュアルゴリズムの実行時方式を選ぶ上で有効であることが分かる。

7.7 本章のまとめ

従来からのハッシュ結合方式に関して、入出力コストによる性能比較と各方式における問題点の考察を行ない、ハイブリッドハッシュ結合方式などで提案されている主記憶の有効利用による性能の改善が大容量リレーションの処理における入出力コストの大きさに対しわずかであり、また、データの分布が偏っている場合には十分対応できないことを示した。これらの問題を解決するために大容量リレーションに対するハッシュ結合方式として GN ハッシュ結合方式を提案した。本方式では，実行時に入出力コストを比較することでネストループハッシュ結合方式と Grace ハッシュ結合方式の 2 つから処理方式を決定することにより、データの分布の偏りによりオーバフロー入出力クラスタが生成される場合、従来のハッシュ結合処理方式に比べ、性能を改善することが可能であることを示した。また， GN ハッシュ結合方式で用いらる入出力コストの算出式及び評価式の有効性については、実際に機能ディスクシステム [134] 上に GN ハッシュ結合方式を実装し、詳細に検討、確認している [57, 58]。さらに GN ハッシュ結合方式と従来からのハッシュ結合方式との入出力コストによる性能比較を行ない、本方式が入出力クラスタが主記憶入るように分割できる最も理想的な場合に、従来最も性能が良いハイブリッドハッシュ結合方式と比較してほぼ同等の性能であることを示した。また、不均一なデータ分布として Zipf-like 頻度分布を用いた性能比較を行ない、主記憶の１００倍の大容量リレーションにおける結合処理では従来のハッシュ結合方式と比較して入出力コストの増加を押えられ、GN ハッシュ結合方式が最も高い性能を示すことを確認した。

続いて, 偏ったデータ分布として Zipf 分布のデータを用いて、 GN ハッシュアルゴリズムの選択評価条件に関して詳細な性能評価を行った。その結果、 GN ハッシュアルゴリズムが均一分布において得られた評価条件を用いることで偏ったデータ分布においても十分な性能が得られることを確認した。

ハッシュに基づくアルゴリズムは、結合演算に限らず、集約演算、射影演算 (重複除去) 等、他の関係演算にも容易く適応可能であり、また、並列化も容易である。したがって、 GN ハッシュアルゴリズムが結合演算処理以外の関係演算においても、十分に良い性能を得られることが期待できる。

第 8 章

結論

8.1 本研究の成果

関係データベースシステムの高性能化、高機能化を目指した並列処理化について、異なる並列プラットフォーム、共有メモリ環境、分散メモリ環境、分散共有メモリ環境を対象とし、プラットフォームとなるアーキテクチャの特性を考慮し、計算機資源を有効に利用することで、最も効果的なアルゴリズム、処理方式を提案、実際に実機上に実装することでその有効性を示した。また、異なるプラットフォーム上での処理の並列化の開発と同時に、並列処理が容易であり、データの偏りにも性能劣化の少ない新たの演算処理手法として、 GN ハッシュ方式を提案し、その有効性を示した。

機能ディスクシステム（Functional Disk System with Relational database engine : FDS-R）は，プロセッサ本体と二次記憶システム間のＩ／Ｏボトルネックの問題に着目し，二次記憶システムを単なる記憶媒体ではなく，それ自身がデータに対して高レベルな処理機能を持つことにより，関係データベース・システムとしての二次記憶の性能向上を図ったものである。特に，大容量のステージング・バッファと複数台のプロセッサを導入することで，大規模データ処理を効率良くかつ高速に行っている。また，関係代数演算を支援する専用ディスク・コントローラを新しく開発し，専用ハードウェアによる ” 機能 ” も実装している。すでにＦＤＳ－Ｒ第１版の試作機を開発し，基本性能についての計測を行い，一般の商用データベース・システムと比較して高い性能を得られることを確認した。また，ＦＤＳ－Ｒ上におけるデータベース・システムのプロトタイプとして， QUEL Subset System の実装を行った。システムに適合した入出力ドライバ、入出力ライブラリおよび共有メモリ管理機構ライブラリを既存の OS9 をベースとして新たに構築、試作機上に実装した。さらに、このシステム上において、 QUEL をベースとする関係データベース問合せシステムおよび「機能ディスクシステム」のハッシュ機構と利用した並列問合せ処理システムを構築し、開発したハッシュ機構が実用に耐えるものであることを示した [57]。続いて第１版で得られた知見をもとに新たにＦＤＳ－Ｒ第２版を開発し，大規模リレーションに対する関係代数演算の処理方式の考察を行い，本システムに適合した方式の実装を行た。第１版におけるデータ処理は，ＩＤＣによるデータのフィルタリング後の処理結果はステージング・バッファに収まるものとして実現されていた。これに対し，第２版 [134, 59] では，大容量データの処理方式として，さらにＩＤＣに対する制御とステージング・バッファに対する管理方式を追加することにより，ステージング・バッファから溢れるデータに対する処理をサポートした。

さらに、機能ディスクシステムにおける関係代数演算処理方式として本システムに最適化したＧＮハッシュ方式を実装した。本方式では，実行時に入出力コストを比較することで処理方式をハッシュを用いたネストループ方式と Grace ハッシュ方式のいずれかに決定する。機能ディスクシステムでは，汎用 OS を用いた場合に通常データベースシステムに生じるオーバヘッドコストを大幅に削減することで関係代数演算の処理コストは入出力バウンドとなっており，ＧＮハッシュ方式を有効に実装することが可能である。 GN ハッシュ方式で用いられている入出力コストの算出式及び評価式について検討を行い，試作機上での計測結果から本方式におけるアルゴリズム選択評価方式が有効であることを示した。また、メモリのサイズに係わらず、 Grace ハッシュ方式の処理時間がほぼ一定である

ことから、機能ディスクシステムが大規模リレーションの処理において有効であることが確認できた[59]。

1984 年に SEQUENT B2100 が発表され、続いて Multimax, Symmetry 等の CPU が数十台規模の共有メモリ並列計算機が 1990 年前半に商用化された。一方、関係データベース処理はメインフレーム上にてビジネス中心に広く利用されるにつれ、処理の高速化を強く求められるようになり、関係データベースシステムの新しいプラットフォームとして共有メモリ計算機が着目された。そこで、共有メモリ並列計算機上における並列処理による関係データベース処理の高速化の可能性を明確化することを目的とし、最も負荷の重い結合演算を対象に、商用マルチプロセッサであるシンメトリＳ８１に GRACE ハッシュ結合演算技法を実装すると共に評価を行ない、ほぼ理想的な並列処理効果を確認することができた。すなわち、ハッシュ結合演算技法の並列化に関し考察し、共有メモリマルチプロセッサに適したプロセスモデルを明らかにするとともにシンメトリＳ８１上に実装し、スプリットフェイズ、ジョインフェイズ各々に関し並列度を評価し、ディスクからの入出力に追従した処理が可能であることを示した。さらに、ディスク１台当たりに対しプロセッサを２台ずつ増加させることにより、システムを理想的にスケールアップ可能なことを明らかにし、共有メモリマルチプロセッサにより関係データベース処理を効率良く並列処理できることを示した。特にデータベース処理は入出力負荷が大きく、駆動ディスク台数を考慮に入れた並列処理効果の評価が不可欠である。しかしながら、多数の CPU を搭載した共有メモリ計算機が商用化された時点で、このような点に関する十分な性能評価は報告されていなかった。今回のディスク８台、プロセッサ１６台からなる比較的大規模な商用マシンの上での GRACE ハッシュ結合演算技法の実装とその性能評価結果により、関係データベースシステムのプラットフォームとしての共有メモリ並列計算機の可能性を示すことがいち早くできた [62, 133]。

1994 年に開始されたカルフォルニア大学バークレー校の NOW プロジェクトに代表されるように、スケーラビリティとコストパフォーマンスの観点から、共有メモリ型計算機から高性能な WS を高速ネットワークスイッチで結合した分散メモリ型計算機が開発され、 IBM SP ２等の分散メモリ型並列計算機が商用化されるようになった。並列関係データベースシステムの高速化は主として共有メモリ上での技法が論じられきたが、新たに分散メモリ環境における並列化が求められるようになった。大量のデータがネットワーク上を転送されるデータベース処理において、ネットワーク処理コストは処理性能に大きな影響を与える。分散メモリ環境における多重結合演算処理について考察し、 CPU 処理コスト、ネットワーク転送、主記憶サイズ、入出力転送レートなどのシステム資源が与えられた場合に、効率良く多重結合演算のスケジューリングを決定する方式について提案した。近年の分散メモリシステムにおける大容量データベースシステム処理の観点から、多重結合演算のパイプライン処理において、大容量のデータをスムースにネットワーク転送することが重要な鍵となる。我々のアルゴリズムでは、まず、このパイプライン処理においてネットワークバンド幅を十分に利用したバランスシード木を生成する。続いて、生成されたバランスシード木の集合から互いを組み合わせることで、最適な問合せ木の生成を行う。分散メモリシステムにおけるパイプライン処理による結合演算のコスト式を導入し、それを基にシミュレーションにより提案した多重結合演算スケジューリング手法と従

来からの手法の比較を行った。評価結果から、我々の BST 手法が従来の Left Deep 木、 Right Deep 木、セグメント right deep 木などと比較し、生成される結果木の質が良いだけでなく、劣化する率も比較的低いことを示した。特にネットワークバンド幅が限られている場合には、我々のアルゴリズムはネットワークコストを考慮しない従来の手法より優れている。さらに、ネットワークバンド幅に制限がない場合でも、最適解の生成率は従来の手法よりも良く、また、性能劣化は少なく、生成される結果の質が高いことが確認できた。我々の手法は最適解を多く得ることができる一方、探索空間も全空間探索手法と比較しても十分に現実的であることを示した [139]。

大規模共有メモリを実現する新たな並列計算機アーキテクチャとして、 1994 年に KSR1, Convex 等の分散共有メモリ型計算機が商用化された。分散共有メモリ計算機上の並列データベース処理の実装方式の検討を行うため，並列ハッシュ結合演算を取り上げ，分散共有メモリアーキテクチャに適合したバッファ管理方式を提案し，実際に商用分散共有メモリ計算機 (HP Exemplar SPP 1600) 上に実装することで，我々の提案したバッファ管理方式の有効性を確認した．データベース処理は本質的に広大なアドレス空間をランダムに検索する必要があるため，分散共有メモリ計算機を用いた場合，単純な実装では共有メモリへのランダムなアクセスによりメモリ一貫性処理負荷が高くなり，処理性能が下がるという問題点がある．本研究では，分散共有メモリ計算機上での並列ハッシュ結合演算処理モデルを提案し，バッファ領域をハッシュテーブル領域，データバッファ領域と区分し，参照，書き込みのアクセス特性を考慮することで， SE， SHT， LHT， LHT-R 方式の 4 方式を提案し，これらの方式を SPP 1600 上に実装し，性能評価を行った．その結果， CPU 処理コストに関しては，単純な SE 方式の結果と比較して，最も性能のよい LHT-R 方式では， 50% 以上の性能向上が得られることを確認した．また，入出力コストも含めた処理時間の結果を示し，現在のディスク転送速度を前提とした場合，分散共有メモリ計算機上で性能向上を図るためには，分散共有メモリアーキテクチャに適合したメモリ管理方式を用い，メモリアクセスコストを低減する必要があることを示した．続いて、分散共有メモリ計算機のデータベース処理に対するスケーラビリティ、拡張可能性などを検討するために、シミュレーションを用いて分散共有メモリ計算機の性能に大きな影響を与えるリソース (キャッシュサイズ、ノード数、メモリアクセスコスト比、データスキュー) を変化させて性能評価を行い、分散共有メモリ計算機のスケーラビリティを確認した [86, 140]。

関係データベース演算の中で、リレーション同士のリンクを動的に行う結合演算は問い合わせの記述性を高める反面，ファイル間相互の関連付けが静的なリンクで実現されているネットワークモデル等に比べ、動的に双方のリレーションを検索，比較するため単純な処理方式ではその処理負荷は非常に重くなる。この様な背景から，関係データベースシステムの高速化を目指し、特に結合演算を中心とした効率の良い関係代数演算処理方式の研究が数多く行われている。本研究ではハッシュ結合方式に関して、入出力コストによる性能比較と各方式における問題点の考察を行ない、ハイブリッドハッシュ結合方式などで提案されている主記憶の有効利用による性能の改善が大容量リレーションの処理における入出力コストの大きさに対しわずかであり、また、データの分布が偏っている場合には十分対応できないことを示した。これらの問題を解決するために大容量リレーションに対するハッシュ結合方式として GN ハッシュ結合方式を提案した。本方式では，実行時に入出力コストを比較すること

でネストループハッシュ結合方式と Grace ハッシュ結合方式の２つから処理方式を決定することにより、データの分布の偏りによりオーバフロー入出力クラスタが生成される場合、従来のハッシュ結合処理方式に比べ、性能を改善することが可能であることを示した。また, GN ハッシュ結合方式で用いらる入出力コストの算出式及び評価式の有効性については、実際に機能ディスクシステム上に GN ハッシュ結合方式を実装し、詳細に検討、確認した。さらに GN ハッシュ結合方式と従来からのハッシュ結合方式との入出力コストによる性能比較を行ない、本方式が入出力クラスタが主記憶入るように分割できる最も理想的な場合に、従来最も性能が良いハイブリッドハッシュ結合方式と比較してほぼ同等の性能であることを示した。また、不均一なデータ分布として Zipf-like 頻度分布を用いた性能比較を行ない、主記憶の１００倍の大容量リレーションにおける結合処理では従来のハッシュ結合方式と比較して入出力コストの増加を押えられ、 GN ハッシュ結合方式が最も高い性能を示すことを確認した。続いて, 偏ったデータ分布として Zipf 分布のデータを用いて、 GN ハッシュアルゴリズムの選択評価条件に関して詳細な性能評価を行った。その結果、 GN ハッシュアルゴリズムが均一分布において得られた評価条件を用いることで偏ったデータ分布においても十分な性能が得られることを確認した [138, 85]。

8.2 今後の課題

本研究では、関係データベースの性能向上を主として並列アーキテクチャへの実装の観点から、意志決定支援システムなどに見られるアドホックな問合せ処理を対象として考察してきた。提案した並列処理方式は、そのアーキテクチャが現れた時期の IC 技術、計算機技術を反映してはいるが、計算機資源と速度という面では相対的な部分も多く、十分に現在の並列計算機環境および並列関係データベースシステムの状況においても有用であると考える。

しかしながら、その時点の技術においては、容量、転送速度ともに限界のあったストレージ機構、ネットワークの伝送速度とその普及など、現状の技術に照らし合わせ新たに考察すべきことも多い。2006 年現在、 1988 年に提唱された RAID はディスクの小型化により一匡体で数十 TB の容量をある。また、 RIAD に加え、 SAN, NAS などの新しい大容量ストレージシステム [93] は、個々のディスク容量、ストレージキャッシュの増大、高速なファイバチャネルスイッチによる結合、汎用 CPU が用いられたディスクコントローラ、システムコントローラ等一つの計算機システムと見なされるだけ高機能化している。これらの二次記憶システム上にて、機能ディスクシステムの概念をハードウェアとして実現するのではなく、ソフトウェア、あるいは一つのライブラリとして提供することで、関係データベース演算処理の限られた支援から、関係データベースシステム全体への支援を可能とする機構へ発展させることができる。

同様に、大規模エンタプライズサーバでは、 2006 年の段階で数十 GB とディスク容量に匹敵するだけの共有メモリを搭載することが可能となっている。それに伴い、大容量データを扱う関係データベースの性能ボトルネックも二次記憶システム等の入出力機器からのデータ転送速度と CPU の処理性能のみならず、キャッシュと主記憶のアクセス速度の差ストレージから主記憶へデータ転送コス

トの高さなど新たにボトルネックとなる部分が多く存在する [121]。本研究で得られた共有メモリ計算機の結合演算処理で触れたストレージ性能と CPU 処理、バッファ管理システムのバランス、あるいは分散共有メモリ計算機の実装にて述べたデータ局所性を考慮したアクセス方式など、これらの大規模エンタプライズサーバ上でいかに適用するかは新たな課題である。

謝辞

本研究を進めるにあたっては、多くの方々の御指導、御協力を賜りました。

本研究の指導教官である喜連川優教授には、日頃から研究全般にわたっての御指導を賜り、研究を進めていく上において、職務上の配慮から研究の環境まで幅広く多大なご配慮を頂きました。本研究は助手として喜連川研究室に所属してから今日までの、異なる並列計算機環境における並列データベース処理方式の研究をまとめたものであり、これほど長期間に渡って研究を継続、遂行することが可能となりましたのも、喜連川教授の温かい御助言とお励ましに寄るものです。

本研究は喜連川研究室に所属された多くの職員、学生の皆さんの協力なくしては成り立たないものでした。新米の助手として着任してすぐ、中山雅哉氏、楊維康氏、原田リリアン氏、鈴木孝氏には、研究を進める上での様々な面において温かい支援をいただきました。機能ディスクシステムの研究においては、ハードウェアの開発を手掛けていたいただいたアムスク株式会社の辰野氏に様々なアドバイスをいただきました。また、デバッグ、ソフトウェアの開発環境を整えるにあたって、平野聡氏、中村稔氏にお手伝いいただきました。共有メモリ計算機の研究では、津高新一郎氏に実装、実験をお手伝いいただきました。分散共有メモリ計算機における並列結合演算処理の研究では今井洋臣氏に実装、実験のお手伝いをいただきました。助手の根本利弘氏には、多くの研究室全般に渡る職務に関し、多大なるご協力をいただきました。

最後に、本研究成果をまとめるまでの長い時間を、常に温かく見守り、優しく励まし、忍耐と寛容をもって支えてくれました夫、哲夫と娘、美央、母 石原百合恵に心から感謝します。

参考文献

[1] N.M.Aboulenein, S.Gjessing, J. R.Goodman and P.J.Woest: *Hardware Support for Synchronization in the Scalable Coherent Interface(SCI).*

[2] C.Amza, A.L.Cox, S.Dwarkadas, P.Kelher, H.Lu, R.Rajamony, W.Yu and W.Zwaenpoel: *TreadMarks:Shared memory Computing on Networks of Wrokstations*, IEEE Computer, Vol.29, No.2, pp.18-28,1996

[3] Astrahan,M.M., Blasgen,M.W., Chamberlin,D.D., Eswaran,K.P., Gray,J.N., Griffiths,P.P., King,W.F., Lorie,R.A., McJones,P.R., Mehl,J.W., Putzolu,G.R., Traiger,I.L., Wade,B.W. and Watoson, V. : *System R: Relational Approach to Database Management*, ACM Trans. Database Syst., Vol.1 , No.3, pp.189-222(1976)

[4] Bergsten, B., Couprie, M., and Valduriez, P.: *Prototyping DBS3, a shared-memory parallel database sytem*, Proc. of PDIS(1991)

[5] Berra,P.B. and Oliver,E.: *The Role of Associative Array Processor in Data Base Machine Architecture*, IEEE COMPUTER, Vol.12, No.3 (1979)

[6] Bitton,D. et al.: *Benchmarking Database Systems a Systematic Approach*, Proc. of VLDB (1984)

[7] Blasgen,M.W. and Eswaran, K.P.: *Storage and Access in Relational Database*, IBM Systems Journal, Vol.6, No.4, pp.363-377(1977)

[8] Haran Boral: *Parallelism in Bubba*, Proc. of DPDS"88, pp.68-71(1988)

[9] Haran Boral: *Parallelism in Bubba*, Proc. of DPDS"88, pp.68-71(1988)

[10] L.Bouganim, D.Florescu and P.Valduriez: *Dynamic Load Balancing in Hierarchical Parallel Database Systems*, Proc. of 96 VLDB, pp.436-447(1996)

[11] L.Bouganim, B.Dageville and P.Valduriez: *Adaptive Parallel Query Execution in DBS3*, Proc. of EDBT'96, pp.481-484(1996)

[12] Bratbergsengen,K.: *Hashing Method and Relational Algebra Operations*, Proc. 10th VLDB, pp.323-333(1984)

[13] Bubb.E.:" *Implementation of a Relational Databse by Means of Specialiazed Hardware*", ACM Trans. on Database Systems, Vol.4, No.1, pp.1-29(1979)

[14] J.B.Carter, J.K.Bennet and W.Zwaenepoel: *Implementation and Performance of Munin*, Proceedings of the 13th ACM Symposium on Operating Systems Principles ACM,New York,pp152-164. 1991

[15] J.Carter, J.K.Bennet and W.Zwaenepoel: *Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems*, ACM Transaction on Computer Systems Vol.13 No.3,pp205-243, 1995

[16] Chakravarthy, S.: *Divide and Conquer : A Basis for Augmenting a Conventional Query Optimzer with Multiple Query Processing Capabilities* Proc. of DE, pp.482-490 (1991)

[17] J.Chapin, S.A.Herrod, M. Rosenblum,and A. Bupta: *Memory System Performance of UNIX on CC-NUMA Multiprocessors*, SIGMETRICS'95, 1995

[18] Chen, M.S., Yu, P.S. and Wu, K.L.: *Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries* Proc. of DE, pp.58-67 (1992)

[19] Chen, M., Lo, M., Yu, P.S., and Young, H.C.: *Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins* Proc. of VLDB, pp.15-26 (1992)

[20] Christodoulakis,S.: *Implementation of Certain Assumptions in Database Performance Evaluation*, ACM Trans.Database Syst., pp.163-186, Vol.9, No.2 (1984)

[21] Chou,H.T. et al. :" *An Evaluation of Buffer Management Strategies for Relational Database Systems*", Proc. of VLDB, pp.127-141(1985)

[22] CONVEX: *Exemplar SPP1000/SPP1200 Architecture*, Order Number DHW-014,Document Number 081-023430-002.

[23] George P. Copeland, William Alexander, Ellen E. Boughter, Tom W. Keller: *Data Placement In Bubba*, Proc. of SIGMOD"88 pp.99-108 (1988)

[24] B.Dageville, P.Casadessus, P.Boral-Salamet: *The Impact of the KSR1 AllCache Architecture on the Behaivor of the DBS3 Parallel DBMS*, Proc. of Parallel Architectures and Language, 1994

[25] DeWitt,D.J. and Gerber,R. : *Multiprocessor hash-based join algorithms*, Proc of 11th VLDB, pp.151-164(1985)

[26] DeWitt,D, Katz,R., Olken,F., Shapiro,L.D., Stonebraker,M.R. and Wood,D. :" *Implementation Techniques for Main Memory Database*", Proc of SIGMOD ,pp.1-8(1984)

[27] DeWitt,D.J. et al. : *GAMMA - A High Performance Dataflow Database Machine* , Proc. of VLDB, pp.43-59 (1986)

[28] DeWitt,D.J. : *A Single User Evaluation of The GAMMA Database Machine*, Proc. of IWDM, pp.43-59 (1987)

[29] DeWitt,D.J.: *A Performance Analysis of the gamma database machine*, Proc. of SIGMOD"88, pp.350-360 (1988)

[30] DeWitt,D.J., and Gray,J. : *Parallel Database Systems: The Future of High Performance Database Systems*, ACM Communications, Vol.35, No.6, pp.85-98(1992)

[31] DeWitt, D.J., Naughton, J.F., Schneider, D.A. and Seshdari, S.: *Practical Skew Handling in Pralle Joins*, Proc. of VLDB, pp.27-40(1992)

[32] DeWitt, D.J., Naughton, J.F. and Schneider, D.A.: *Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting*, Proc. of PDIS, pp.280-291(1991)

[33] Faloutsos, C., Ng, R. and Sellis, T. *Predictive Load Control for Flexible Buffer Allocation*, Proc. of VLDB, pp.265-274 (1991)

[34] Graefe, G. : *Query Evaluation Techniques for Large Databases*, ACM Computing Surveys, Vol.25, No.2, pp.73-170(1993)

[35] Graefe, G.: *Encapsulation of Parallelism in the Volcatno Query Processing System*, Proc. of SIGMOD, pp.102-111(1990)

[36] Graefe, G. and McKenna, W.J.: *The Volcano Optinizer Generator: Extensibility and Efficient Search*, Proc. of DE, pp.209-218(1993)

[37] Gray J.(ed.) : *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann Publishers(1991)

[38] Haas, P.J. and Swami, A.N.: *Sequential Sampling Processor forQuery Size Estimation* Proc. of SIGMOD, pp.341-350 (1992)

[39] Hagman,R.B.:" *Performance Analysis of Several Back-End Database Architectures*", ACM Trans. on Database Systems, Vol.11, No.1, pp.1-26(1986)

[40] Hirano, Y., Hoshino, T. and Inoue, U.: *Load Design and Implementation of Parallel Database Processing on a Shared Memory Multiprocessor System*, Proc. of International Workshop on Future Database 92, pp.337-346 (1992)

[41] Hirano, Y., Satoh, T., Inoue, U. and Teranaka, K.: *Load Balancing Algorithm for Parallel Database Processing on Shared Memory Multiprocessor*, Proc. of PDIS, pp.210-217 (1991)

[42] Hong, W. : *Exploiting Inter-Operation Parallelism in XPRS*, Proc. of SIGMOD, pp.19-28(1992)

[43] Hong, W. : *Optimization of Parallel Query Execution Plan in XPRS*, Proc. of PDIS, pp.218-225(1991)

[44] Hong, W. : *Optimization of Parallel Query Execution Plan in XPRS*, Journal of Parallel and Distributed Database Systems, pp.9-32(1993)

[45] Hua, K.A. and Lee, C.: *An adaptive data placement scheme for parallel database computer systems*, Proc. of VLDB, pp.493-506 (1990)

[46] Hua, K.A. and Lee, C.: *Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning*, Proc. of VLDB, pp.525-535 (1991)

[47] Hua, K.A. and Lee, C.: *Interconnecting Shared-Everything Systems for Efficient Parallel Query Processing*, Proc. of DE, pp.262-270 (1991)

[48] Hua, K.A., Lo, Y.L. and Young, H.C.: *Including The Load Balancing Issue in The Optimization of Multi-Way Join Queries for Shared-Nothing Database Computers*, Proc. of DE, pp.74-83 (1993)

[49] Hua, K.A., Lo, Y.L. and Young, H.C.: *Considering Data Skew Factor in Multi-Way Join Query Optimization for Parallel Execution*, VLDB Journal, Vol.2, pp.303-330 (1993)

[50] Ushio Inoue, Haruo Hayami, Hideki Fukuoka, Kenji Suzuki: *RINDA - A Relational Database Processor for Non-Indexed Queries*, Proc. of DASFAA'89 pp.382-386 (1989)

[51] Kai Li & Paul Hudak: *Memory Coherence in Shared Virtual Memory Systems*, ACM Transaction on Computer Systems, Vol.7, No.4, pp.321-359, 1989

[52] Kaplan,J. :" *Buffer Management Policies in a Database Sysmtems*", MS.Th., Univ. of California, Berkley(1980)

[53] Kitsuregawa,M., Nakano,M. and Takagi,M.: *Query Execution for Large Relations on Functional Disk System*, Proc. 6th IEEE Int. Conf. on Data Engineering, pp.159-167 (1989)

[54] Kitsuregawa,M., Nakano,M. Harada,L. and Takagi,M.: *Performance Evaluation of Functional Disk System with Nonuniform Data Distribution*, Proc. 2nd IEEE Int. Sym. on DPDS, pp.80-89 (1990)

[55] Kitsuregawa,M., Nakano,M. and Takagi,M.: *Performance Evaluation of Functional Disk System (FDS–R2)*, Proc. 8th IEEE Int. Conf. on Data Engineering, pp.416-425 (1991)

[56] Kitsuregawa,M., Nakano,M., Hatada,L. and Takagai,M: "*Functional Disk System for Relational Database Engine*", Proc. IEEE Int. Conf. on Data Engineering, pp.88-95 (1987)

[57] Kitsuregawa,M., Nakano,M. and Takagi,M.:" *Query Execution for Large Relations on Functional Disk System*, Proc. IEEE Int. Conf. on Data Engineering,pp.159-167(1989)

[58] Kitsuregawa,M., Nakano,M. and Takagi,M.:" *Performance Evaluation of Functional Disk System with the Nonuniform Data Distriubution*, Proc. IEEE Int. Symp. on Parallel and Distributed Database systems, pp.159-167 (1990)

[59] Kitsuregawa,M., Nakano,M. and Takagi,M.:" *Performance Evaluation of Functional Disk System(FDS-R2)*, Proc. IEEE Int. Conf. on Data Engineering, pp.159-167 (1991)

[60] Kitsuregawa,M., and Ogawa, Y. : *Bucket Spreading Prallel Hash: A New, Robus, Parallel Hash Join Mehod*, Proc. of International Conference on VLDB, pp.210-221 (1990)

[61] Kitsuregawa,M., Tanaka,H. and Moto-oka,T. : *Application of Hash to Data Base Machine and Its Architecture*, New Generation Computing, Vol.1, No.1, pp.62-74(1983)

[62] Kitsuregawa,M., Tsudaka,S., Nakano,M. and Takagi M. : *Parallel Grace Hash Join on Shared-everything Multiprocessor : Implementation and Performance Evaluation on Symmetry S81*, Proc. of DE,pp.256-264(1992)

[63] Kiyoki, Y., Hasegawa, R. and Amamia, M.: *A Stream-Oriented Parallel Processing Scheme for Relational Database Operations*, Proc. of International Coference of Prallel Processing, pp.1013-1020 (1986)

[64] Kiyoki, Y., Kurosawa, T., Kato, K. and Masuda, T.: *The Software Architecture of a Parallel Processing System for Advanced Database Applications*, Proc. of 7th International Conference on Data Engineering, pp,220-229 (1991)

[65] Knuth,D.E. : *The Art of Computer Programming*, Vol.3, Addison-Wesley, Massachusetts(1973)

[66] Kojima,T. et al :" *IDP - A Main Storage Based Vector Database Machine*", Proc. of IWDM(1987)

[67] J.Kushkin, D.Ofelt, M.Heinrich, J.Heinlein, R.Simoni, K.Gharachorloo, J.Chapin, D.Nakahira, J.Baxter, M.Horowitz, A.Gupta, M.Rosenblum and J.Hennesy: *The Stanford FLASH Multiprocessor*, Proc. of Int' Symposium on Computer Architecture'94, pp.302-313, 1994

[68] Lakshmi, M.S. and Yu, P.S.: *Effective of Parallel Joins*, IEEE Trans. on Knowledge and Data Engineering, Vol.2, No.4, pp.410-424 (1990)

[69] Lanzeltte, R.S.G. and Valduriez, P.: *Extending the search starategy in query optimizer*, Proc. of VLDB, pp.363-373(1991)

[70] Lanzelotte, R.S.G., Valduriez, P. and Zait, M.: *Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies*, Proc. of SIGMOD, pp.256-265(1992)

[71] Lanzelotte, R.S.G., Valduriez, P. and Zait, M.: *On the Effectiveness of Optinization Serach Strategies for Parallel Execution Spaces*, Proc. of VLDB, pp.493-504(1993)

[72] Lee, C. and Chang, Z.A.: *Workload Balance and Page Access Scheduling for Parallel Joins in Shared-Nothing Systems*, Proc. of DE, pp.411-418 (1993)

[73] Lo, M.L., Chen, M.S., Ravishankar, C.V. and Yu, P.S.: *On Optiamal Processor Allocation to Support Pipelined Hash Joins*, Proc. of SIGMOD, pp.69-78 (1993)

[74] Lu, H. Tan, K.L. and Shan, M.C: *Hash-based Join Algorithms for Multiprocessor Computers with Shared Memory*, Proc. of VLDB, (1990)

[75] Lu, H. Shan, M.C and Tan, K.L. : *Optimization of Multi-Way Join Queries for Parallel Execution*, Proc. of VLDB, pp.549-560(1991)

[76] Lu, H. and Tan, K.L. : *Dynamic Load-balanced Task-Oriented Database Query Processing in Parallel Systems*, Proc. of EDBT, pp.358-372(1992)

[77] Lu, H. and Tan, K.L. : *Pipelining Multi-way Join Queries in Shared-Nothing Systems*, Submitted Paper for DE(1993)

[78] Lu, H. and Tan, K.: *Batch Query Processing in Shared-Nothing Multiprocessors* Proc. of DASFAA, pp.45-55 (1995)

[79] Matthias A.Blumrich,Kai Li,Richard Alpert,Cezary Dubnicki,and Edward W.Felten: *Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer*, Proceedings of the 21st Annual International Symposium on Computer Architecture, pp.142-153, 1994

[80] Mehta, M., Soloviev, V. and DeWitt, D.J.: *Batch Scheduling in Parallel Database Systems*, Proc. of DE, pp.400-410(1993)

[81] P.Mishira and M.H.Eich : *Join processing in relational databases*, ACM Computing Surveys, Vol.24, No.1, pp.63-113(1992)

[82] Moss, J.: *Getting the Operating SYstem Out of the Way*, IEEE Database Engineering, Vol.9, No.3(1986)

[83] Murphy, M.C. and Shan, M.C.: *Execution Plan Balancing*, Proc. of DE, pp.698-706 (1991)

[84] Nagi M.Aboulenein,Stein Gjessing,James: R.Goodman and Philip J.Woest: *Hardware Support for Synchronization in the Scalable Coherent Interface(SCI).*

[85] Miyuki Nakano, Masaru Kitsuregawa: *Examination of Criteria for Choosing a Run Time Method in GN Hash Join Algorithm*, IEICE Transactions on Information and Systems, Vol.E79-D ・ No.11, pp.1561-1569(1996)

[86] M.Nakano, H.Imai and M.Kitsuregawa : *Performance Analysis of Parallel Hash Join Algorithms on a Distributed Shared Memory* Proc. of DE, pp.76-85(1998)

[87] Nakayama,M., Kitsuregawa,M., and Takagi M. : *The Effect of Bucket Tuning in the Dynamic Hybrid GRACE Hash*, Proc. of 15th VLDB, pp.257-266 (1989)

[88] Niccum, T.M., Srivastava, J., Himatsingka, B. and Li, J.: *A Tree-Decomposition Approach to Parallel Query Optimization*, Submitted for DE94 (1993)

[89] Omiecinski, E.: *Performance Analysys of a Load Balancing Hash-Join Algorithm for a Shared Memory Multiprocessor*, Proc. of VLDB, pp.375-385 (1991)

[90] Ozkarahan,E. et al.: *RAP - Assosiative Processor for Database Management*, AFIPS, Vol.44(1975)

[91] Pang,H., Carey, M.J. and Livny, M.: *Partially Preemptible Hash Joins*, Proc. of SIGMOD, pp.59-68(1993)

[92] S.Pramanik and W.R.Tout : *The NUMA with Clusters of Processors for Parallel Join*, Int. Journal on Knowledge and Data Engineering, Vo.9, No.4, pp.653-660(1998)

[93] W. Curtis Preston : *Using SANs and NAS : Help for Storage Adiministrators*, ISBN 10;0-596-00153-3, ISBN 13:9780596001537, O'REILLY (2002)

[94] Pucheral, P. and Thevenin, J.M.: *Piplined Query Processing in the DBGraph Storage Model*, Proc. of EDBT, pp.516-533(1992)

[95] Tetsuji Satoh, Hideaki Takeda, Ushio Inoue, Hideki Fukuoka: *Acceleration of Join Operations by a Relational Database Processor, RINDA*, Proc. of DASFAA'91 pp.243-248 (1991)

[96] Schneider,J.A. and DeWitt,D.J.: *A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor*, Proc. of SIGMOD, pp.110-121 (1989)

[97] Schneider,J.A.: *Complex Query Processing in Multiprocessor Database Machines*, Computer Science Tech. Rep. No.965, Univ. of Wisconsin (1990)

[98] Schneider, J.A. and DeWitt, D.J.: *Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines*, Proc. of VLDB, pp.469-480(1990)

[99] Selinger, G., et al: *Access Path Selection in a Relational Database Management System*, Proc of SIGMOD, pp.189-222(1979)

[100] Sellis, Timos K.: *Multiple-Query Optinmization* ACM ToDS, Vol.13, No.1, pp.23-52(1988)

[101] Seshadri, S. and Naughton, J.F.: *Sampling Issues in Parallel Database Systems*, Proc. of EDBT, pp.328-343(1992)

[102] A.Shardal and J.F.Naughton: *Using Shared Virtual Memory for Parallel Join Processing*, Proc. of SIGMOD '93, pp.119-128, 1993

[103] Shapiro,L.D. and DeWitt,D.J.: *Join Processing in Database Systems with Large Main Memory*, ACM Trans. Database Syst., Vol.11, No.3, pp.239-264(1986)

[104] Shiekita, E.J., Young, H.C. and Tan, K.L.: *Multi-Join Optimization for Symmetric Multiprocessor* Proc. of VLDB, pp.479-492 (1993)

[105] Solviev, V.: *A Truncating Hash Algorithm for Processing Band-Join Queries*, Proc. of DE, pp.419-427(1993)

[106] Li, J., Srivastava, J. and Rotem, D.: *CMD: A Multidimensional Declustering Method for Parallel Database Systems*, Proc. of VLDB, pp.3-14 (1992)

[107] Srivastava, J. and Elsesser, G.: *Optimizing Multi-Join Queries in Parallel Relational Databases*, Proc. of DE, pp.84-92 (1993)

[108] Stonebraker,M.: *Operating System Support for Data base Management*, CACM, Vol.24, No.7 (1979)

[109] Stonebraker, M. et al. : *The Design of XPRS*, Proc. of VLDB, pp.318-330, (1988)

[110] Stonebraker,M.R., Wong,E., Kreps,P. and Held,G.D. : *The Design and Implementation of INGRES*, ACM Trans. Database Syst., Vol.1, No.2, pp.97-137(1976)

[111] Su,S.Y. and Lipovski,J.: *CASSIM: A Cellular System for Very Large Database*, Proc. of VLDB(1975)

[112] Swami, A.N. and Iyer, B.R.: *A Polynomial Time Algorithm for Optimizing Join Queries* Proc. of DE, pp.345-354 (1993)

[113] Tamura, T., Nakamura, M., Kitusregawa, M. and Ogawa,, Y.: *Implementation and Performance Evaluation of the Parallel Relational Database Server SDC-II*, Proc. of 25th International Conference on Parallel Processing (ICPP"96), I-212 - I-221 (1996)

[114] Tamura, T. and Kitusregawa, M.: *Implementation and Evaluation of the Bucket Flattering Omega Network of the Prallel Relational Database Server SDC-II* Proc. of 5th International Conference on Database Systems for Advanced Applications (DASFAA"97), pp.471-480 (1997)

[115] CORPORATE Tandem Perfomance Group.: *A benchmark of non-stop SQL on the debit credit transaction & Facilities*, Proc. of SIGMODJ"88, pp.337-341 (1988)

[116] Teradata Corp.: *DBC/1012 Data Base Computer Concepts & Facilities*, C02-0001-05

[117] Ubell,M :" *The Intelligent Database Machine(IDM)*", Query Processing in Database Systems, ed. by Won Kim et al,, Springer-Verlag,(1985)

[118] Valduriez,P.: *Parallel Database Systems: Open Problems and New Issues*, Distributed and Parallel Databaes, Vol.1, No.2, pp.137-165(1993)

[119] Valduriez, P.: *Parallel Database Systems: the case for shared-something*, Proc. of DE, pp.460-465(1993)

[120] Walton, C.B., Dale, A.G. and Jenevein, R.M.: *A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins*, Proc. of VLDB, pp.537-548 (1991)

[121] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki and Babak Falasafi: *Temporal Streaming of Shared Memory*, IEEE ISCA'05, pp.222-233(2005)

[122] Wilkinson,W.K., et al.: *KEV - A Kernel for Bubba*, Prof. of IWDM, pp.29-42(1987)

[123] Wolf, J.L., Dias, D.M. and Yu, P.S.: *An effective algorithm for parallelizing sort merge in the presence of data skew*, Proc. of DPDS, (1990)

[124] Wolf, J.L., Dias, D.M., Yu, P.S. and Turek, J.: *An effective algorithm for parallelizing hash joins in the presence of data skew*, Proc. of DE, pp.200-209(1991)

[125] Wolf, J.L., Iyer, B., Pattipati, K. and Turek, J.: *Optimal Buffer partitioning for the nested block join algorithm*, Proc. of DE, pp.510-519(1991)

[126] Wolf, J.L., Dias, D.M., Yu, P.S. and Turek, J.: *Comparative performance of parallel join algorithms*, Proc. of PDIS, pp.78-88(1991)

[127] Wolf, J.L., Dias, D.M., Yu, P.S. and Turek, J.: *A Parallel Sort Merge Join Algorithm for Managing Data Skew*, IEEE Trans. on Parallel and Distributed Systems, Vol.4, No.1, pp.70-86(1993)

[128] Wolf, J.L., Dias, D.M., Yu, P.S. and Turek, J.: *Multi Query Optimization...*, Submitted paper for DE(1993)

[129] Yamane,Y: *Hash Join Method and Relational Algebra Operation*, Proc.of FODO, pp.388-398 (1985)

[130] Ziane, M., Zait, M. and Boral-Salamet, P.: *Parallel Query Processing in DBS3*, Proc. of DE, pp.93-102(1993)

[131] Ziane, M., Zait, M. and Boral-Salamet, P.: *Parallel Query Processing with Zigzag Trees*, Journal of VLDB,Vol.2, No.3, pp.277-302(1993)

[132] 井上潮 他:" データベースプロセッサ *RINDA* のアーキテクチャ", 第３７回情報処理全国大会,5Q-4(1988)

[133] 喜連川優、津高新一朗、中野美由紀: 共有メモリ型マルチプロセッサによる並列ハッシュ結合演算処理とその評価、情報処理学会論文誌、(1992)

[134] 喜連川優、中野美由紀: 機能ディスクシステム – 関係データベースシステムとその評価 –, 電子通信情報学会論文誌, Vol.J74-D-I,No.8,pp.496-507(1991)

[135] 喜連川優, 中山雅哉、高木幹雄: 動的処理バケット選択手法に基づくハッシュ結合処理方式とその性能評価, 情報処理学会論文誌, Vol.30, No.8, pp.1024-1032 (1989)

[136] 中野美由紀, 喜連川優、高木幹雄: 密結合マルチプロセッサに於ける関係代数演算の評価 - 結合演算 -、情報処理学会第 35 回全国大会講演論文集, 4CC-1, 1987

[137] 中野美由紀, 喜連川優: 機能ディスクシステム第 2 版における関係代数演算処理方式とその性能評価, アドバンストデータベースシンポジウム, pp.91-98 (1988)

[138] 中野美由紀、喜連川優: GN ハッシュ結合方式とその性能評価, 情報処理学会論文誌, Vol.35, No.9, pp.1861-1873(1994)

[139] 中野, 新谷, 喜連川: 並列データベースシステムにおける多重結合演算処理の最適化とその評価, 情報処理学会 アーキテクチャ研究会, SWOPP'95 (1995)

[140] 中野美由紀, 今井洋臣、喜連川優: 分散共有メモリ計算機における並列ハッシュ結合演算処理の性能解析, 電子情報通信学会論文誌, Vol.J82-D-I, No.1, pp.82-97 (1999)

[141] 津高新一郎, 中野美由紀, 喜連川優, 高木幹雄: 共有メモリ型マルチプロセッサマシンにおける並列結合演算処理並列／分散／協調処理に関する『大沼』サマー・ワークショップ第４回ＳＷｏＰＰ大沼'91 電子情報通信学会, CPSY91-6, pp.17-24, 1991

付録 A

並列データベースシステムにおける多重問合せ処理最適化技法

A.1 バッチ問合せ処理最適化とは

近年、高性能なマイクロプロセッサの開発、大容量メモリの低価格化、高速ネットワーク、ディスクの低価格化、小型化などを背景に商用並列計算機が多数登場し、多くの関係データベースシステムが並列計算機の上に実装されるようになってきた．並列関係データベースにおける並列処理の研究では関係データベースにおける演算内並列処理, 複数演算間並列処理の研究が負荷の重い多重結合演算を中心に多く行なわれているが, 多重問合せ間並列処理に関しては, いまだ十分な研究が行なわれていない [80]．特に, 単一問合せ内の並列多重結合演算スケジューリングの最適化については多くの研究が行なわれているが, この結果を基にした並列多重問合せの最適化についてはほとんど研究されていない.

[100, 16] において提案されている従来の多重問合せ処理の最適化では, 問合せ言語の述部に着目している。発行される複数の問合せの中から、静的データベースの情報と共に、共通のリレーションを参照するする部分問合せを言語レベルで抜き出し、同時処理を行なっている。本方式では、共有メモリ環境であれば、共有できる部分問合せを共有メモリ上に保持しておくことで、並列処理をすることも可能であるが、分散メモリ環境では、リレーションの配置、大きさおよびいずれのノードにリソースをロードするかなど、実装する上で様々な課題が残っており、提案された方式では並列化は容易ではない。

[80] では, 並列データベースシステム上での多重問合せの最適化としてオペレーションレベルでの最適化を提案し, システム環境パラメタを考慮することで複数問合せの並列処理を実現している．しかしながら, この最適化では近年の単一問合せの最適化で採用されているパイプライン処理を考慮せず、問合せの個々の演算を対象とした最適化を行なっている.

[78] では, 並列計算機上における単一問合せの最適化の結果として right-deep 木が導出された場合を対象として, 多重問合せ処理のスケジューリングを行なっている．本方式で採用している多重結合演算の最適化技法 [19] では一連のパイプライン処理の単位であるセグメントごとの結果をディスクに書き戻しているため, 最近の単一問合せ最適化技法で採用している中間結果を主記憶にのせ高速化を図る多重結合演算パイプライン処理方式に適応することはできない.

本章では、関係データベース処理の中でも負荷の重い多重結合演算を中心に構成されるバッチ問合せ処理の最適スケジューリング方式について非共有型並列計算機上で検討を行なう.

A.2 並列多重問合せ処理の最適化

分散メモリ型並列計算機の多重結合演算からなる個々の問合せに関しては [139, 104] などで提案されている最適化技法によりスケジューリングが決定される. これらの方式ではいずれもネットワーク上のデータ転送と入出力のオーバラップを前提としたパイプライン並列処理を採用している.

図 A.1に多重結合演算処理の木の形状と、その際の処理の単位 (セグメント) がどのように決定されるかについて示す。各セグメントの処理に関しては、すでに第 2 章 2.5.5 節の分散共有メモリ環境における多重結合演算の並列処理の流れ、および第 5 章 5.2 節にて詳細に述べているので、ここでは

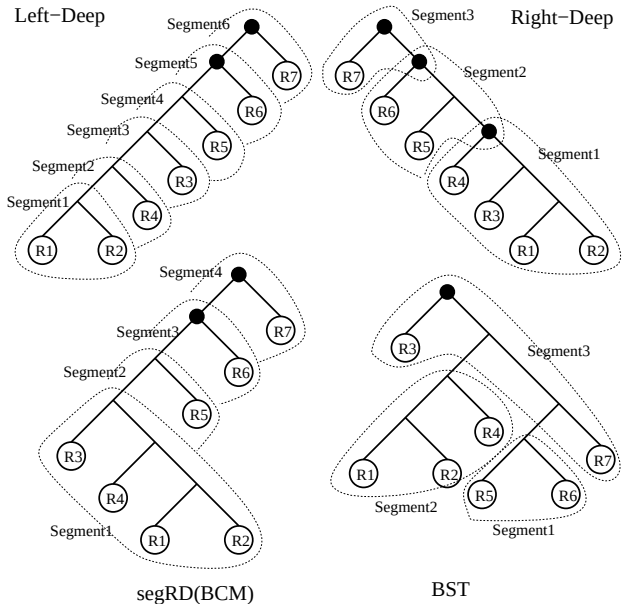

図 A.1: セグメントの切り方

省略する。図中に黒丸で現れされているノードは、中間結果が主記憶に収まらず、ディスクに書き戻されることを示している。

多重結合演算処理の処理単位として、分散メモリ環境の並列化を最も行いやすいセグメント RD 木を基として、多重問合せ処理の最適化を考える。ここでは、多重問合せは、複数の結合演算処理から構成されちえるとする。一般に、複数の問合せ処理は、問合せが発行された順に、問合せ毎に問合せ内のスケジューリングの最適化がなされ、そのプランにしたがって処理される。その際、問合せごとに対象リレーションが異なっていれば、逐次に問合せごとに処理の並列化を行ったとしても、性能に影響はない。しかしながら、近年の意志決定支援システムなどの処理においては、繰り返し参照されるリレーション、複数の問合せにまたがって参照されるリレーションなどが多く見受けられる。また、バッチ処理などで一連のデータベース処理が行われる場合には、複数の問合せでほとんど同じリレーションが利用されると考えられる。

A.2.1 資源共有の効果

ここで、多重問合せにおいて資源の共有を行うことによる、性能向上への効果および、資源共有とし何を考えるべきかについて、簡単な例を基に考察する。ここで、個々の問合せは多重結合演算のみで構成され、多重結合演算のスケジューリング最適化については、すでに、第５章にて論じた方式で行われているものとする。現在の多重結合演算処理では、小さな Right-Deep 木の形状をしたセグメントが、個々の並列処理の対象となる。ここでも、複数の問合せにおけるセグメント同士の処理スケジュールについて考える。

図 A.2に事例となる問合せの結合グラフとスケジューリング木を示す。簡単のために、それぞれ

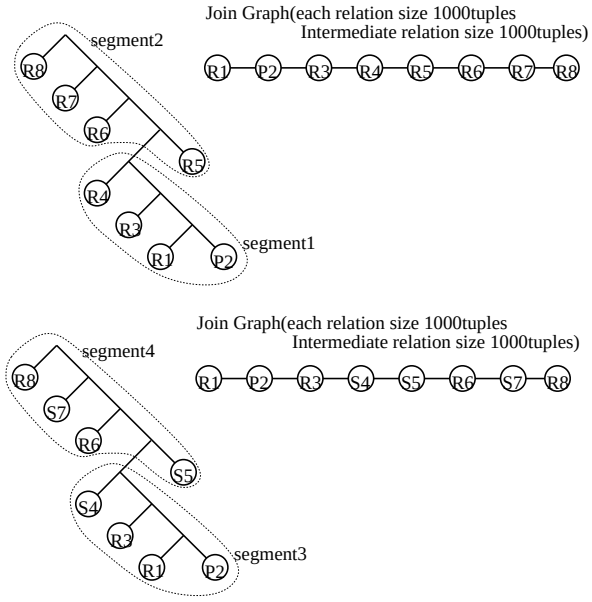

図 A.2: 例；処理木と結合グラフ

のリレーションサイズは同一で１０００タプルとし、主記憶は１つの問合せあたり４０００タプルが割り当てられいるとする。また、ネットワークの転送幅が入出力の転送幅の５倍とする。図から分かるように、２つの問合せをまず与えられた計算機資源に合わせた形でセグメント RD へと分割される。この図では Query1 は segment1 と segment2 に分割、 Query2 は segment3 と segment4 にスケジューリングされている。最も共有部分の多いセグメント同士を同時に処理するよう、実行プランを作成しなくてはならない。

ここで、セグメント間の資源の共有について具体的に検討を行う。図 A.3は図 A.2に示した二つの問合せで共通のリレーションについて色分けしたものである。セグメントが処理の基本単位となるため、単純に処理を行う場合には、 segment1, 2, 3, 4 と番号のとおりに順次処理が行われる。しかしながら、同時に複数のセグメントの処理が可能であるならば、この例では segment1 に対して、 segment3 あるいは segment3 終了後の segment ４が同時実行可能なセグメントとなる。そこで、 segment 間で共通に利用されるリソースにより重みづけをし、複数の問合せのセグメント間での同時実行の組み合わせを求める。

セグメント間で共有することもできるリソースは、直感的には、そのセグメントで読み出すソースリレーションは簡単に共有することができる。図 A.3では、同じ色で記してあるリレーション Ｒ １、Ｐ ２、Ｒ ３、Ｒ ６、Ｒ はなどである。また、セグメント RD のそれぞれの結合演算処理をハッシュ結合方式で行うことを前提とすると、結合属性が同じ場合には、主記憶上に作成されたハッシュテーブルを共有することができる。同様に、セグメントのプローブフェーズにおいては、図中のＰ ２あるいは中間結果のＴ １、Ｔ ２などのリレーションを共有することができる。

同時実行するセグメントを選ぶための重み付けは、共有することのできる資源のコストの大きさ

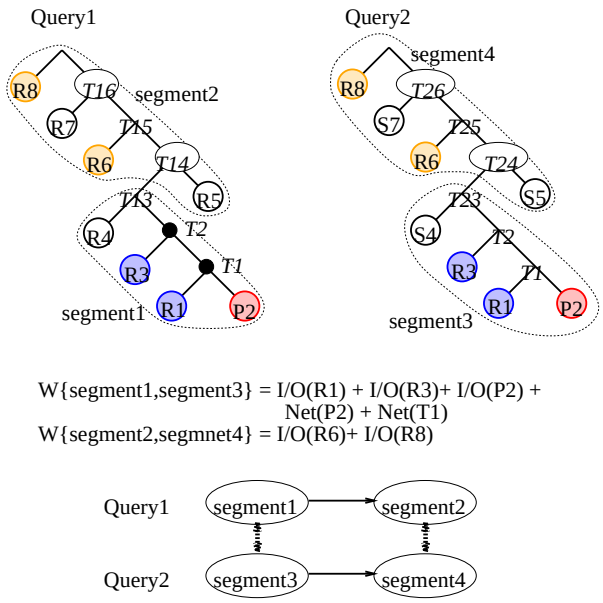

図 A.3: 複数セグメント間の共有度合と処理順序

とする。例えば、図 A.3では、 segment1 と segment3 は共有リソースがあるため、重み付けの式は図中の真ん中に表されたようなものになる。同様に segment1 と segment4 は共有リソースがないため、同時実行の対象とならない。ついで、 segment2 の場合、 segment3 との間には共有リソースがないため、重み付けは 0 となる。一方、 segment2 と segment4 はソースリレーション R 6 と R 8 が共有できるため、そのコストが重み付けとなる。このようにして、計算された重み付けで、問合せ内の各セグメントとのグラフを作成し、セグメント単位での実行順序を決定する。この例では、 segment1 とよび segment3 がそれぞれの問合せの先頭のセグメントであり、また、双方の間の重み付けの値が正であるため、資源の共有と共に同時実行可能となる。同様に segment2 と segment4 も資源の共有実行可能と見なされる。

Query1 の segment1 と Query2 の segment3 の同時実行処理の流れを図 A.4に示す。図中の上部にスケジューリング木を、下部にハッシュテーブルとそれを走査するパイプライン処理の矢印で表す。図 A.4では、 R1 と P2 の結合結合属性、 R1 と P2 の結合演算の結果と R3 との結合属性は同じ属性を使うものとする。この場合、下部の図で着色された部分のハッシュテーブルは、双方の問合せ共、共有できると共に、結合後の中間結果もそのまま利用可能となり R 3 までの結合演算処理は問合せごとに行う必要はない。したがって、 segment1 と segment3 の処理の流れは、次のとおりとなる。まず R1,R2,R4,S4 のハッシュテーブルが主記憶上に生成される。続いて、リレーション P2 が読み出され、 R1、 R2 のハッシュテーブルが走査され、その結果が segment1 の R4 のハッシュテーブルを探査するプローブプロセスと segment2 の S4 のハッシュテーブルを探査するプローブプロセスへと引き渡される。それぞれのプローブプロセスはその結果をそれぞれ、 T13 および T23 のハッシュテーブルを生成するビルドプロセスへと送られる。 segment1 と segment3 の同時実行処理が終わると、

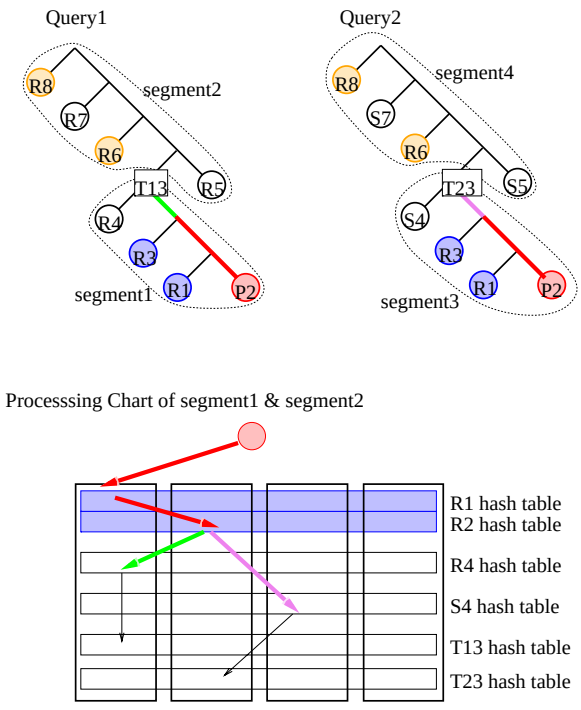

図 A.4: 資源共有時の実行の流れ（1）

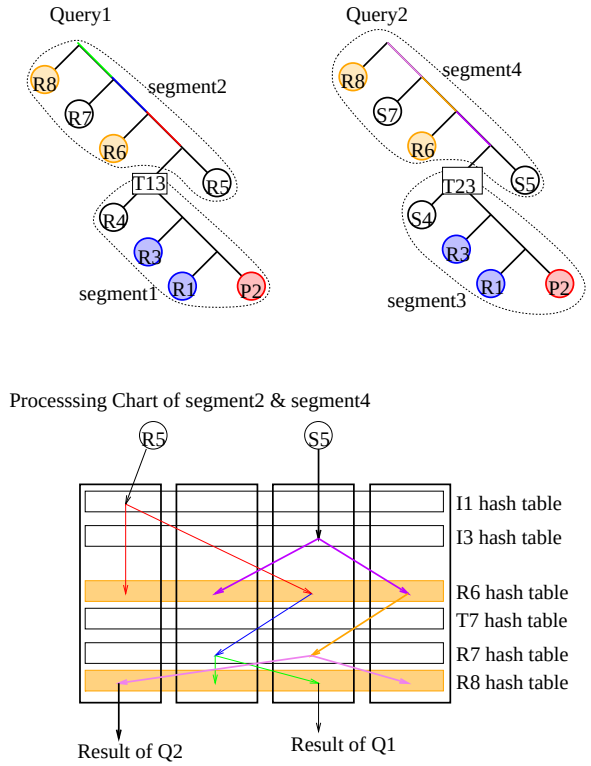

図 A.5: 資源共有時の実行の流れ（2）

主記憶上に T13 と T23 のハッシュテーブルのみが保持される。

Query1 の segment2 と Query2 の segment4 の同時実行処理の流れを図 A.5に示す。図中の上部にスケジューリング木を、下部にハッシュテーブルとそれを走査するパイプライン処理の矢印で表す。図 A.5では、 R6 と P7 のソースリレーションの読み出しのみが共通である。この場合、下部の図で着色された部分のハッシュテーブルは、双方の問合せ共、共有に利用できる。したがって、 segment2 と segment4 の処理の流れは、次のとおりとなる。保持されている T13,T23 のハッシュテーブルと共にまず R6,R7,S7,R8 のハッシュテーブルが主記憶上に生成される。続いて、それぞれのセグメントごとにパイプラン的にプローブ処理を行うために、該当のソースリレーション R5 と S5 が読み出され、主記憶上のハッシュテーブルが走査される。この場合、結合属性が一致した結合演算はすでに存在していないため、個別に主記憶上のハッシュテーブルの走査が行われ、結果がディスクへと書き戻される。

この二つの問合せにおいて、セグメントにおける資源を共有したことにより、どれだけ処理コストが低減するかについて、簡単なコスト値を用いて表 A.1に示す。ここで、いずれのリレーションおよび中間結果も 1000 とし、第 5 章の議論と同様に、 CPU 処理コストは入出力およびネットワーク転送コストとオーバラップできるものとする。また、ネットワーク転送幅は入出力転送幅の 5 倍と仮定する。表の上部が Query1 と Query2 を個別に処理したときのコストである。一方、表の下部の Q1+Q2 の部分が segment の資源共有を行った場合のコストである。具体例からも分かるように、共有される

Query #	segment#	Build		Probe		Total
		io cost	net cost	io cost	net cost	
Query1	segment1	3000	600	1000	800	4000
	segment2	3000	600	2000	800	5000
	total					9000
Query2	segment3	3000	600	1000	800	4000
	segment4	3000	600	2000	800	5000
	total					9000
total						18000
Q1+Q2	seg1+3	4000	800	1000	1200	5200
	seg2	3000	600	2000	1400	5000
	seg4	1000	200	2000	1400	3000
total						13200

表 A.1: 資源共有時の効果に関するコスト

資源の入出力およびハッシュテーブルの生成コストを削減することで、およそ 25% コストが小さくなっている。これは、処理に長時間かかる意志決定支援システムの問合せなどでは十分に大きな利得と考えられる。

A.2.2 セグメントベースバッチ最適化方式

我々はすでに最近の単一問合せ最適化で得られたセグメントから構成される問合せ木を基に多重結合演算処理の最適スケジューリング方式として、生成されるパイプラインを最大限利用することによりシステム全体資源利用を効率的に行うセグメントをシード (BST: Balanced Seed Tree) とする方式を提案している。この方式において提案した分散メモリ環境における並列結合演算処理のコスト式を用い、新たに多重問合せ処理における処理プランの最適化方式を提案する。

この方式では、それぞれの単一の問合せ処理には BST 方式あるいは segmented RD 方式など適用し、あらかじめ、問合せごとの最適化スケジューリングはなされているものとする。結果として得られたセグメントを基にして、さきほどの例で示したように各セグメントの共有資源による重み付けグラフを作成し、それに従ってセグメントの実行順序を定めて行く。また、セグメント間で共有できるパイプラインは最大限利用するが、主記憶の制限などにより、中間結果が主記憶に保持できない場合には、そのセグメントの途中でパイプラインを切断し、いったん、中間結果をディスクに書き戻す。そして、切り離したセグメントのパイプラインのデータをセグメント木の下部の処理が終わった時点で、改めて次のパイプライン処理として読み出しを開始する。つまり、パイプラインを切断しなくてはならない場合には複数セグメント内のパイプラインを構成している結合演算の処理を分割し,

```
memsize = total memory of nodes;
QSET = specified queries set;
BSET = φ;
OPT-Set = φ;
BSET = cut_queries(QSET); WSET = weigt_segment(BSET);
do {
      tmpset = makePLAN(BSET);
      OPT-Set = costPLAN(tmpset);
      permute BSET in queries;
}until (all permutaions in queries of BSET are checked)


makePLAN :
queryset = BSET;
while(queryset != null){
    the seed query = top of queryset;
    while(all segment of seed query are not scheduled){
        get the target segment from the seed query;
        while (find a candidate segment by weight factor)
        and (free memory is available) {
            check sequence between segments;
            merge target and candidate segments;
            delete the candidate segment from queryset;
        }
    if the seed query is scheduled,
        set the second query to the seed query from queryset
    }
}
```

表 A.2: セグメントベース最適化方式の流れ

その中間結果を次のセグメントで利用することで全体の処理コストが最適結果となるようアルゴリズム (セグメントベース最適化方式) を提案する。本方式では、それぞれの単一問合せ処理の最適化された結果の木の形状は保持されているため、計算コストはさほど大きくない。また、ネットワークを通したデータ転送コストと入出力コストのオーバラップを考慮することで単純に主記憶による入出力コストの削減を目指すだけでなく実際の限られたシステム資源に対応できる最適化スケジューリングを生成することが可能である.

セグメント間での資源共有の重み付けには以下にあげられる 6 つの要素について、共有することで削減できる入出力コスト、ネットワーク転送コストをそのまま重み付けの値として用いる。

1. ビルドフェーズでのソース・リレーション
2. プローブフェーズでのソース・リレーション
3. パイプライン上での中間結果
4. 主記憶上でハッシュテーブルとしての中間結果
5. プローブリレーションとしてのディスク上の中間結果
6. ビルドリレーションとしてのディスク上の中間結果

上記の 6 つ条件については, 該当する複数問合せの全セグメント集合の中から条件を満足する部分木のグループを探し出し、セグメント単位のコスト低減をまず求める。しかし、全体のコストの最適化を得るためには、それぞれのセグメント間で求めたコストに加え、セグメントごとのスケジューリ

ングによる中間結果の書き戻し、主記憶上への保持が可能であるかんど、単一問合せの最適結果を逐次に実行する場合とは異なるスケジューリングコストを求めることが必要である.

セグメントベース最適化方式の処理の流れを表 A.2に示す. それぞれの問合せはすでに最適化されているおり,cut_queries() によりセグメント単位に分割する.weight_segment() により各問合せ間の weight factor を計算する. 本方式の weight factor は, 共有可能なパイプラインサイズ, 削減可能な入出力サイズと共有することにより増加するネットワーク上のデータサイズとの差から求められる. 最後に,makePLAN により segment を一つの処理単位として, 問合せ内の segment 処理シーケンスは変更せず,priority query を決めることで同時に処理する weight factor の大きいな segment を決定し, 処理スケジュールを求める. 求めたスケジュールの処理コストを計算し, 最も小さなコストのものをバッチ処理最適化処理とする.

本方式では, そもそも問合せ内のセグメント処理順序は変更することがないため, セグメント処理順序に合わないセグメント同士は,weight factor が大きくても同時に処理をすることはない. 従って, 処理スケジューリングを行なうための探索空間は小さくなるが, 生成されるスケジューリングの質は常に最適なものが得られるとは限らない.

A.2.3 オペレーションベースバッチ最適化方式

```
Operation Base Optimization ALGORITHM :
memsize = total memory size of Pnode;
JGRAPHSET = set of join graphs;
Rel-SET = φ;
OPT-Set = φ;
while(JGRAPHSET != φ) {
      Rel-SET = weight_rel(JGRAPHSET);
      OPT-Set += makeSEG(Rel-SET,JGRAPHSET);
}
makeSEG :
used memsize = 0;
find the maximum weight relation
set maximum relation to the probe relation
while (memsize > used memsize) {
      find the maximum relation which has an edge
      to the probe relation and assign it to the hash table;
      used memsize += the relation size;
      reduce join grapn in JOINGRAPHSET
      calculate network cost
      if (network cost > I/O of the probe relation)
            break;
}
```

表 A.3: オペレーションベース最適化方式の流れ

前述のセグメントベース最適化方式は,2 フェーズ最適化方式であり, 各問合せ内での処理シーケンスの変更は考慮していない. 従って, 複数の問合せ間で最も利用度の高いリレーションが異なる問合せの中で同じ処理シーケンスでセグメントの中に配置されるとは限らず, 最適なスケジューリングを得られない可能性が高い. そこで, 問合せ最適化とバッチ処理最適化を同時に行なう最適化方式 (オペレーションベース最適化方式) を提案する. 本方式では, まずジョイングラフを基に各結合演算に利用されるリレーションごとの weight factor を計算し, 最も weight factor の重いリレーションをベース

プローブリレーションとし, 複数問合せの処理をあらかじめ一つのセグメント処理として生成する.

オペレーションベース最適化方式の処理の流れを表?? に示す. メインループでは, すべての問合せがスケジュールされるまで, 共有セグメントを生成する. まず,weight_rel() により, relation の共有度についてそれぞれの問合せごとの join graph から weight factor の計算を行なう. weight faactor には各リレーションごとに各問合せで共有されている数, それぞれの join graph のエッジから得られる共有パイプ数について I/O およびネットワークサイズで正規化したものを用いる. 続いて, 計算されたリレーションの共有度を基にセグメント単位でのスケジュールを生成する. 本方式では, 最も重い weight factor を持つリレーションをプローブリレーションとすることで, パイプライン共有度を上げる heuristic を採用している. また, ハッシュリレーションはプローブリレーションに対するエッジを持っているものを優先的に配置するが, 主記憶に収まる範囲でハッシュテーブルを生成するとともにネットワークコストがプローブリレーションの入出力コストを越えないようにする. 一段のセグメントにおけるスケジュールが決定した段階で, それぞれの問合せの join graph を reduction して, 次のセグメントでのリレーションの weight factor を再計算する.

本方式では, セグメントベース最適化方式と異なり, 問合せごとの最適化ではなく, バッチとしての最適化をオペレーションベース (特にここでは多重結合演算を対象とした) で行なった. 本方式では従来の問合せごとの最適化によるオペレーションシーケンスが定まっていないため, リレーション共有及びパイプライン共有を各セグメントごとに有効にスケジューリングすることが可能である.

A.3 性能評価

Description	Value
number of nodes	16
clock rate of individual processor	100MHz
total memory size	150MB
data transfer rate of sequential read from disk	4.8MB/sec
data transfer rate of random read and write to disk	2.4 MB/sec
data transfer rate through the network	24MB/sec
number of instructions to move a tuple from disk buffer (network buffer) to local memory	50
number of instructions to split a join attribute	100
number of instructions to hash a join attribute	100
number of instructions to probe an entry of hash table with a tuple	500

表 A.4: 性能評価の環境 (1) – システムパラメタ –

本節ではセグメントベース最適化方式の効果を示すために、最適化を行わない場合と比較して、どれだけコストの低減が得られるかについて、シミュレーションによる評価を行った。表 A.4に、こ

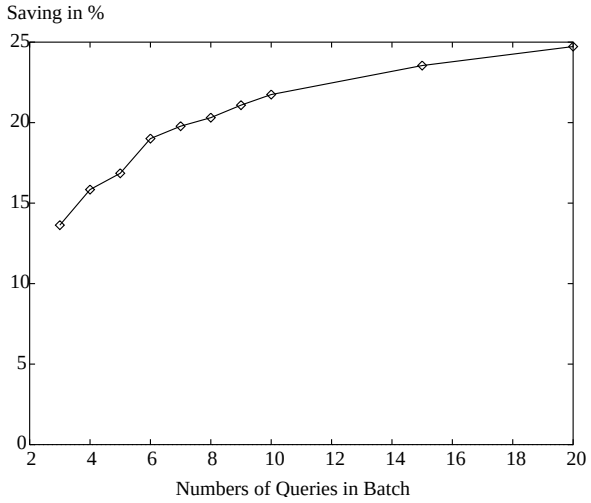

図 A.6: 最適化の結果 (バッチサイズが変化した場合)

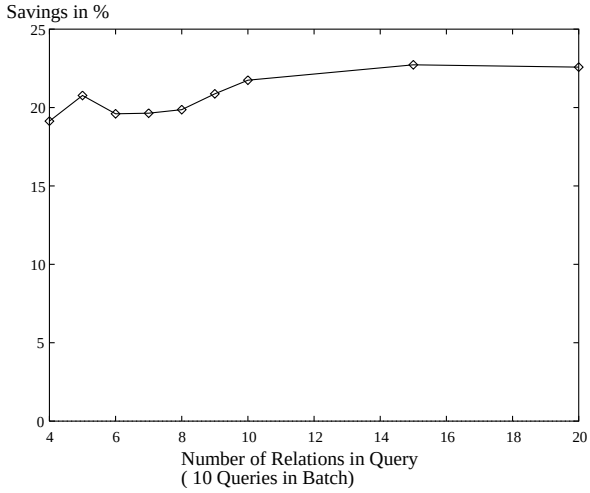

図 A.7: 最適化の結果 (リレーションサイズが変化した場合)

のシミュレーションで用いた分散メモリ環境のシステム構成 (イーサネットで結合された PC クラスタ相当) を示す。ノードは１６台とし、それぞれのノードに主記憶１５０ MB が搭載されている。また、シミュレーションに用いたコスト式は第５章の多重結合演算最適化方式で提案したコスト式を用いている。

複数問合せは１００個用意し、ソースリレーションは、１００リレーションの中から 80:20 のルールに従って、選ばれる。各リレーションの結合グラフは文献 [19] に従う。リレーションサイズは 100K タプルから 200K タプルまで変化し、タプル長は２５６バイトとする。また、選択率は 100 プルから 400K タプルまで変化する。ソースリレーションは全てのノードに均一に分散するとした。

シミュレーション結果を図 A.6、図 A.7、図 A.8に示す。複数問合せ処理の最適化結果が、最適化を行わなかった場合と比較して、どれだけコスト削減が可能となったかを異なる複数問合せを１００個流した平均の値で示している。

図 A.6は、複数の問合せをまとめた最適化する際の問合せの個数を変化させた場合の結果を示している。図から分かるとおり、いずれの最適化結果も 10% 以上のコスト削減となっている。しかし、最適化の対象となる問合せ数が少ないと、資源として共有できるリレーションの候補が減ってしまうため、一度にまとめて実行する問合せ数が少ないと最適化により性能向上も少なくなる。一方、問合せ数が８個以上のものに対して最適化を行うと、 20% 以上のコスト削減が見込まれる。

図 A.7は、 10 個の問合せからなる複数問合せの問合せ内リレーションの数を変化させた場合の結果を示している。図から分かるとおり、問合せが 10 個であるため、いずれの最適化結果も 20% 以上のコスト削減となっている。しかし、リレーション自体は１００個の中から 80:20 で選ばれるため、頻繁に共有、参照されるリレーションの数自体は限られており、一つの問合せに含まれるリレーションの数が増えても、さほど共有度は上がらないと。結果としてコストの削減はほぼ一定となっている。

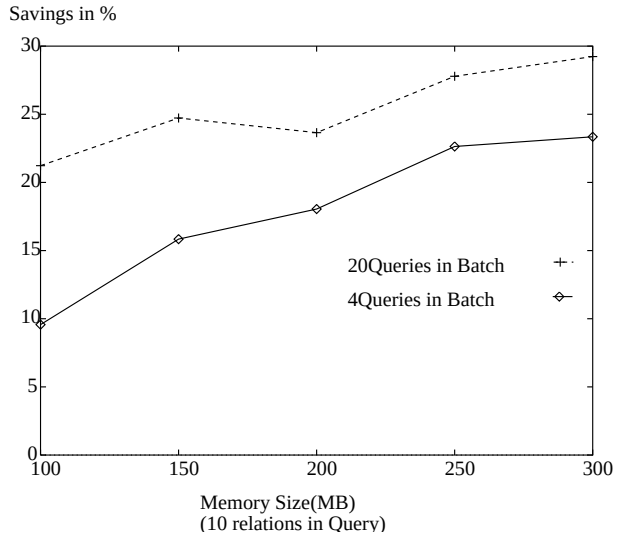

図 A.8: 最適化の結果 (メモリサイズが変化した場合)

図 A.8は、主記憶のサイズを変化させた場合の結果を示している。複数問合せとしては、２０個の問合せからなるものと４個の問合せからなるものを選び、いずれの問合せでもリレーションは１０で構成されている。図から分かるとおり、主記憶が増えると、資源の共有による最適化のコスト削減は大きくなる。これは、主記憶上に保持できるリレーションの数が増えるにつれ、入出力コストを削減することができることによる。図 A.6でも示したように、最適化の対象となる問合せ数が少ない (4 個) の場合は、得られる利得は少なくなるが、一方問合せ数が限られているため、主記憶の保持できる効果が期待できるため、主記憶が大きくなるほど、コスト削減の効果が得られている。

A.4 本章のまとめ

非共有型並列計算機環境における関係データベースシステムの性能向上を目指し、最も負荷が重い多重結合演算処理から構成されるバッチ問い合わせ処理スケジューリングの最適化について検討を行なった。本報告では、多重問い合わせの最適化において従来考慮されていなかったパイプライン処理について, 重み付け条件を入れることで最大限パイプライン効果を活かすようにスケジューリングする方式としてセグメントベース最適化方式とオペレーションベース最適化方式を提案した。またセグメントベース最適化方式について、シミュレーションによる性能評価を行い、最適化の効果について示した。今後, 両方式についてその特長および有効性を確認すべく, 複数のテストデータベースを用いた性能比較を行なうと共に, データベース性能評価用ベンチマーク TPC-D を用いた性能評価を行なう予定である.

付録 B

発表文献

1. 国内学会など

(1) 学会誌

1. 分散共有メモリ計算機における並列ハッシュ結合演算処理の性能解析
   中野 美由紀, 今井 洋臣, 喜連川 優
   電子情報通信学会論文誌, Vol.J82-D-I, No.1, pp.82-97, 1999.1

2. Examination of Criteria for Choosing a Run Time Method in GN Hash Join Algorithm
   Miyuki Nakano, Masaru Kitsuregawa
   IEICE Transactions on Information and Systems, Vol.E79-D ・ No.11, pp.1561-1569, 1996.11

3. GN ハッシュ結合方式とその評価
   中野美由紀, 喜連川優
   情報処理学会論文誌, Vol.35, No.9, pp.1861-1873, 1994.9

4. 共有メモリ型マルチプロセッサによる並列ハッシュ結合演算処理とその評価
   喜連川優, 津高新一郎, 中野美由紀
   情報処理学会論文誌, Vol.34, No.5, pp.1019-1030, 1993.5

5. 機能ディスクシステム: 関係データベース処理とその性能評価
   喜連川優, 中野美由紀
   電子情報通信学会招待論文, Vol.J74-D-I No.8, pp.496-507, 1991.8

(2) 国内シンポジウム

1. 機能ディスクシステム第２版
   中野美由紀、喜連川優、高木幹雄
   情報処理学会コンピュータアーキテクチャシンポジウム論文集, pp.37-46, 1988

2. 機能ディスクシステム第２版における関係代数演算処理方式とその評価
   中野美由紀、平野聡、喜連川優、高木幹雄
   情報処理学会アドバンストデータベースシンポジウム, pp.91-98, 1988

(3) **国内研究会**

1. 分散共有メモリ計算機上におけるデータスキューに対する結合演算処理の性能解析
   中野 美由紀，喜連川 優:
   情報処理学会、データベース研究会報告,Vol.121, No.7, pp,37-44, 2000

2. 共有メモリマルチプロセッサ上でのトランスポーズドファイルを用いた並列関係問合せ処理
   武藤 精吾，田村 孝之， 中野 美由紀，喜連川 優:
   電子情報通信学会データ工学研究会,Vol.97, No.161, pp,13-18, 1997

3. 分散共有メモリ計算機における並列ハッシュ結合演算処理方式の設計と実装
   今井 洋臣， 中野 美由紀，喜連川 優:
   電子情報通信学会データ工学研究会,Vol.97, No.161, pp,19-24, 1997

4. 並列データベースシステムにおける多重結合演算処理の最適化とその評価
   中野美由紀, 新谷隆彦, 喜連川優
   情報処理学会研究報告計算機アーキテクチャ研究会 Vol.95,No.80, 1995.8

5. 並列データベースシステムにおける多重結合演算処理の最適化
   中野美由紀, 喜連川優
   電子情報通信学会 1994 年並列／分散／協調処理に関する『琉球』サマーワークショップ， CPSY 94-27, pp.1-8, 1994.7

6. 共有メモリ型マルチプロセッサマシンにおける並列結合演算処理
   津高新一郎， 中野美由紀，喜連川優，高木幹雄
   並列／分散／協調処理に関する『大沼』サマー・ワークショップ第４回ＳＷｏＰＰ大沼'91 電子情報通信学会, CPSY91-6, pp.17-24, 1991.7

7. KD-tree based File Organization in FDS-R - FDS-R Performance Evaluation of the Join Oepration -
   原田リリアン、 中野美由紀、喜連川優、高木幹雄
   電子情報通信学会データ工学研究会、 DE88-4, pp.25-32, 1988.

8. 機能ディスクシステムにおける大規模リレーションの性能評価
   中野美由紀、平野聡、喜連川優、高木幹雄電子情報通信学会データ工学研究会、 DE88-4, pp.25-32, 1988.

9. 機能ディスク・システム (FDS-R) における問合わせ処理方式
   中野美由紀、喜連川優、高木幹雄
   電子情報通信学会データ工学研究会、 DE86-25, pp.19-24, 1987.3

10. 機能ディスク・システム (FDS-R) の性能評価
喜連川優、原田リリアン、 中野美由紀、高木幹雄
電子情報通信学会データ工学研究会、 DE86-26, pp.25-30, 1987.3

11. 機能ディスクシステムとその評価
喜連川優、原田リリアン、 中野美由紀、高木幹雄
情報処理学会計算機アーキテクチャ研究会、 1986.7

(4) **国内大会**

1. 共有メモリ型計算機上でのトランスポーズドファイルを用いた並列関係問合せ処理の実装方式とその評価
武藤精吾, 田村孝之, 中野美由紀, 喜連川優情報処理学会第 55 回全国大会講演論文集, Vol.3・No.6F-1, 1997.

2. 分散共有メモリ並列計算機における関係結合演算の性能解析
中野美由紀, 今井洋臣, 喜連川優
情報処理学会第 55 回全国大会講演論文集, Vol.3・ No.4AC-4, 1997.

3. トランスポーズドファイル上での並列結合演算処理方式に関する一考察
武藤精吾・田村孝之・ 中野美由紀・喜連川優情報処理学会第 54 回全国大会講演論文集, 3R-6, 1997.3

4. 分散共有メモリ計算機における並列ハッシュ結合演算方式の実装
今井洋臣, 中野美由紀, 喜連川優情報処理学会第 54 回全国大会講演論文集, Vol.3・ No.1R-1,1997.3

5. 分散共有メモリ計算機における並列ハッシュ結合演算方式の一考察
今井洋臣, 中野 美由紀, 喜連川　優情報処理学会第 53 回全国大会講演論文集, Vol.3 ・ No.1R-8, 1996.9

6. 並列関係データベースシステムにおけるバッチ問合せ処理最適化技法の検討
中野美由紀, 喜連川優
情報処理学会第 52 回 (平成７年後期) 全国大会講演論文集,3Q-03, 1996.3

7. 並列関係データベースシステムにおける多重問合せ最適化に関する一考察
中野美由紀, 新谷隆彦, 喜連川優
情報処理学会第 51 回（平成７年後期）全国大会講演論文集 7D-1, 1995.9

8. 並列計算機 AP1000DDV における多重結合演算の実装とその評価
新谷隆彦, 中野美由紀, 喜連川優
情報処理学会第 51 回（平成７年後期）全国大会講演論文集 7D-4, 1995.9

9. 並列データベースシステムにおける多重結合演算の静的最適化技法の考察
中野美由紀, 喜連川優
情報処理学会第 49 回（平成６年後期）全国大会講演論文集, 7W-1,1994.9

10. 不均一なデータ分布におけるＧＮハッシュ結合方式の性能評価
中野美由紀, 喜連川優
情報処理学会第 48 回（平成６年前期）全国大会講演論文集, 1F-5, 1994.3,

11. 共有メモリ型マルチプロセッサ Symmetry S81 による複合問合せ処理方式
津高新一郎, 中野美由紀, 喜連川優, 高木幹雄
情報処理学会第 43 回（平成３年後期）全国大会講演論文集, 6N-8, 1991.10,

12. Symmetry S81 における結合演算の並列処理の性能評価
津高新一郎, 中野美由紀, 喜連川優, 高木幹雄
情報処理学会第 42 回（平成３年前期）全国大会講演論文集, 4L-10, pp.167-168,1991.3

13. Symmetry s81 における結合演算の並列処理に関する考察
津高新一郎, 中野美由紀, 喜連川優, 高木幹雄
情報処理学会第 41 回（平成２年後期）全国大会講演論文集, 1D-6, 1990.9,

14. Symmetry S81 における GRACE HASH 方式の実装と評価
津高新一郎、 中野美由紀、喜連川優、高木幹雄
情報処理学会第 40 回（平成２年前期）全国大会講演論文集, 5H-5, 1990.3,

15. ＧＮハッシュ結合方式－入出力コストによるハッシュ結合方式の性能比較－
中野美由紀、喜連川優、高木幹雄
情報処理学会第 40 回（平成２年前期）全国大会講演論文集, 7H-4, 1990.3,

16. 不均一分布データに対するハッシュを用いた結合演算アルゴリズムの性能評価－機能ディスクシステムの場合－
中野美由紀、喜連川優、高木幹雄情報処理学会第 39 回（平成元年後期）全国大会講演論文集, 3N-9, 1989.10,

17. 機能ディスクシステム (FDS-R) 第２版に於ける結合演算処理方式
中野美由紀, 平野聡、喜連川優、高木幹雄
情報処理学会第 36 回全国大会講演論文集, 5E-1, 1988.3

18. 機能ディスクシステム (FDS-R) におけるファイル編成の検討
原田リリアン、 中野美由紀, 喜連川優、高木幹雄
情報処理学会第 36 回全国大会講演論文集, 5E-2, 1988.3

19. 機能ディスクシステム (FDS-R) 第２版に於ける結合演算の性能評価
中野美由紀, 平野聡、喜連川優、高木幹雄
昭和６３年電子情報通信学会秋期全国大会講演論文集， D-1-18, 1988

20. 機能ディスクシステム第２版に於ける結合演算処理の考察
中野美由紀, 平野聡、喜連川優、高木幹雄
情報処理学会第 37 回全国大会講演論文集， 7Q-3, 1988

21. 機能ディスクシステム第２版に於ける集約演算処理の考察
中野美由紀, 平野聡、喜連川優、高木幹雄情報処理学会第 38 回全国大会講演論文集， 4Q-4, 1988

22. 密結合マルチプロセッサに於ける関係代数演算の評価 - 結合演算 -
中野美由紀, 喜連川優、高木幹雄
情報処理学会第 35 回全国大会講演論文集， 4CC-1, 1987.9

23. 機能ディスクシステム (FDS-R) に於ける非一様分布データに対する性能評価
原田リリアン、 中野美由紀, 喜連川優、高木幹雄
情報処理学会第 35 回全国大会講演論文集， 4CC-3, 1987.9

24. 機能ディスクシステム (FDS-R) に於ける Query 処理方式
中野美由紀, 喜連川優、高木幹雄
情報処理学会第 34 回全国大会講演論文集， 5Q-7, 1987.3

25. 機能ディスクシステム (FDS-R) への QUEL サブセットの実装
中野美由紀, 喜連川優、高木幹雄
情報処理学会第 33 回全国大会講演論文集， 5H-8, 1986.10

26. 機能ディスクシステム (FDS-R) に於ける Aggregation Query の性能評価
原田リリアン、 中野美由紀, 喜連川優、高木幹雄
情報処理学会第 33 回全国大会講演論文集， 5H-9, 1986.10

27. 機能ディスクシステム (FDS-R) に於ける Projection Query の性能評価
原田リリアン、 中野美由紀, 喜連川優、高木幹雄
情報処理学会第 34 回全国大会講演論文集， 5Q-8, 1987.3

28. 機能ディスクシステムにおけるシステムソフトウエアの設計
喜連川優、 中野美由紀, 高木幹雄
情報処理学会第 32 回全国大会講演論文集， 5S-3, 1986.3

2. 国際会議発表など

1. Performance Analysis of Paralell Hash Joins on a Dsitributed Shared Memory Machine
   Miyuki Nakano, Hiroomi Imai and Masaru Kitsuregawa
   IEEE International Conference on Data Engineering, pp.76-85(1998)

2. Implementation and Evaluation of Parallel Relational Query Processing Using Transposed Files on Shared Memory Multiprocessors
   Seigo Muto, Takayuki Tamura, Miyuki Nakano, Masaru Kitsuregawa,
   Advanced Database Research and Development Series Vol.8,pp.192-198,1998

3. Parallel Query Processing using Transposed Files:
   Seigo Muto, Takayuki Tamura, Miyuki Nakano, Masaru Kitsuregawa,
   Advanced Systems for Integration of Media and User Environments'98, pp.156-166,1998

4. GN Hash Join Algorithm :A Robust Algorithm for Non-uniform Data Distribution
   Miyuki Nakano and Masaru Kitsuregawa
   International Symposium on Advanced Database Technologies and Their Integration(ADTI'94), pp.121-128, Nara, Japan, 1994.10

5. An Effective Parallel Processing of Multi-Way Joins by Considering Resources Consumptions
   Lilian Harada, Naoki Akaboshi and Miyuki Nakano, Proc. of 6th Int. Conf. on Computing and Information, (1994)

6. Parallel Execution Plans of Multi-Way Joins in Shared-Nothing Database Systems : On Trade-Offs of System Resources Consumption
   Lilian Harada, Miyuki Nakano
   International Symposium on Advanced Database Technologies and Their Integration(ADTI'94), pp.113-120(1994)

7. Parallel GRACE Hash Join on Shared-Everything Multiprocessor:Implimentation and Performance Eveluation on Symmetry S81
   Masaru Kitsuregawa,Shinichiro Tsudaka,Miyuki Nakano Proc. of IEEE 8th Int. Conf. on Data Engineering, pp.256-264, 1992.2

8. Performance Evalution of Functional Disk System(FDS-R2)
   Masaru Kitsuregawa, Miyuki Nakano,Mikio Takagi
   Proc. of IEEE 7th Int. Conf. Engineering pp.416-425, 1991.4

9. Query Processing Method for Multi-Attribute Clusterd Relations
   Lilian Harada, Miyuki Nakano, Masaru Kitsuregawa, Mikio Takagi
   16th. International Conference on Very Large Data Bases, pp.59-70, Brisbane, Australia, 1990.8

10. Performance Evalution of Functional Disk System with Nonuniform Data Distribution
    Masaru Kitsuregawa, Miyuki Nakano, Lilian Harada, Mikio Takagi
    Proceedings of the second International Symposium on Databases in Parallel and Distributed Systems, pp.80-89, Dublin, Ireland, 1990.7

11. Functional Disk System as a High Performance Relational Storage
    Masaru Kitsuregawa,Miyuki Nakano,Mikio Takagi
    International Symposium on Database Systems for Advanced Applications (Invited Paper) pp.243-250 1989.4

12. Query Execution for Large Relations on Fuctional Disk System
    Masaru Kitsuregawa, Miyuki Nakano,Mikio Takagi
    IEEE Proc of 5th Int. Conf. on Data Engineering, 1989

13. Functional Disk System for Relational Database
    Masaru Kitsuregawa, Miyuki Nakano,Lilian Harada,Mikio Takagi
    Proc of 3rd Int. Conf. on Data Engineering, 1987

14. Performance Evaluation of Functional Disk System with Aggregation Query
    Masaru Kitsuregawa,Miyuki Nakano,Lilian Harada,Mikio Takagi
    IEEE Proc. of International Conference on Computer Design, pp.206-210, 1986

3. その他

1. 第６章 データベース・マシンのアーキテクチャ
   喜連川優、 中野美由紀
   新しい計算機アーキテクチャ、丸善、 1991,3

2. データベース・マシン
   喜連川優、 中野美由紀
   並列コンピュータ・アーキテクチャ bit 共立出版 vol.21 No.4 pp.403-416 1989.3