大規模ＰＣクラスタによる超並列関係データベースサーバの構築

田村孝之

(1)

大規模 PC クラスタによる

超並列関係データベースサーバの構築

田 村 孝 之

目次

図一覧

表一覧

第 1 章

序論

1.1 本研究の目的

近年のデータベース応用では, 更新系の性能向上に伴って蓄積された大量のデータに統計的な解析を加え, 意思決定に利用することへの関心が高まっている. 大規模なデータ解析処理には高い I/O 性能が要求されるため, 並列計算機の利用が一般化しつつあるが, 現行の商用 DBMS においては並列化や意思決定問合せへの対応が不十分であり, 高価な専用ハードウェアや大量資源の投入, あるいは個々の事例に特化した最適化などによって性能向上を図っているのが実状である. そこで, 著者の属する東京大学生産技術研究所喜連川研究室では, 意思決定問合せ処理の飛躍的な性能向上を目指し, 高性能な並列関係演算アルゴリズムを独自の並列アーキテクチャおよびシステムソフトウェア上に実現する, SDC – Super Database Computer – プロジェクトを数年に亙って進めてきた.

一方, マイクロプロセッサや標準バスの高速化によるパーソナルコンピュータ (PC) の性能向上や, ATM に代表される次世代標準ネットワークの技術開発の進展は著しく, これらを構成要素として並列処理システムを構築することで従来の超並列計算機に対して圧倒的に高い性能対価格比が実現可能になってきた. そこで本研究では, 100 台の PC を ATM スイッチで結合した大規模 PC クラスタ上に, SDC システムソフトウェアを大規模化 / 一般化した並列意思決定問合せ処理系を構築することで, PC クラスタによる超並列関係データベースサーバ実現の可能性を示すことを目的とする.

1.2 関係データベースシステムにおける意思決定問合せ処理

1970 年に E.F.Codd によって提案された関係データベースシステム (Relational Database System) [7] は, その堅固な理論的基盤を背景に広く用いられるようになってきた. ネットワーク型に代表される関係型以前のデータベースシステムにおいては, データの物理的な格納法とアクセス法が直接反映され, データベース定義にデータ間の関連が明示的に含まれていたため, データの検索はリンクを辿る手続き的なものになり, リンクの存在しない経路を辿るアクセスができないという難点が存在した. これに対して関係データベースシステムでは, データはリレーション (関係) と呼ばれる表形式に抽象化されており, データ間の関連はリレーションに対して定義された関係代数演算を用いて動的に導出される. そのため, データの検索は結果リレーションを導出する関係代数式として非手続き的に記述することができ, アクセス方法の自由度も高い.

関係データベース処理はその形態に応じて以下のように分類できる.

- オンライントランザクション処理 (OLTP)

- 定型問合せ処理
- 非定型問合せ処理

オンライントランザクション処理は現金自動引き出しシステムや座席予約システムに代表されるものであり, 小さな粒度の更新処理が高い頻度でかつ並行して発生することを特徴とする. また, 定型問合せ処理は企業における会計業務などに見られ, 一日あるいは一ヵ月などの一定期間毎に既定のレポートを作成するものである. これらの利用形態においては頻繁に発生する検索処理を予測できるため, インデックスを作成し, コストの高いリレーション全体の検索を避けることで高速化可能である. また, アクセス頻度の高い一部のデータを主記憶にキャッシュするために大量メモリの投入とバッファ管理の最適化が行われる. しかし, 非定型問合せ処理においては ad hoc な (一度限りの) 問合せが発行されるため, インデックスやバッファは有効に機能せず, リレーションの全件検索の発生は避けられない. そのため, OLTP 性能が改善されるほど処理性能の乖離が大きくなり, 従来の関係データベースの利用はほとんど OLTP に限られてきた.

一方, 近年のハードウェアの格段の進歩と OLTP 技術の向上に伴い, 企業等の保有するデータベースが数テラバイト規模に達するに至り, このような大規模データベースを解析し, 企業活動を行う上での意思決定に有効活用することが求められるようになってきた. これまで OLTP 性能の標準ベンチマークとして TPC-A, TPC-B, および TPC-C の策定を行ってきた TPC (Transaction Processing Performance Council) も, 意思決定支援システムの標準ベンチマークとして TPC-D ベンチマークを 1995 年に発表した. また, 従来の関係データベースの枠組を超える新たなアプリケーションとして, OLAP (On Line Analytical Processing) やデータマイニングなどが注目を集め, 盛んに研究が行われている.

意思決定問合せに代表される大規模データベース解析処理には,

- 大量データにアクセス
- 非定型問合せを発行
- 負荷の高い関係代数演算 (結合演算, 集計演算など) を多用

という特徴があり, OLTP を重視してきた従来の関係データベースシステムでは実用に堪える十分な性能が得られないのは明らかである. そのため, このような処理の高性能化に関して過去多大な研究開発がなされてきた [38]. それらは, データベース処理専用プロセッサと並列データベースマシンの 2 通りのアプローチに大別できる. 前者は, 選択, ソート, 結合処理等を専用プロセッサで行うことで主プロセッサ (メインフレーム等) の処理負荷を軽減

し, 高速化を図るものである. 代表的なシステムとして, RINDA (NTT)[37], IDP/RDSP (日立)[20], GREO (三菱)[46], DBE (東芝) [40], DBA (富士通)[39] 等が挙げられ, 主として日本国内で盛んに研究ならびに商用化が行われてきた. 一方, 並列データベースマシンは複数のディスク装置やプロセッサを用いることで関係データベース処理に内在するデータ並列性を最大限に抽出しようとするものであり, ウィスコンシン大学の DeWitt らによる GAMMA[11, 12] や Teradata 社による DBC1012[8, 31, 4] 等が比較的初期に開発されたマシンとして知られている.

近年, 商用並列マシンの普及に伴い, IBM, Oracle, Sybase, Informix 等の代表的な DBMS ベンダが従来の単一プロセッサ版の DBMS を並列プラットフォーム上に移植して販売するようになった [1, 15]. しかし, これらの多くは OLTP を中心に発展してきた従来のシステムを継承したものであり,

- 全件検索時のバルク I/O
- 結合 / 集計演算処理

において十分な性能が得られているとは言い難い. これに対して, Red Brick Systems 社の Red Brick [27] のように, OLTP を犠牲にして意思決定問合せ処理の性能向上を行うシステムも受け入れられるようになってきたことから, リレーションの全件検索に最適化したアクセス法と高性能な関係演算アルゴリズムを備えた並列関係データベースシステムの構築は依然として重要な課題であると言える.

1.3 SDC プロジェクト

東京大学生産技術研究所喜連川研究室ではこれまで専用プロセッサと並列データベースマシンの両アプローチに亙って関係データベースの問合せ処理の高速化に関する研究を行ってきた. それらの内, 並列データベースマシン開発のプロジェクトは SDC — Super Database Computer — プロジェクトとして進められ, 1989 年から 1991 年にかけて第一版試作機 SDC-I [47, 45, 18] が, 1991 年から 1994 年にかけて第二版試作機 SDC-II [42, 41, 43, 33] が開発された.

SDC プロジェクトの目的は,

- 並列関係演算アルゴリズム
- スケーラブルな並列アーキテクチャ

- 問合せ処理指向のシステムソフトウェア

を備えた並列データベースマシンをマイクロプロセッサを用いて実現することであった. SDC では共有バスによる密結合マルチプロセッサシステムを処理モジュールとし, 複数の処理モジュールをオメガネットワークで相互接続した専用アーキテクチャが採用された. SDC-I は MC68020 20MHz 5 台, SCSI 8 インチディスク 2 台, および 8 MB の共有メモリを持つ処理モジュール 2 台から成り, SDC-II は MC68040 25MHz 7 台, SCSI 3.5 インチディスク 4 台 32 MB の共有メモリを持つ処理モジュール 8 台から成る. 要素プロセッサには市販のマイクロプロセッサを用いたが, 共有バス, 磁気ディスク I/F, 相互結合網などは全てボードレベルから設計された. そのハードウェア上に専用のシステムソフトウェアを構築し, 高性能な並列関係演算アルゴリズムを実装することで, 非常に優れた処理効率を示すことができ, マイクロプロセッサによる並列関係データベース処理の有効性が実証された.

しかし, 専用ハードウェアの開発に多大な時間を要したため, 最先端の商用並列 DBMS と比較すると要素プロセッサの速度, ディスク容量, メモリ容量などの点で見劣りするようになったことは否めない. また, 陳腐化したハードウェア要素部品をアップグレードすることも困難であり, 最新ハードウェアを利用するためにはシステム全体の構築を一から繰り返さなければならない. 結果として SDC は, 投入したハードウェア資源に対する性能で評価すると非常に優位にあるものの, 絶対性能を商用機と比較することは困難な状況にあることが明らかになった.

1.4 SDC から 大規模 PC クラスタへ

近年 Intel 社の Pentium 系マイクロプロセッサの高性能化や高速な PCI バスの標準化によってパーソナルコンピュータ (PC) の性能は著しい向上を続けており, 少なくとも整数演算性能に関しては, 高価なワークステーションと肩を並べるまでに至っている. 同時に, 年間数千万台にも達する市場規模を背景に低価格化も急激に進んでいる. これまでの並列計算機の歩みにおいては, 処理要素として専用プロセッサを用いたものから市販プロセッサ, さらには市販のワークステーションへという流れがあったが, 価格対性能比で圧倒的に有利な PC がこれに取って替わることは必至であると思われる.

一方, 商用の超並列コンピュータである IBM SP-2 や富士通 AP3000 では, 処理ノードには市販のワークステーションを採用しているものの, ノード間ネットワークには独自に開発された専用ハードウェアを用いており, 標準ネットワークに比べてバンド幅, レイテンシー (遅延) いずれも高い性能を提供している. しかしながら, 次世代標準 LAN の候補として

ATM やギガビットイーサネットなど幾つかの方式が提案され, 急速に技術開発が進められており, 十分に高いバンド幅が実現されるに至っている. そのため, ノード間の同期の頻度が少なく, レイテンシーの影響をあまり受けないアプリケーションに取っては, 標準ネットワークの採用によりコストを下げることが可能になってきている.

このような状況から, 独自ハードウェアの開発を要していた SDC を標準ハードウェアへ全面的に転換することで, 開発期間が短縮されると共に, 要素部品の急速な高性能化 / 低価格化に伴うアップグレードも可能になり, より大規模な並列関係データベースシステムの構築が可能になると考えられる. そこで, 本研究では要素ハードウェアとして「商用計算機 PC + 商用ネットワーク ATM」という構成を採用し, 100 ノードから成る大規模 PC クラスタの構築を行い, その上に並列関係問合せ処理系の実装を行った. これまでにも標準ハードウェアによる PC クラスタの研究は行われてきたが, 対象は全て数値演算であったため, 「商用計算機 WS + 専用ネットワーク」という構成のワークステーション (WS) クラスタには絶対性能で大きく劣っていた [32, 5]. しかし, データベース処理には浮動小数点演算性能やレイテンシーは大きく影響しないため, 性能・価格比のみならず絶対性能においても十分な成果を期待できる.

まず, SDC プロジェクトの最終的な評価を行い, その成果を継承・発展可能なものにするために, さまざまな問合せの実行や異なるプラットフォームへの移植を前提として SDC システムソフトウェアの再設計を行い, 並列問合せ処理系 DBKernel として提示した. そして, DBKernel を SDC-II 上に実装し, 標準ベンチマーク TPC-D (1 GB/10 GB) を用いた性能評価を行うことにより, SDC-II はハードウェア性能に比して高い性能を達成しているものの, 開発に多大な時間を要したため, 最先端の商用並列機とは絶対性能で大きな格差が生じていることを示した. 次に, 標準的な Unix 上で動作する DBKernel を大規模 PC クラスタ上に実装し, 100 GB の TPC-D ベンチマークによる性能評価を行った. その結果, 最も高性能な既存商用システムに対しても最高で 5 倍以上の問合せ処理性能が得られ, PC クラスタの有効性が単に性能対価格比だけに留まらず, 絶対性能においても充分存在すること, また処理系を意思決定問合せに最適化することの効果を明らかにした.

1.5 本論文の構成

以下, 第 2章では意思決定問合せに頻繁に用いられる結合演算や集計演算などの関係演算とその並列処理アルゴリズムの概要を示す. 第 3章では SDC のシステムソフトウェアを一般化・大規模化するために新たに設計された並列関係問合せ処理系 DBKernel について述べる. 次に, 第 4章では, SDC-II のハードウェアアーキテクチャとその上での DBKernel の

実装方法について説明する．そして，TPC-D ベンチマークを用いた性能評価結果を示し，公表されている商用並列システムの性能との比較を行う．第5章では，標準ハードウェアのみで構成された 100 ノード ATM 結合大規模 PC クラスタと可搬性を重視して行った DBKernel の実装方法について述べる．さらに前章と同様，TPC-D ベンチマーク (100GB) による性能評価を示し，システム全体の絶対性能においてもハイエンド商用並列システムを凌駕していることを明らかにする．第6章では，現実のデータに存在するデータスキューのもたらす問題を解決するための負荷分散アルゴリズムについて説明する．まず，SDC におけるアルゴリズムとハードウェアの協調による再分配スキューの解決法を述べ，その性能評価を行う．次に，結合演算スキューを解決するための動的負荷分散アルゴリズムを提案し，大規模 PC クラスタ上での実装と性能評価について述べる．第7章では，不要な属性の読み込みを回避することで I/O ボトルネックを改善する，トランスポーズドファイル編成の効果について述べる．第8章では，マルチユーザ環境での I/O 共有を実現する手法と多数の演算子をコンテクスト切替えなしに並行して評価する技法について述べ，それぞれの効果を示す．第9章では，本論文のまとめを行い，今後の課題について触れる．

第 2 章

関係演算とその並列処理方式

2.1 選択演算・射影演算

関係データベースシステムにおいては, リレーションを操作するための関係代数演算および集合演算や, 種々の統計値を求めるための集計演算 (COUNT, MIN, MAX, SUM, AVG) がサポートされており, ユーザはこれらの演算を用いて検索条件を指定する. 関係データベースシステムの基礎については文献 [9, 13] に譲り, ここでは後のベンチマークにおいて使用される演算の内, 重要なものについて述べる事にする.

関係代数演算の中で最も単純なものが選択演算 (selection) である. 選択演算は, 単一のリレーションの各タプルの内, 問合せの述語に指定された条件を満たすタプルのみからなるリレーションを生成する演算である. 選択条件に指定された属性に関するインデックスが存在する場合は, インデックスをたどる事で直接目的とするタプルを取り出す事が可能であるが, そうでない場合はリレーションの全タプルを読み, 条件を満たすものだけを出力するという方法で実現しなければならない. また, インデックスが存在する場合でも, 取り出されるタプル数が多い場合には, インデックスの格納されたディスクページをアクセスするコストや, ランダムアクセスによる I/O 性能の低下により, フルスキャンを用いた方が効率的な事もある.

また, 射影演算 (projection) は一つのリレーションからある属性のみを取り出して新たなリレーションを生成する演算である. 取り出される属性に元のリレーションの基本キーが全て含まれる場合は, 結果のリレーション中のタプルが一意に識別可能になるが, それ以外の場合には重複したタプルが生成される可能性がある. 本来の関係モデルにおいてはリレーション中のタプルは全て一意である事が要求されるため, 射影演算の際には常に重複したタプルを除去しなければならない. しかし, 関係データベースシステムの標準問合せ言語である SQL においては便宜上純粋な関係モデルとは若干異なるモデルを基にしているため, 重複除去は以下のようにユーザが `distinct` を用いて明示的に指定しない限り実行されない.

```
select distinct(Nationality) from Students
```

重複除去を伴わない場合の射影演算は, 選択演算同様単純であり, リレーション (この場合は SQL の概念になるためテーブルと呼ぶ方が適当) をフルスキャンし, 各入力タプル (行) から必要な属性 (列) だけを取り出して出力するという実現方法になる. インデックスは射影演算には効果がないが, 本論文の第 7章で述べるトランスポーズドファイルを用いるとそれぞれの属性に独立してアクセスできるため, 不要な属性の読み込みを避ける事ができる.

重複除去は比較的負荷の高い演算であり, その実行方式にはソートに基づくものとハッシュに基づくものとが挙げられる [14]. 結果リレーションの大きさが主記憶容量を越える場合

には, 重複除去前のタプルを複数の中間ファイルに分割して出力する必要がある. この中間ファイルは, ハッシュに基づくアルゴリズムにおいてはバケットと呼ばれ, それぞれのバケット内の全てのタプルが同一のハッシュ値を持つように分割される. この処理は, 各バケットから生成される結果リレーションの大きさが主記憶容量より小さくなるまで繰り返される. その後, 各バケットに対して以下の処理を行う.

1. 主記憶上にハッシュテーブルを作成.
2. バケットからタプルを一つ読み込む.
   バケットが空になったら終了.
3. タプルの全ての属性をキーとしてハッシュテーブルを検索.
4. 検索が失敗した場合, 重複するタプルはまだ存在しないので, 入力タプルをハッシュテーブルに登録し, 2. に戻る.
5. 検索が成功した場合, 重複するタプルが存在する事になるので入力タプルを捨て, 2. に戻る.

バケットの全てのタプルを処理し終えた時にハッシュテーブル上に登録されているタプルが結果となる. また, 各バケットの処理は相互に関連がないため, 異なる処理ノードで並列に処理する事が可能である. この場合, 全ノードの主記憶を利用できるため, 中間ファイルの生成を省略できる可能性が高くなる.

2.2 集計演算

集計演算は種々の統計値を計算するためのものであり, SQL では以下のものが用意されている.

COUNT – タプル数を求める.

MIN – 数値属性について最小値を求める.

MAX – 数値属性について最大値を求める.

SUM – 数値属性について総和を求める.

AVG – 数値属性について平均値を求める.

集計演算はリレーション全体だけでなく,

```
select Age, count(*)
from   Students
group by Age
```

などのように用いて, 指定した属性値の異なるグループ毎に適用することもできる.

グループ毎の集計は負荷の高い演算であり, 重複除去アルゴリズムと同様, その実行方式にはソートに基づくものとハッシュに基づくものとが存在する. ハッシュに基づくアルゴリズムにおいては, グループ数が主記憶上で処理可能なほど小さくなるまで入力タプルのグループ化属性にハッシュ関数を施して複数のバケットを生成する. その後, 各バケットを以下のように処理する.

1. 主記憶上にハッシュテーブルを作成.
2. バケットからタプルを一つ読み込む.
   バケットが空になったら終了.
3. タプルのグループ化属性をキーとしてハッシュテーブルを検索.
4. 検索が失敗した場合, ハッシュテーブルに項目を登録し, 初期化する. COUNT については初期値 1 を, MIN, MAX, SUM については入力タプルの値を, AVG については合計値として入力タプルの値を, また要素数として 1 をそれぞれ設定する. 2. に戻る.
5. 検索が成功した場合, ハッシュテーブル上の項目に対し, 集計演算の種類に応じて以下の処理を行う.

   COUNT – 値を 1 増やす.

   MIN – 入力タプルの値の方が小さかったらその値を設定.

   MAX – 入力タプルの値の方が大きかったらその値を設定.

   SUM – 入力タプルの値を加える.

   AVG – 合計値に入力タプルの値を加え, 要素数を 1 増やす.

   2. に戻る.

バケットの全てのタプルを処理し終えた時にハッシュテーブル上に登録されている項目が結果となる. また, このアルゴリズムも各バケットを異なる処理ノードに割り当てる事で並列

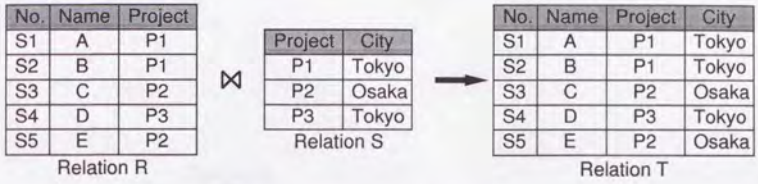


図 2.1: 結合演算の例

化が可能である [29]. ただし, グループ数が小さく, 単一ノードにおいてもバケット分割なしにハッシュテーブルを生成することができる場合には, それぞれのノードで部分的な結果を求めてそれを最後に収集することで処理ノード間の通信量を大きく減らす事ができる.

2.3 結合演算

結合演算 (join) は 2 つのリレーション中のタプルを指定された結合条件によって結合し, 新たなリレーションを導出する関係演算である. 図 2.1 にリレーション R と S とを,

```
select No, Name, Project, City
from   R, S
where  R.Project = S.Project
```

の条件の下に結合してリレーション T を導出する結合演算の例を示す.

結合演算は関係代数演算の中でもその処理負荷が極めて高いことから種々の研究がなされてきた [25]. 結合演算の手法としてはネストループ結合法, ソートマージ結合法, ハッシュ結合法の 3 つの方式が考案されているが, 特に大規模な演算においてはハッシュ結合手法が最も優位であることが知られている [10].

ハッシュ結合演算は, 一方のリレーションのタプルの結合属性値に対するハッシュ値を基にメモリ上にハッシュテーブルを作成するビルドフェーズと, 他方のリレーションのタプルでハッシュテーブルを探索し, 結果を得るプローブフェーズから成る. リレーションの大きさがメモリに収まらない場合には, ビルドフェーズに先立ってスプリットフェーズを実行し, リレーションを複数のバケットに分割する. バケットはメモリに収まるように分割し, バケット同士の突き合わせをすべてメモリ上で行なうため, バケット分割後のディスク入出力

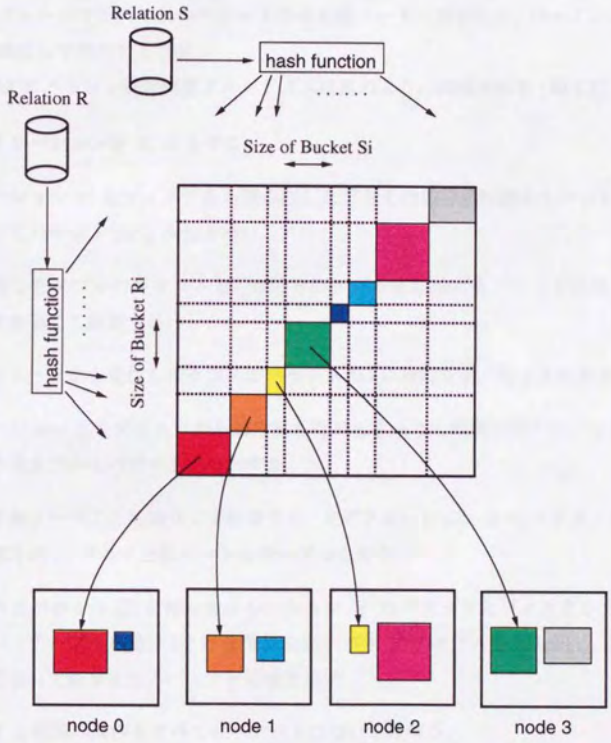


図 2.2: ハッシュ分割による結合演算の並列化

は最小限に押えられる．また，各バケットの処理を異なるノード上で行なうことにより容易に並列化可能である (図 2.2)．

2.3.1 並列 GRACE ハッシュ結合演算アルゴリズム

代表的な並列ハッシュ結合演算アルゴリズムである並列 GRACE ハッシュ結合演算では，スプリットフェーズでそれぞれのバケットを各処理ノードに振りわけ，ジョインフェーズをノード毎に独立して実行する [19]．

並列 GRACE ハッシュ結合演算アルゴリズムは次のように実装される (図 2.3)．

1. 入力リレーションを R, S とする．
2. リレーション R をディスクから読み出しながらその結合属性値からハッシュ関数 h_1 を用いてバケット ID を決定する．
3. 読み出したタプルのバケット ID からそのタプルを処理するノードを決定し，ネットワークを通じて転送する．
4. ネットワークから受信したタプルをバケット ID に対応するバケットに格納する．
5. リレーション S をディスクから読み出しながら 2 ～ 4 と同様に同一のハッシュ関数に基づき全タプルをバケットに分割する．
6. 以下を各ノードごとに独立して処理する．まずリレーション R のバケットをディスクから読み出し，メモリ上にハッシュテーブルを作る．
7. 6 と同じバケット ID を持ったリレーション S のバケットをディスクから読み出し，ハッシュテーブルを走査し，結合演算を施すべき R のタプルを探し出し，タプルの結合を行なって結果を出力バッファに書き出す．
8. 6 ...7 と同様の操作をすべてのバケットについて行なう．

以上が基本的な並列 GRACE ハッシュ結合演算のアルゴリズムであるが，現実の処理ではリレーションの分割に先立って選択演算等が施される場合が多く，この場合分割後の R リレーションのバケットの合計サイズは元のリレーションサイズよりも小さくなる事が予想される．また，分割後のバケットのサイズのばらつきは使用するハッシュ関数とリレーションの結合属性値の分布との関係によって決まり，既知でない場合も多い．したがって，バケットサイズが最大のものがステージングバッファの容量を超える事もあり得るため，この場合バ

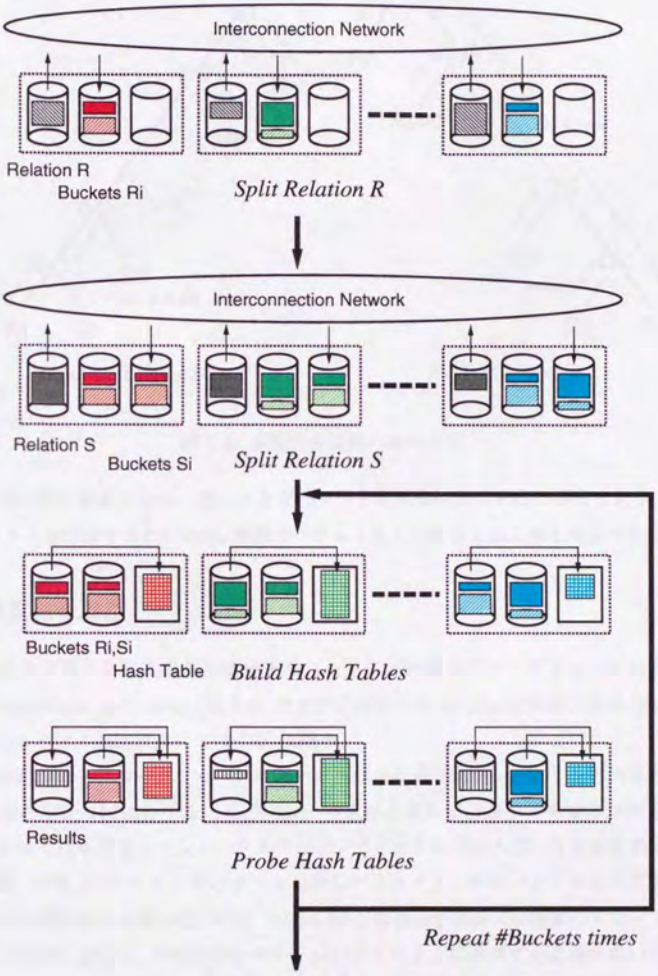
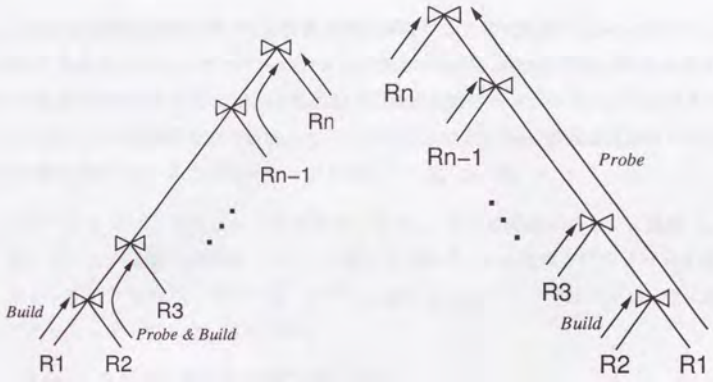


図 2.3: 並列 GRACE ハッシュ結合演算アルゴリズム



(a) Left-deep 木　　(b) Right-deep 木

図 2.4: 多重結合演算の実行方式

ケットの再分割が必要となる. 逆に小さなバケットが大量に生成される場合もあり, この場合 I/O コストを低減するためには, 複数のバケットを 1 つにまとめる事も有効である.

2.4 多重結合演算

複数の結合演算からなる多重結合演算はビルドフェーズとプローブフェーズの組合せによって Right-deep, Left-deep (図 2.4) およびそれ以外の Bushy に分類される [28]. 各々の特徴とコンパイル技法については [30] に譲る.

多重結合演算の各プローブフェーズにおいて生成される中間リレーションの大きさは各々の結合演算の結合率に依存する. Left-deep 多重結合演算では途中で結合率が大きくなるような問合せでは中間リレーションがステージングメモリに収まらなくなる場合が起こり得る. この際, 中間リレーションをバケット分割しディスク上に中間ファイルを生成することになるため処理性能が大幅に低下する. Right-deep 多重結合演算では複数のプローブ処理をメモリ上で同時に実行し, 中間結果をステージングメモリ上に展開する必要がないためこのような問題は生じない. すなわち, Right-deep 多重結合演算は Left-deep 多重結合演算等に比べ処理性能の向上が望めるが, その代わりに複数のプローブ処理をメモリ上で連続して行なうため結合演算処理の負荷が高く, またシステムの制御も複雑なものとなる.

2.4.1 Left-deep 多重結合演算

Left-deep 多重結合演算は単一結合演算を順次実行する方式であるが, 各々のプローブフェーズにおいて次段のハッシュテーブルの大きさが既知であり, かつ主記憶に収まる場合には, プローブ結果を中間ファイルに出力する代わりに直接次段のハッシュテーブルに挿入することで I/O コストの削減が可能である. このような場合, Left-deep 多重結合演算では以下のように処理が進行する. 入力リレーションを $R_1, \cdots, R_n$ とする.

1. リレーション R_1 をディスクから読出しながら, 結合属性値にハッシュ関数 h_1 を適用してハッシュ値を求める. ハッシュ値から当該タプルの処理を行うノードを決定し, ネットワークを通じて送出する. タプルを受け取ったノードではハッシュ値に基づいてハッシュテーブル H_1 を生成する.

   $i = 2, \cdots, n$ に対して以下の処理を繰り返す.

2. リレーション R_i をディスクから読み出しながら, 結合属性値にハッシュ関数 h_{i-1} を適用してハッシュ値を求め, 対応するノードに送出する. 受信したタプルに対して結合演算を施すべき R_{i-1} のタプルをハッシュテーブル H_{i-1} から探索し, タプルの結合を行う.

3. $i = n$ の場合は生成したタプルをディスクに書き出し, 処理を終了する.

   それ以外の場合は生成したタプルにハッシュ関数 h_i を施してハッシュ値を求め, 対応するノードに送出する. タプルを受け取ったノードではハッシュ値に基づいてハッシュテーブル H_i を生成する.

   リレーション R_i の全てのタプルを処理し終えたら, ハッシュテーブル H_{i-1} に使われていたメモリを解放する.

Left-deep 多重結合演算は処理方式が単一結合演算に類似しているため実装は比較的容易であり, またプロセッサに対する負荷もプローブ処理の際に並行してハッシュテーブルを生成する処理に相当する分が増加するのみである. また, ハッシュテーブルも同時にはたかだか 2 つしか必要としないため, メモリに対する制約もあまり厳しくない. しかし, 多重結合演算の各プローブフェーズにおいて生成される中間リレーションの大きさは各々の結合演算の結合率に依存し, 一般には予測不可能であるため, 中間のプローブフェーズで生成するハッシュテーブルがステージングメモリ上に収まらない場合が起こり得る. このような場合, 中間ファイルを生成し, 独立した複数の結合演算として実行することになるため I/O 回数の増加とそれに伴う処理時間の増大を引き起こすこととなる.

2.4.2 Right-deep 多重結合演算

Right-deep 多重結合演算はあらかじめ複数のハッシュテーブルを生成し，これらのハッシュテーブルに対する複数のプローブ処理をパイプライン的に実行するものである．Right-deep 多重結合演算では中間結果全体をメモリに展開しないため，Left-deep 多重結合演算と異なり，中間のプローブフェーズにおける結合結果が増大しても中間ファイルを作成する必要が無い．その一方で，すべてのリレーションがメモリ上にロードできるとは限らないため，そのような場合は多重結合演算を複数のセグメントに分割して実行する必要がある．この際にどのような単位で，あるいはどのプローブを境界にしてセグメント分割を行うかは Right-deep 多重結合演算の最適化において極めて重要な問題であり，処理性能に与える影響も大きい．この様な多重結合演算における最適化に関しては現在様々な研究が活発に行われており，例えば文献 [6] などがある．以降ではスケジューリングが既に終了しているものとし，プローブ用リレーションを除くすべてのリレーションがメモリにハッシュテーブルとしてロード可能であると仮定する．セグメント分割が必要な場合は分割後の個々のセグメントを対象とするものとする．

入力リレーション $R_1, \cdots, R_n$ に対する Right-deep 多重結合演算は次のように処理が進行する．

1. $i = 2, \cdots, n$ に対して順次 R_i を読み出し，ハッシュ関数 h_{i-1} を適用してハッシュ値を求め，対応するノードに送出する．受信したタプルに対してはハッシュ値に基づいてハッシュテーブル H_{i-1} を作成する．

2. R_1 の各タプルにハッシュ関数 h_1 を適用してハッシュ値を求め，対応するノードに送出する．

   $i = 1, \cdots, n-2$ に対して以下を並行して処理する．

   ハッシュ関数 h_i に対応する受信タプルに対してハッシュテーブル H_i をプローブし，タプルを結合する．生成されたタプルにハッシュ関数 h_{i+1} を適用してハッシュ値を求め，対応するノードに送出する．

   ハッシュ関数 h_{n-1} に対応する受信タプルに対してハッシュテーブル H_{n-1} をプローブし，タプルを結合した結果をディスクに書き出す．

Right-deep 多重結合演算ではプローブフェーズで複数のプローブ処理を実行しなければならない．これは個々のタプルについて見るとシーケンシャルな処理であるが，処理全体を見ると複数のプローブ処理が並行して実行されることになる．したがって，複数のプローブ

処理を同時処理することによるプロセッサ負荷はLeft-deep 多重結合演算に比較して大きなものとなる．また，リレーションの数に応じて必要とするハッシュテーブルも増加するのでメモリに対する制約も比較的厳しい．

その一方で，Left-deep 多重結合演算と異なり，多重結合演算の中間リレーションをステージングメモリに展開せずに直接次のステージのプローブ処理に使用する為，中間リレーションがステージングメモリから溢れるようなことはない．

第 3 章

並列関係問合せ処理系 DBKernel の設計

3.1 並列問合せ実行モデル

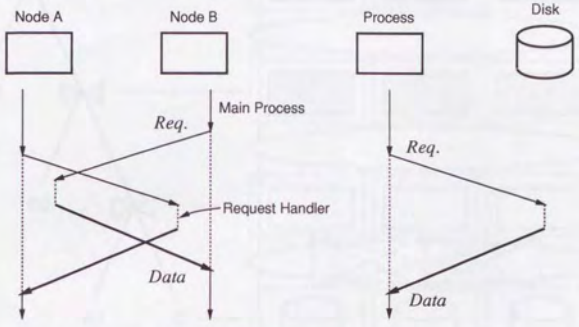
並列関係問合せ処理においては, 関係データベース処理に内在するデータ並列性を最大限に抽出することと大量のデータ流を効率良く処理することが重要であり, SDC システムソフトウェアの初期の実装においてはこれらを達成することが大きな目標とされた. しかし, 結果として出来上がったシステムは, SDC のアーキテクチャやハッシュ結合アルゴリズムに強く依存したものになってしまい, 異なるアーキテクチャへの移植やタイプの異なる演算を実装するには大きな困難が伴うことが明らかになった. そこで, DBKernel の設計に当たっては, 高い性能を維持しつつ, 可搬性や演算の柔軟性を得るため, SDC システムソフトウェアの本質的な特徴を抽出し, 基本的な実行モデルとした. その基本実行モデルとは, 以下に述べるデータ駆動モデルと Function Shipping モデルである.

データ駆動モデル

DBKernel のデータ駆動モデルに対し, 通常の OS 上で実行されるデータ処理のモデルをここではプロセス駆動モデルと呼ぶ. プロセス駆動モデルにおいては, プロセスはコントロールフローに従って入力プリミティブを呼び出すことでアクセス要求を発行し, データの到着を待つ. データが到着するとプロセスは再開され, データの処理を行う (図 3.1). そのため, 処理ノード間のリモートデータ転送には要求 / 応答というメッセージが往復することになり, ネットワークの通信遅延が性能に大きく影響を及ぼす. また, ディスクに対するアクセスにおいてはブロックを指定してデータを読み書きするためランダムアクセスが発生しやすい.

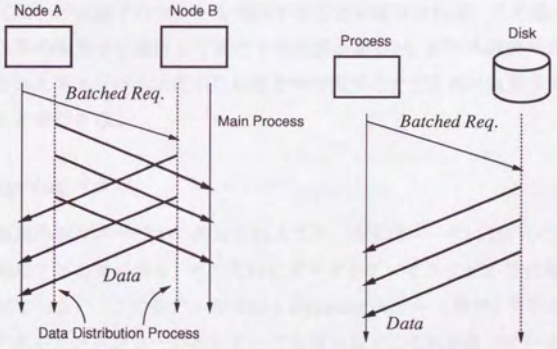
これに対して, データ駆動モデルではデータフローが主であり, プロセスが最初に要求を一括して発行した後は, データ到着イベントへの応答としてデータが処理される (図 3.2). 処理ノード間のリモートデータ転送の際はデータの内容に応じて割り当てノードを決定するデータ分配プロセスを用いることで要求 / 応答という閉ループをなくすことができる. そのため, ネットワークの通信遅延には大きな影響を受けず, バンド幅だけが要求されることになる. また, ディスクに対しても, 物理的なブロック配置に応じたアクセスが可能であり, ランダムアクセスによるスループットの低下を防ぐことができる. プロセス駆動モデルとデータ駆動モデルの中間として非同期 I/O が存在するが, DBKernel ではより抽象度の高いデータ駆動モデルを採用することにした.

実際の問合せ処理においては, 図 3.3 のように複数の演算子をパイプライン的に処理する必要がある. データ並列性を最大限に抽出するためには, 演算子内の並列化, すなわち全ての



(a) リモートデータアクセス　(b) ディスクアクセス

図 3.1: プロセス駆動モデル



(a) リモートデータアクセス　(b) ディスクアクセス

図 3.2: データ駆動モデル

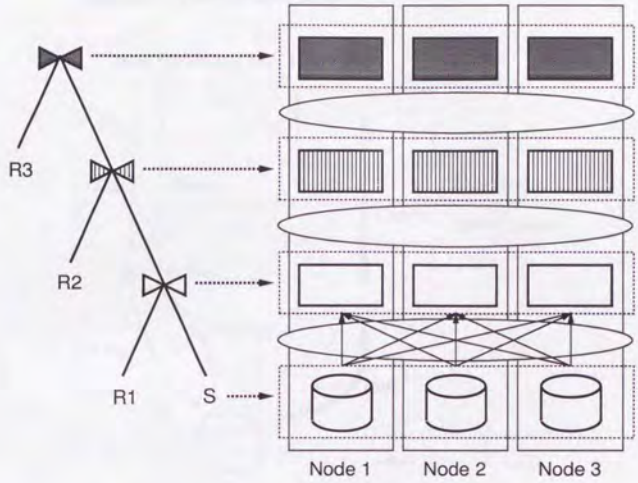


図 3.3: 多重結合演算におけるパイプライン処理

ノードにおいて同一の演算子のセットを実行することが要求される. この場合, 単一ノード内においても複数の演算子を並行して実行する必要があるが, データ駆動モデルにおいてはデータが入力されたストリームに応じた処理を呼び出すことで自然に演算子間のスケジューリングを行うことができる.

Function Shipping モデル

問合せ毎の処理内容はユーザから指定されるため, 通常はユーザに近いシステムの上位階層でデータを処理する必要があり, そのためにデータをデバイスに近い下位層から上位層に移動しなければならない. このモデルを Data shipping モデルと呼び, クライアント / サーバ環境では, クライアントがサーバからデータを受け取り, それ自身でデータを処理することに相当する. このモデルでは, 転送データ量に対して処理内容が単純な場合, データ転送のコストがボトルネックになってしまうという問題がある.

しかし, データ駆動モデルでは入力プリミティブが不要であり, その上, 問合せで実行される演算子のほとんどは単純であるため, データ処理の内容は数行の小さな関数として簡潔に記述することができる. そこで, データの代わりにプログラムを上位層から下位層に移動

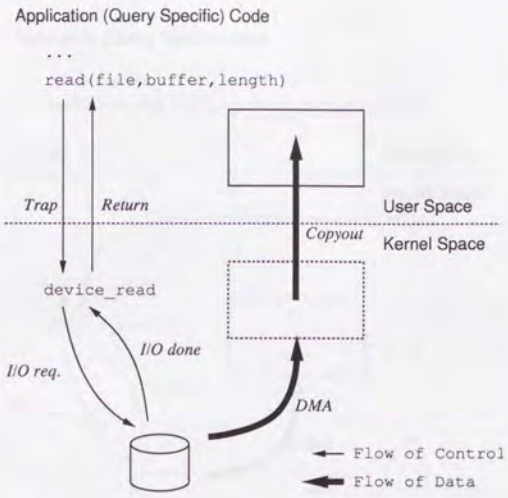


図 3.4: Data shipping モデルに基づくデータ処理

し, データを低い階層で処理することで上記の問題を解決することができる. このモデルを Function shipping モデルと呼び, クライアント / サーバ環境においてはクライアントが処理の内容をサーバに通知してサーバにデータ処理を実行させ, 結果だけを受け取る方式に相当する.

図 3.4と図 3.5にそれぞれ Data shipping モデルと Function shipping モデルに基づくデータ処理の概念図を, OS のシステムコールの階層を例に用いて示す. ユーザ空間へ直接 DMA を行うことで, Data shipping におけるデータのコピーを減らすことは可能であるが, コントロールフローがシステム階層を横切るのを防ぐことはできない. Function shipping によればデータフローだけでなくコントロールフローもカーネル空間で閉じるという利点がある.

Function shipping を実現するためには, 下位の階層に処理の内容を通知する方法を定義しなければならない. 最も単純な方法は, あらかじめ定義した操作を用意しておき, その名前だけを指定する方法である. これは初期の SDC において取られていた方法と本質的に等しいが, 処理の柔軟性の面で難がある. そこで, 処理内容自体を記述するだけの記述力のあるものとして, インタープリタ用の中間言語やプロセッサのネイティブコードが考えられるが, ここでは後者を採用することにした. 中間言語を用いる方式はセキュリティの面で強固であ

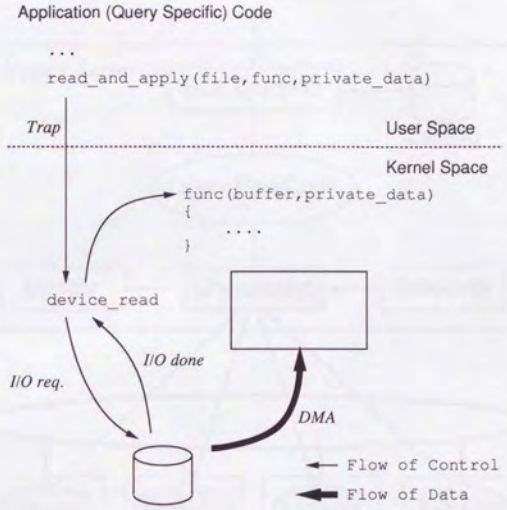


図 3.5: Function shipping モデルに基づくデータ処理

るという利点があるが, インタープリタや実行時コンパイラなどを整備する必要がある. プロセッサのネイティブコードを渡す方式は, ダイナミックリンクライブラリを備える最近の OS では容易に実現可能であり, また実行内容の自由度も高い.

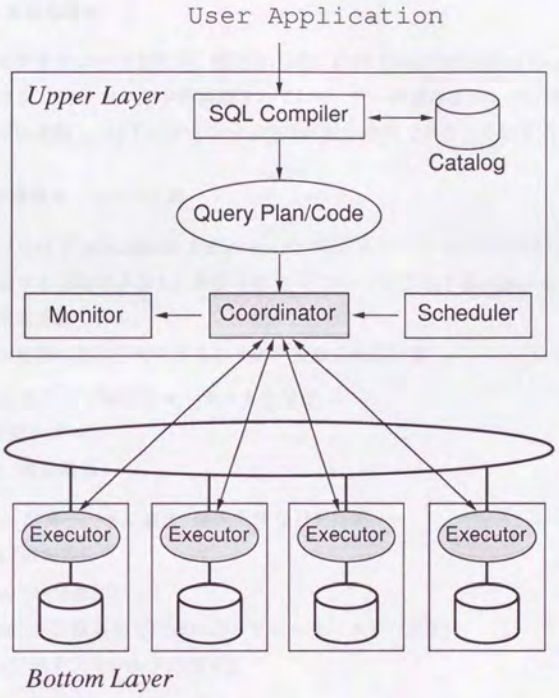


図 3.6: DBKernel の全体構成

3.2 DBKernel 上位層の構成

DBKernel は全体として図 3.6のような構成を採る．ユーザはシステムに SQL 等の非手続き型言語を用いて問合せを発行するため，サーバのアーキテクチャや並列性については意識する必要がない．SQL 問合せはデータベース定義情報を格納するシステムカタログを参照してコンパイルされ，さらに実行プランの最適化とコード生成が行われる．そして，スケジューラからの指示に基づいて実行コードが全ノードにブロードキャストされ，問合せの並列実行が行われる (SPMD 方式)．以下に DBKernel 上位層の構成要素を示す．

3.2.1 SQL 言語処理系

ユーザとのインタフェースを司る. 現在は, SQL の内 Data Definition Language の一部のみを受け付ける, ddlc コマンドが実装されている. データ定義をコンパイルした結果をシステムカタログに登録し, 以下に示すコード生成の際に参照できるようにする.

3.2.2 問合せ最適化・コード生成

現時点では, Data Manipulation Language の SQL インタフェースがないため, 最適化済のプランを記述する言語を入力し, 実行可能な C コードを生成する, qplanc (Query Plan Compiler) が用意されている.

問合せプラン言語には以下のプリミティブが用意されている.

- 宣言子 — 出力タプルのフォーマットを指定.
  @tmp (中間タプル),
  @result (最終結果)
- 修飾子 — 演算子の後に置き, 選択条件などを指定.
  @select (選択条件),
  @hashcmp (結合条件),
  @hashupd (集計演算などにおけるハッシュエントリの更新),
  @order (ソートフィールドの指定),
  など.
- 演算子
  @repart (データの再分配),
  @build (ハッシュテーブルへの挿入),
  @build1 (重複検査を行うハッシュテーブルへの挿入),
  @probe_repart (ハッシュテーブルの検索, 結果の再分配),
  @write (ファイルへの書き込み),
  @unbuild (ハッシュテーブルをファイルに変換),
  など.

3.2.3 問合せ実行制御

dbkc (DBKernel Coordinator) なるコマンドが用意されており, 複数ノードの実行制御や状態監視をはじめ, バックエンドの処理ノードとの全てのインタフェースを司る.

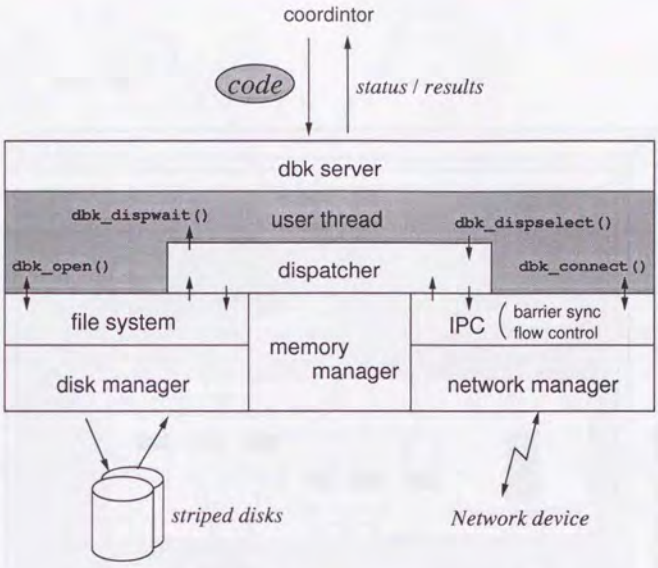


図 3.7: DBKernel 下位層のソフトウェアアーキテクチャ

3.2.4 問合せ実行モニタ

`dbkview` (DBKernel Viewer) なるコマンドが用意されており, 任意ノードの実行時の状態監視を GUI ベースで行う.

3.3 DBKernel 下位層の構成

DBKernel のバックエンド実行系のソフトウェアアーキテクチャを図 3.7に示す. 最大限の効率を達成するためとプラットフォーム依存性を吸収するために, ファイルシステム, プロセス (スレッド) 間通信, メモリ管理などの独自のモジュール群を用意している. ファイルシステムを用意することにより, ディスクに raw device としてアクセスすることができ, OS のバッファリングやディスクブロック割当ての非効率性を回避することが可能となった. また, プロセス間通信のレイヤは従来の OS の通信プリミティブが一対一あるいは一対多通信を仮定しており, 多対多通信の際には繁雑なコードを書く必要があるのを緩和し, implicit な

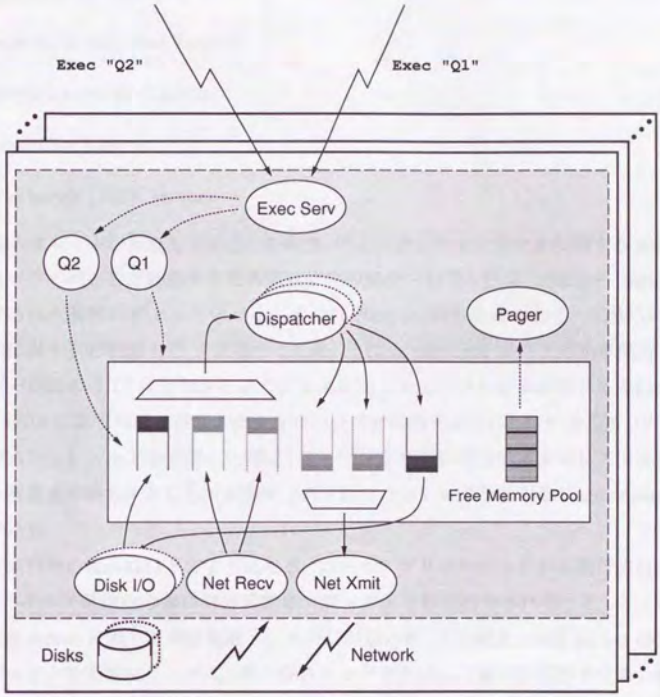


図 3.8: DBKernel 下位層のスレッド構成

バリア同期やフローコントロールを提供する．さらに，メモリ管理ルーチンは実メモリを意識したメモリ割当を行い，OS のページングによる影響を排除する．Coordinator から送られて来た実行コードは dbk_ プレフィクスを持つルーチンから成るこれらのモジュール群を呼び出し，OS に対してシステムコールを直接発行することはない．したがって，プラットフォーム依存性を Disk manager や Network manager の部分で吸収することで実行コードの可搬性を得ることが出来る．

図 3.8に DBKernel のスレッド構成を示す．

1. Pager daemon

2. Disk manager daemon
3. Network transmitter daemon
4. Network receiver daemon
5. Dispatcher daemon
6. Exec server (DBK server)

の 6 種類のスレッドが常駐している. この内, ディスクとネットワークに関する daemon はデバイスドライバとして機能するため実装依存の部分である. Disk manager daemon はストライプされた複数のディスクデバイスのそれぞれについて生成される. DBKernel ではディスクに対する非同期 I/O を実現するために, Disk manager がファイル単位のアクセス要求をページ毎の I/O に変換するようにしている. 一方, ファイルの書き込みにおいては delayed write により可能な限りメモリ上のバッファに残すようにしているため, フリーメモリの量がスレッショルドを下回った時に, オープンされているファイルのリストを走査してバッファ消費量が最も大きなものからディスクにフラッシュする処理を Pager daemon が受け持っている.

問合せ実行時の流れは以下のようになる. ユーザアプリケーションから発行された問合せは, コンパイル・最適化の後に実行可能オブジェクトとして出力される. そして, 各ノード上の DBK server に対して実行要求メッセージが送られ, その結果 DBK server は指定されたオブジェクトを動的にリンクし, 新たなスレッドを生成して実行を開始させる. 各ノード上のスレッドからの出力は DBK server を経由し実行結果としてユーザアプリケーションに返される. 問合せ毎に新たに生成されるスレッドは, 入出力をオープンし, 入力データを処理するコールバックルーチンを設定した後は休止状態となる. 実際にデータを処理するのは Dispatcher であり, あらゆる入力イベントをこのスレッドが扱う. Dispatcher はマルチプロセッサの処理ノード上では複数のインスタンスが生成される.

3.4 DBKernel 上でのデータ駆動並列プログラミング

上述のように, DBKernel 上で実行されるプログラムは UNIX などの OS 上で動作するプログラムとはかなり異なったものになる. ここでは, 並列ハッシュ結合演算のスプリットフェーズに似た処理を行うプログラム `split` を例に用いて説明する.

33〜34ページに簡略化したプログラムのソースコードを示す. このソースコードを含むファイルは C コンパイラによりコンパイルされるが, その時に位置独立なオブジェクトコー

ドを生成するオプションを指定する．リンケージエディタにより出力されたオブジェクトファイルはそれ自体では実行されない．その代わりに，Coordinator の働きをするプログラムによって複数のノードの DBKernel に対して送付される．この要求を User request server が受け付けると新たなスレッドを生成し，そのオブジェクトモジュールのエントリポイントから実行が開始されるように設定する．

このプログラムのエントリポイントは第 14 行から始まる関数 split() である．第 23 行では，関数 dbk_getpeercnt() によってこの split プログラムが同時に幾つのノード上で実行されているかを調べている．これはシステム全体のノード数より小さいこともあり，Coordinator に与えるオプションでどのノードの集合 (peerset) に対してプログラムを並列実行させるかを指定する．

第 25 行から第 27 行では，それぞれ入力ファイル，通信チャネル，出力ファイルをオープンしている．ファイルを扱う関数 dbk_open() は UNIX のシステムコール open() のセマンティクスを踏襲しているが，通信チャネルをオープンする dbk_connect() は UNIX の socket() や connect() とは大きく異なる．ここでの引数の意味は，全ての “peerset” 間で論理ノード番号をアドレスとして通信するということであり，最も頻繁に使われるものである．この関数のリターン時には，通信に参加する全てのノード間でバリア同期が取られたことが保証される．

第 29 行から第 32 行では，ローカル変数の値が構造体のフィールドに代入されている．これは，ファイル入力に対するコールバック関数への引数となる．第 34 行で入力プライオリティの最大値 (最低優先度) を取得し，第 35 行でファイル入力 infd に対してコールバック関数 split_proc() とファイナライズ関数 split_term() が設定され，またコールバック関数への引数が指定されている．この結果，Dispatcher daemon により infd に対応するファイルの読み出しが開始され，その各タプルに対してコールバック関数 split_proc() が呼び出される．

続く第 38 行から第 40 行では，1 つ高いプライオリティでネットワーク入力 sockfd に対するコールバック関数 store_proc() とファイナライズ関数 store_term() が設定され，コールバック関数への引数として出力ファイルの識別子 outfd が指定されている．

その後，このスレッドは dbk_dispwait() を呼び出して Dispatcher から End-Of-File イベントが通知されるのを待つ．全ての入力を処理し終えて dbk_dispwait() が帰ると dbk_fsync() によりバッファリングされていたブロックをディスクに書き出して終了する．

第 50 行から第 55 行までは，入力ファイルに対するコールバック関数 split_proc() である．引数 data と size はそれぞれ入力タプルのメモリ上でのアドレスとバイト長を示す．

タプルの先頭から指定されたオフセットにある整数属性をキーと見做し, その論理ノード数に対する剰余を宛先論理ノードアドレスとしてタプルを転送する.

第 57 行から第 60 行までのファイナライズ関数 split_term() はネットワーク出力に対し dbk_shutdown() を呼ぶことで End-Of-File イベントを通知する.

第 62 行から第 66 行までのネットワーク入力に対するコールバック関数 store_proc() は受け取ったタプルを出力ファイルに書き出している. ファイナライズ関数 store_term() は何もする必要がないので本体は空である.

以上のように, プログラムに対応するスレッドは初期設定をした後はスリープ状態であり, 実際の処理は Dispatcher のコンテクスト内でコールバック関数を呼び出すことにより行われる. コールバック関数には一つの入力タプルに対する処理を記述すればよく, どのコールバック関数をどのような順序で呼び出すかは全て DBKernel の内部で処理される. したがって, DBKernel 上では複雑なデータ処理をその本質のみに集中して記述することが可能である.

```
 1 /* split.c - split a file in parallel */
 2 #include <dbkern/types.h>
 3 #include <dbkern/dbkstd.h>
 4
 5 /* defines */
 6 struct split_args {
 7     int outfd;
 8     off_t offset;
 9     uint32_t nnodes;
10     uint32_t nodeid;
11 };
12
13 /* entry */
14 void split (const char *infile, const char *outfile, off_t offset)
15 {
16     uint32_t nnodes;
17     int infd;
18     int sockfd;
19     int outfd;
20     int pri;
21     struct split_args args;
22
23     nnodes = dbk_getpeercnt();
24
25     infd   = dbk_open(infile, O_RDONLY, 0);
26     sockfd = dbk_connect(AT_DEFAULT, 0, NULL, 0);
27     outfd  = dbk_open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
28
29     args.outfd  = sockfd;
30     args.offset = offset;
31     args.nnodes = nnodes;
32     args.nodeid = dbk_getlogid();
33
34     pri = dbk_sysconf(DBK_CF_IOPRI_MAX) - 1;
```

```
35      dbk_dispselect(infd, pri,
36                      split_proc, split_term, &args, sizeof(args));
37
38      --pri;
39      dbk_dispselect(sockfd, pri,
40                      store_proc, store_term, (void *) outfd, 0);
41
42      dbk_dispwait();
43
44      dbk_fsync(outfd);
45      dbk_exit(0);
46  }
47
48  /* callback functions */
49
50  static int split_proc(caddr_t data, size_t size, struct split_args *ap)
51  {
52      uint32_t key = *(uint32_t *) (data + ap->offset);
53      uint32_t dest = key % ap->nnodes;
54      return (dbk_send(ap->outfd, data, size, dest));
55  }
56
57  static void split_term(struct split_args *ap)
58  {
59      dbk_shutdown(ap->outfd);
60  }
61
62  static int store_proc(caddr_t data, size_t size, void *ap)
63  {
64      int outfd = (int) ap;
65      return (dbk_write(outfd, data, size));
66  }
67
68  static void store_term (void) {}
```

第 4 章

SDC-II における DBKernel の実装と性能評価

4.1 SDC-II のハードウェアアーキテクチャ

SDC-II 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スケーラビリティを, 処理ノードを全共有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実現している. また, I/O 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 I/O 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.

SDC-II のハードウェアアーキテクチャを図 4.1 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で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持つオメガネットワークを独自に開発した.

データ処理モジュールは以下のような構成要素から成る.

Data Processor (DP) データベース処理を実行するプロセッサで各 DPM 当たり最大で 6 台の DP が実装される. 各 DP は,

- MC68040 25MHz マイクロプロセッサ
- 2MB ローカルメモリ
- 共有バスインタフェース
- MC68901 MFP (タイマ / シリアルインタフェース)
- ロケーションモニタ (プロセッサ間割り込み要求用)

を備えている.

Intelligent Disk Controller (DC) 2 本の SCSI バスインタフェースと, それぞれの DMA チャネルおよび FIFO を備え, デバイスからの割り込み処理を専用に行う制御用プロセッサを持つ. 各 DPM 当たり 2 台の DC が実装される. リレーションは 4 台のディスクにタプル単位でストライプ化して格納される. 以下のものを備える.

- MC68340 16MHz マイクロコントローラ
- 1MB ローカルメモリ

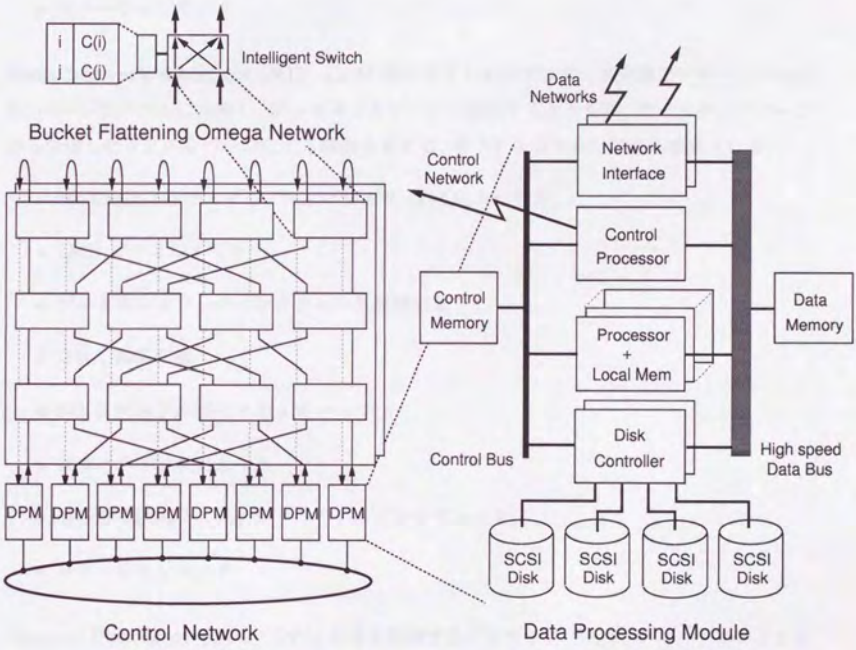


図 4.1: SDC-II のハードウェアアーキテクチャ

- WD33C93B SCSI バスインタフェース × 2
- DMA 制御回路
- 32KB デュアルポート RAM バッファ
- 共有バスインタフェース
- ロケーションモニタ

Data Network Interface (NI) DPM 間のタプル単位でのデータ交換をサポートするため, ページをタプルに分解し, データネットワークに送出するとともに, データネットワークから受信したタプルをページ化する機能を有する. 各 NI は以下のものから構成される.

- MC68020 25MHz マイクロプロセッサ (割り込み処理用)
- 1MB ローカルメモリ
- データネットワークインタフェース制御回路
- DMA 制御回路
- 64KB デュアルポート RAM バッファ
- 共有バスインタフェース
- MC68901 MFP (タイマ / シリアルインタフェース)
- ロケーションモニタ

Control Processor (CP) DPM 全体を制御するプロセッサであり, フロントエンドとの通信を司る. 以下の構成要素が含まれる.

- MC68040 25MHz マイクロプロセッサ
- 4MB ローカルメモリ
- 256KB ブート ROM
- 共有バスインタフェース
- Ethernet インタフェース

- MK68564 シリアルインタフェース
- MK48T87 リアルタイムクロック
- ロケーションモニタ

Data Memory (DM) データ処理用ステージングメモリであり, 32 MB DRAM で構成される. 各プロセッサから DM へのアクセス時間を短縮するために 2 バンク構成を採る.

Control Memory (CM) プロセッサ間の同期・通信のためのメモリであり, 2MB SRAM から構成される.

High Speed Data Bus (HBus) 各プロセッサや DMA 制御回路から DM にアクセスするための共有バス. 16 バイトのバースト転送をサポートしてデータ転送を高速化しているが, バスプロトコルを単純化するためアクセスサイズを 32 bit に限り, read-modify-write サイクルもサポートしていない.

Control Bus (CBus) CM へのアクセスや, 他のプロセッサのローカルメモリ, ロケーションモニタへのアクセスを通じて DPM 内の同期・通信を行うための共有バス. Hbus と異なり, 8, 16, 32 bit 幅のバスアクセスと read-modify-write サイクルをサポートする.

SCSI Disk Fujitsu M2624S (550 MBytes, 2.4 MB/sec) または Fujitsu M2934S (4153 MBytes, 8.6 MB/sec).

4.2 SDC-II における DBKernel の実装

SDC-II では, リアルタイムカーネルである VxWorks を使用しており, DBKernel もカーネルレベルで実装した. 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, ノード内のプロセッサ間の同期プリミティブ (mutex, condition variable) などが備わっていないため, 実装する必要があった.

また, ディスクおよびネットワークに専用の管理 CPU を用いているため, I/O 管理デーモンは直接 I/O 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

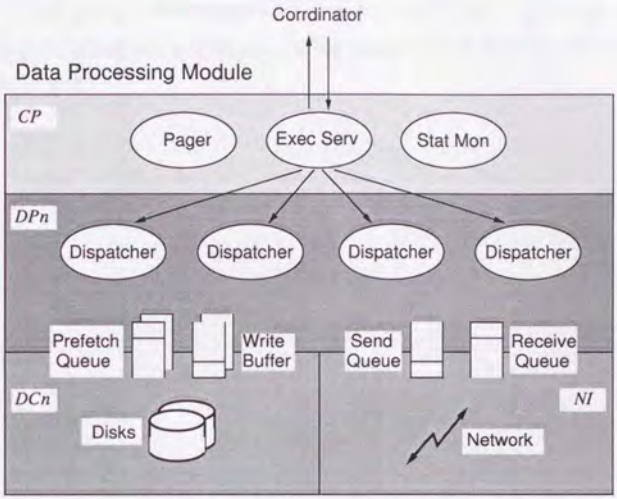


図 4.2: SDC-II のソフトウェア構成

するコマンド発行の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．

サービススレッド (DBK server) は CNet (イーサネット) インタフェースを持つ CP 上で実行される．また，ディスパッチャは各 DP 上で実行されるため複数存在することになる．複数プロセッサからの要求に基づいて処理を割り付けるため，自律的な負荷分散を実現することが可能である．また，DP はディスパッチャ以外の処理を行う必要がないため，入力データの到着をビジーウェイトにより CPU を占有したまま待つことができ，高速な応答性を実現している．

共有メモリが DM と CM とに分離されており，前者が I/O バッファ及びハッシュテーブルのページ用，後者がその他の構造体用として使われるため，メモリ管理ルーチンでは，目的に応じてどちらかの領域のメモリを割り当てるようにしている．

図 4.2に SDC-II 上に実装された DBKernel の構成を示す．CP 上のサービススレッドが受け付けて生成されたユーザスレッドは I/O をオープンして Callback を登録するが，その実行は異なるプロセッサである DP 上で行われる．SDC-II は SMP (Symmetrical Multi Processor) ではないため，プロセッサ毎のローカルアドレス空間は独立であり，関数へのポ

インタをプロセッサ間で直接やりとりすることはできない. そのため, 実行ファイル名と関数のシンボル名の対を受渡しし, DP 側で改めてファイルのダイナミックリンクを行うようにしている.

4.3 TPC-D ベンチマークの概要

TPC-D ベンチマーク [34] は, 大量データをアクセスし, 複雑度の高い問合せが実行される意思決定支援システムを対象としたベンチマークであり, 8 つのリレーションに対して 17 の問合せと 2 つの更新が発行されるようになっている. 図 4.3に TPC-D ベンチマークのリレーションの定義と相互参照関係を示す. ただし, 図中の SF はデータベースサイズのスケールファクタであり, SF = 1 の時の全体のサイズはおよそ 1GB である. 17 の問合せの内容は以下に示すようにビジネス指向のものとなっている.

Q1 Pricing summary report query

特定の期日までに出荷された全ての品目の価格に関して出荷状況別にまとめたレポートを出力する.

出力例:

```
l_returnflag l_linestatus           sum_qty    sum_base_price ...
------------ ------------ ----------------- ----------------- ...
           A            F        3773034.00     5319329289.68 ...
           N            F         100245.00      141459686.10 ...
           N            O        7464940.00    10518546073.98 ...
           R            F        3779140.00     5328886172.99 ...
```

Q2 Minimum cost supplier query

特定の地域で特定の部品を最も低価格で提供している供給元に関する情報の一覧を出力.

出力例:

```
s_acctbal            s_name         n_name   p_partkey         p_mfgr ...
--------- ------------------ -------------- ----------- -------------- ...
  9828.21 Supplier#000000647 UNITED KINGDOM       13120 Manufacturer#5 ...
  9508.37 Supplier#000000070 FRANCE                3563 Manufacturer#1 ...
...
  -845.44 Supplier#000000704 ROMANIA               9926 Manufacturer#5 ...
  -942.73 Supplier#000000563 GERMANY               5797 Manufacturer#1 ...
```

Q3 Shipping priority query

未出荷の注文の内, 収入額の大きいものの一覧を出力.

出力例:

l_orderkey	revenue	o_orderdate	o_shippriority
260930	320547.25	1995-03-12	0
402497	298879.53	1995-02-12	0
...			
97830	273227.06	1995-03-04	0
90276	272233.92	1995-03-04	0

Q4 Order priority checking query

特定の四半期につき, 納入が間に合わなかった注文の数を注文に対する優先順位別に出力.

出力例:

o_orderpriority	order_count
1-URGENT	999
2-HIGH	1002
3-MEDIUM	1021
4-NOT SPECIFIED	997
5-LOW	1089

Q5 Local supplier volume query

特定地域内の各国別に, 顧客と供給元が同じ国に属する注文に対する収入を求める.

出力例:

n_name	revenue
CHINA	7349391.47
INDONESIA	6485853.40

```
INDIA                          5505346.82
JAPAN                          5388883.59
VIETNAM                        4728846.60
```

Q6 Forecasting revenue change query

特定の年に出荷された品目の内, 割引率が一定の範囲にあり, 出荷量が少ないものについて割引の収入に対する影響を求める.

出力例:

```
        revenue
-----------------
     11450588.04
```

Q7 Volume shipping query

特定の国の間の取り引きによる収入を年別に出力.

出力例:

```
supp_nation cust_nation        year          revenue
----------- ----------- ----------- -----------------
FRANCE      GERMANY            1995        4611421.44
FRANCE      GERMANY            1996        4828420.37
GERMANY     FRANCE             1995        6755766.84
GERMANY     FRANCE             1996        5810951.40
```

Q8 National market share query

特定の品目に対する収入について, 特定の地域におけるある国のシェアの年毎の変化を求める.

出力例:

```
      year          mkt_share
----------- -----------------
      1995                .05
      1996                .08
```

Q9 Product type profit measure query

特定の部品に対する利益を注文のあった国と年別に求める.

出力例:

```
                    nation        year        sum_profit
------------------------- ----------- -----------------
ALGERIA                               1998        1946316.01
ALGERIA                               1997        2973825.69
...
VIETNAM                               1993        3764237.18
VIETNAM                               1992        3420922.00
```

Q10 Returned item query

一定期間に注文された部品につき, 返品による損失の大きい顧客の一覧を出力.

出力例:

```
 c_custkey             c_name      revenue    c_acctbal n_name          ...
---------- ------------------ ------------ ------------ --------------- ...
      9722 Customer#000009722    464618.26       474.04 CANADA          ...
     12800 Customer#000012800    444265.64      1900.84 PERU            ...
...
       554 Customer#000000554    373004.47      8395.57 BRAZIL          ...
     13126 Customer#000013126    371722.00      6172.91 INDIA           ...
```

Q11 Important stock identification query

特定の国について, ある部品の一定の割合以上のストックを保持している供給元の一覧を出力.

出力例:

```
 ps_partkey             value
----------- -----------------
      12098       16227681.21
       5134       15709338.52
```

```
...
      5043       10226395.88
     12969       10125777.93
```

Q12 Shipping modes and order priority query

納入が間に合わなかった品目の個数を船便 / 航空便の別に集計.

出力例:

```
l_shipmode high_line_count low_line_count
---------- --------------- --------------
MAIL                   654            950
SHIP                   684           1004
```

Q13 Sales clerk performance query

特定の販売員が受注した注文の内, 顧客からの返品による損失を年別に集計.

出力例:

```
      year          revenue
---------- ----------------
      1992       1262855.73
      1993        964121.03
      1994       1750395.29
      1995        198820.30
```

Q14 Promotion effect query

特定の月の収入の内, 販売促進商品による収入のパーセンテージを求める.

出力例:

```
   promo_revenue
----------------
           16.73
```

Q15 Top supplier query

特定の四半期に出荷された品目につき収入に対する寄与が最も大きい供給元を求める.

出力例:

```
 s_suppkey             s_name ... s_phone             total_revenue
---------- ------------------ ... --------------- -----------------
       389 Supplier#000000389 ... 34-885-883-5717        1418538.21
```

Q16 Part/supplier relationship query

特定の部品につき, 一定の条件を満たす供給元の数を求める.

出力例:

```
   p_brand                    p_type      p_size supplier_cnt
---------- ------------------------- ---------- ------------
Brand#14   SMALL ANODIZED NICKEL             45           12
Brand#22   SMALL BURNISHED BRASS             19           12
...
Brand#53   MEDIUM BURNISHED BRASS            49            3
Brand#54   SMALL POLISHED BRASS               9            3
```

Q17 Small-quantity-order revenue query

特定の品目につき, 小口の注文の取扱をやめたときの収入に対する影響を調べる.

出力例:

```
      avg_yearly
-----------------
        24436.88
```

また, 表 4.1には各問合せに含まれる結合演算と集計演算についてまとめた. 多くの結合演算や集計演算を実行する必要がある問合せでは高い処理効率が要求される. 各問合せの SQL 記述は付録 A に示す.

問合せ	結合演算の数	集計演算の種類
1	—	sum, avg, count
2	4	min
3	2	sum
4	1 ('exists')	count
5	5	sum
6	—	sum
7	5	sum
8	7	sum
9	5	sum
10	3	sum
11	2	sum, 副問合せ結果に対する sum
12	1	sum
13	1	sum
14	1	sum
15	1	sum, 副問合せ結果に対する max
16	2 ('not in')	重複除去を伴う count
17	1	sum, 副問合せ結果に対する avg

表 4.1: TPC-D ベンチマーク問合せの複雑度

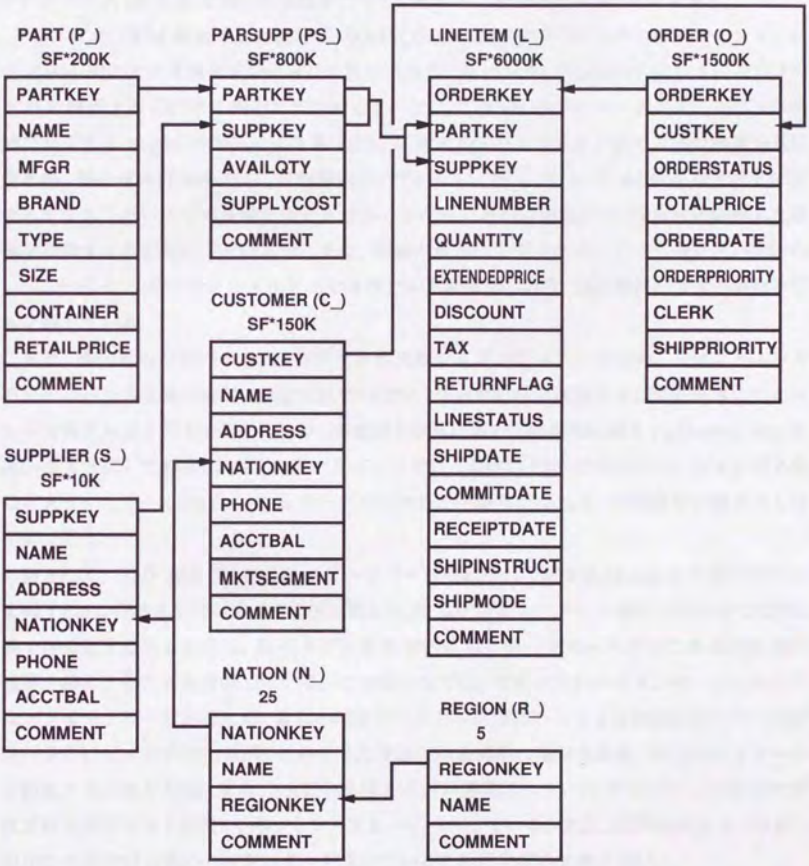


図 4.3: TPC-D ベンチマークのリレーションと参照関係

4.4 TPC-D ベンチマークによる性能評価

TPC-D ベンチマークによる SDC-II の性能評価として, スケールファクタ 1 および 10 のデータベース (表 4.2) を用いた測定を行った. 表 4.3 に問合せ毎の実行時間を示す.

SDC-II の DPM 数は 4 台に固定し, NATION および REGION 以外の各リレーションは各 DPM 間にデクラスタリングされて格納される. NATION, REGION は小さいのでノード 0 に相当する DPM に集中して格納した. なお, TPC-D ベンチマーク用データベース生成プログラム `dbgen` の性質により各 DPM に対するデクラスタリングの方法は範囲分割になるが, 問合せ実行の際にはこの知識は用いていない. SDC-II では, ad-hoc な問合せに対するワーストケースでの性能を向上させることを狙いとしているので, 問合せに依存した最適化は敢えて行わないことにした. また, 同様の理由によりインデックスも全く用いていない. したがって, 全てのファイルアクセスはフルスキャンであり, 選択条件により不要なタプルを捨てている.

また, 問合せのコンパイル, 実行プランの生成は人手で行っているため, フロントエンドのオーバヘッドは実行時間には含まれていない. 問合せの結合演算および集計演算にはハッシュ分割アルゴリズムを用いている. 多重結合演算においては可能な限り right-deep tree を用いるようにしており, 1 GB のデータベースでは LINEITEM 以外のリレーションはそれほど大きくなく, 同時にハッシュテーブルを作ることが可能なため, 中間結果の書き出しは不要である.

例として, 1GB および 10 GB のデータベースそれぞれに対する Query 9 の実行プランを図 4.4および図 4.5 に示す. 木の枝に記した R はタプルをハッシュ値にしたがって DPM 間で再分配することを表し, B はタプルを全 DPM にブロードキャストして各 DPM 毎に複製を持たせることを意味している. この問合せでは, ビルドリレーションのハッシュキーはプライマリキーであるため, 最初のデクラスタリング時にハッシュ分割法を用いていればローカルにビルドが行えるが, このような最適化は意図的に避けたため, 全てネットワークを経由する必要がある. また, NATION はサイズが非常に小さいので, ハッシュ分割をせずにブロードキャストを用い, ネットワークをバイパスしている. また, SUPPLIER と PARTSUPP の間では共通のハッシュキーを用いているため再分配の必要がない.

図 4.4の実行木による結果は, 表 4.3の Query 9 の項の上段に示したものである. 下段のものは, LINEITEM の再分配の代わりに PART のブロードキャストを用いた場合のもので, ネットワークに対する負荷が軽減されるために性能が向上している. 図 4.5でも同様の方法を取っている. この場合, 初めの 3 段の結合演算の後は, バケット分割を伴う left-deep tree になっている.

リレーション	タプル数	タプル長	デクラスタリング
SUPPLIER	10000 × SF	208 bytes	S_Suppkey による範囲分割
PART	200000 × SF	176 bytes	P_Partkey による範囲分割
PARTSUPP	800000 × SF	220 bytes	PS_Partkey による範囲分割
CUSTOMER	150000 × SF	236 bytes	C_Custkey による範囲分割
ORDER	1500000 × SF	140 bytes	O_Orderkey による範囲分割
LINEITEM	6001215 (SF = 1) 59986052 (SF = 10)	156 bytes	L_Orderkey による範囲分割
NATION	25	192 bytes	ノード 0 に集中格納
REGION	5	188 bytes	ノード 0 に集中格納

表 4.2: SDC-II 上の TPC-D テストデータベース

次に, 商用機の性能と比較を行うが, 表 4.3からも分かるように共通する環境がほとんど無いので何らかの正規化を行わなければならない. まず, データベースのサイズが共通だった場合についての比較を図 4.6に示す. 全体的に SDC-II の結果の方が問合せ毎の変動が少なくなっているが, これは常にファイルの全体をスキャンしているのでインデックスの有無による性能の変化が起こらないためと考えられる. インデックスが有効に働く Query 13 や 14 では当然のことながら不利な結果になっているものの, その他の問合せでは処理時間が短くなっている. これは, SDC-II の I/O アクセス性能の高さと多重結合演算の実行効率の高さを示すものと考えられる. 図 4.7は, データベースのサイズに加えて SPECint92 で表した CPU 性能の合計で正規化した場合の図であるが, SDC-II の方が CPU 数が少なく, 古い世代のプロセッサを用いているためさらに有利な結果になっている. データベースのサイズが変わった時に, 性能が線形に変化するとは限らないが, SDC-II 側が初期データ配置やアクセス手段などの点でワーストケースになっていることを考慮すると, この結果は SDC-II の処理効率の高さを示すものと言える.

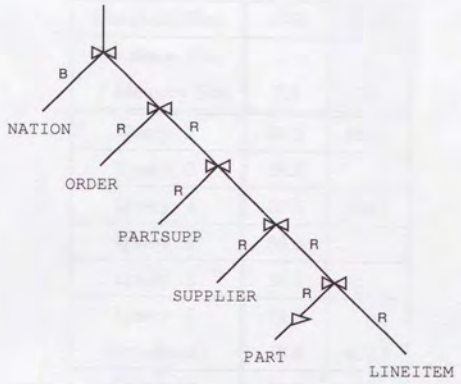


図 4.4: 1GB TPC-D Query 9 の実行プラン

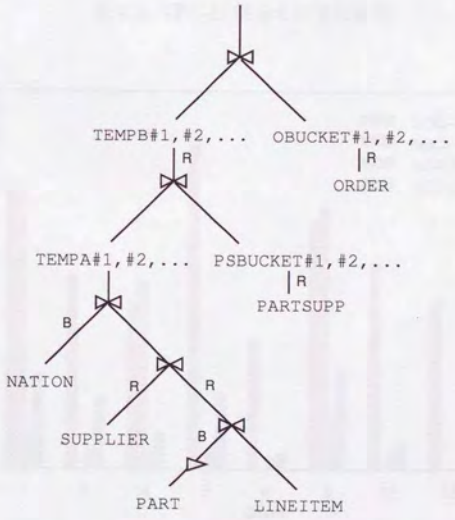


図 4.5: 10GB TPC-D Query 9 の実行プラン

Database Size	1GB	10GB
Database Size / Memory Size	7.8	78
Query 1	53.2	528.5
Query 3	35.6	
Query 4	30.5	346.3
Query 5	59.8	
Query 6	24.0	
Query 9	74.5	
(broadcast)	46.0	482.7
Query 12	38.2	
Query 13	31.1	
Query 14	25.9	

表 4.3: TPC-D 問合せの実行時間

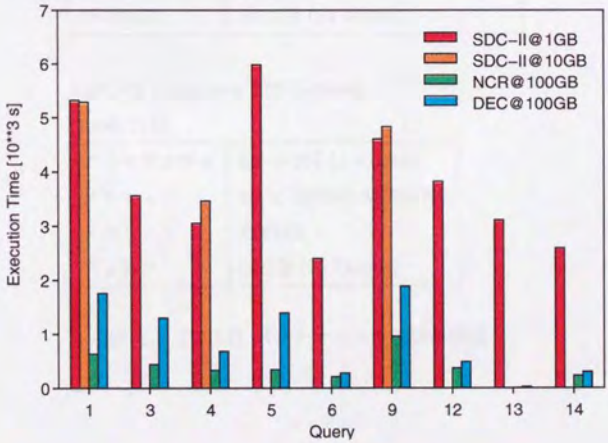


図 4.6: データベースサイズで正規化した実行時間

NCR WorldMark 5100M (running Teradata)
(処理性能 1 位 (1996/6/14))

アーキテクチャ	SE + SN (20 Nodes)
プロセッサ	160 × 133MHz Pentium
メモリ	20GB
ディスク	800GB (400 Drives)

DEC AlphaServer 8400 (running Oracle 7)
(価格性能比 1 位 (1996/11/22))

アーキテクチャ	SE
プロセッサ	12 × 437MHz DECchip 21164
メモリ	24GB
ディスク	361GB (84 Drives)

SDC-II (running DBKernel)
(1996/7/3)

アーキテクチャ	SE + SN (4 Nodes)
プロセッサ	16 × 25MHz MC68040
メモリ	128MB
ディスク	64GB (16 Drives)

表 4.4: TPC-D ベンチマーク性能評価環境

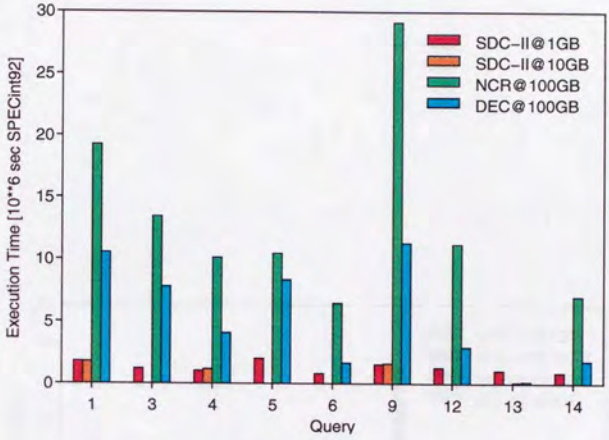


図 4.7: CPU 性能を考慮して正規化した実行時間

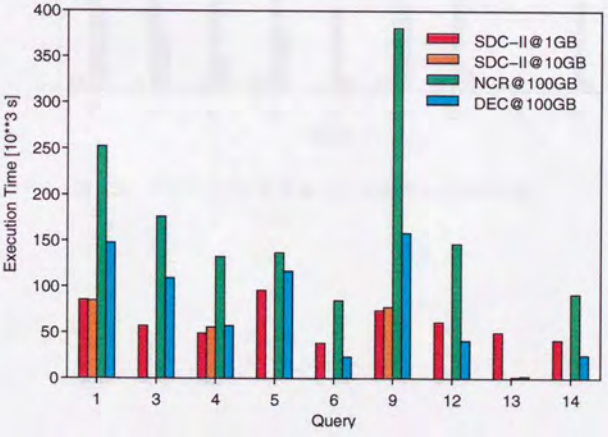


図 4.8: ディスク数を考慮して正規化した実行時間

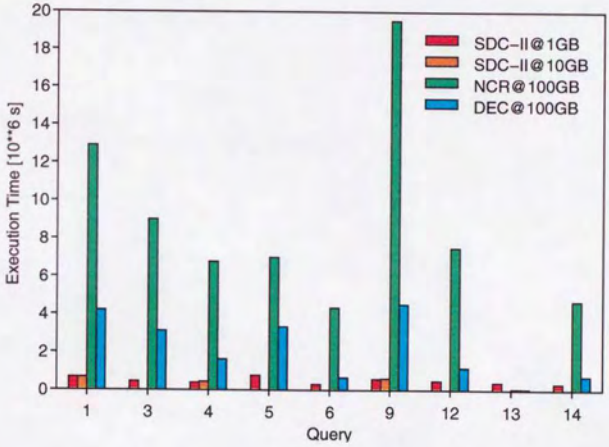


図 4.9: メモリサイズを考慮して正規化した実行時間

第 5 章

大規模 PC クラスタにおける DBKernel の実装と性能評価

5.1 大規模 PC クラスタのシステム構成

今回我々が構築した大規模 PC クラスタは, 図 5.1に示すように 200 MHz Pentium Pro を搭載する PC 100 台を 155 Mbps の ATM および 10 Mbps Ethernet の 2 系統のネットワークで相互結合したシステムである. ATM スイッチには 128 ポート の UTP-5 インタフェースを持つ日立製 AN1000-20 を採用した. 各 PC ノードの構成は表 5.1に示す通りである. また, 障害発生時やインストール時には各 PC のコンソールを直接制御する必要があるが, 設置場所や消費電力を抑えるために 10 ノード毎に 1 台のディスプレイ装置とキーボード / マウスを共有するディスプレイ切替器を備えている. 図 5.2に大規模 PC クラスタの外観を示す.

以下の節では, システムの主要な構成要素について詳述する.

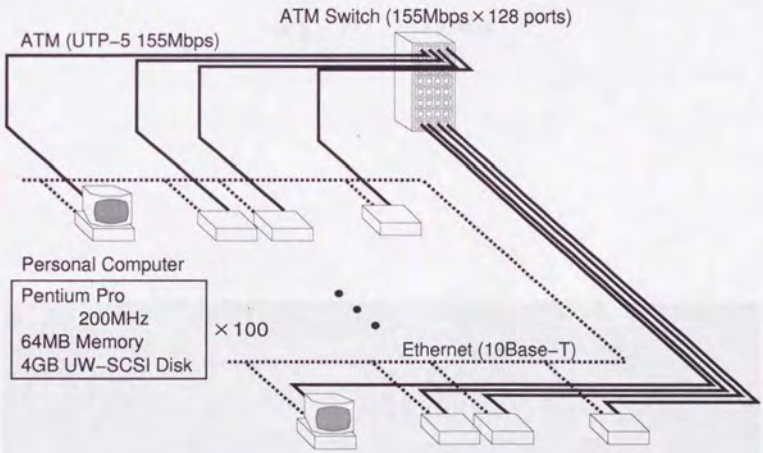


図 5.1: 大規模 PC クラスタのシステム構成

CPU	Intel Pentium Pro 200MHz
Chipset	Intel 440FX
Main Memory	64MB
Disk Drive (SCSI)	Seagate Barracuda 4.3GB
Disk Drive (IDE)	WesternDigital Caviar32500 2.5GB
ATM NIC	Interphase 5515 PCI ATM Adapter
OS	Solaris2.5.1 for x86

表 5.1: PC ノードの構成

図 5.2: 大規模 PC クラスタの外観

5.1.1 ATM スイッチ

ATM スイッチとして採用した日立製 AN1000 モデル 20 (AN1000-20) は, 128 ポートの 155Mbps UTP-5 端子を備え, 収納端子容量の合計が 20Gbps の ATM スイッチングノードである. シグナリングプロトコルは ATM Forum によって定められた UNI4.0 以下に準拠しており, CBR, VBR, ABR, UBR など様々なトラフィッククラスをサポートしている. また接続制御としては SVC と PVC どちらも用いることができ, ポイント・ポイントコネクション及びポイント・マルチポイントコネクションを張ることができる.

AN1000-20 の仕様を表 5.2にまとめる.

5.1.2 ATM NIC (Network Interface Card)

ATM NIC (Network Interface Card) としては, PCI バスに対応した Interphase 社製 5515 PCI ATM Adapter を採用した. 5515 PCI ATM Adapter は ATM ネットワーク上で 155Mbps の転送レートを実現する. この製品には SONET OC-3c のマルチモードファイバの SC コネクタタイプも存在するが, 本システムの構築に当たっては UTP-5 (Unshielded Twist Pair - category 5) の RJ-45 コネクタタイプを用いた. 5515 PCI ATM Adapter の諸元を以下に示す.

- 155 Mbps SONET over UTP copper
- 128KB standard buffer memory
- AAL5 ATM Adaptation Layer
- PCI Local Bus Revision 2.1 compliant
- 32-bit, zero wait-state PCI DMA master
- Up to 64-byte burst transfers
- Dual function memory management for Cell-FIFO or on-board packet reassembly
- Hardware-based Segmentation and Reassembly (SAR) functions

5515 PCI ATM Adapter のデバイスドライバには, コネクションの接続制御として SVC (Switched Virtual Channel) 対応のものと PVC (Permanent Virtual Channel) 対応のものとが存在する. PVC ドライバは, RFC-1483 LLC/SNAP encapsulation をサポートする IP

ATM セル仕様	ITU-T 勧告 I.361 に準拠
スイッチング容量	20Gbps
収容ポート数 (最大)	128
スイッチング方式	共通バッファ形 ATM スイッチ
論理インタフェース	
ユーザネットワーク IF	ILMI, UNI3.0/3.1/4.0 シグナリング
ノード間 IF	IISP, P-NNI フェーズ 1
OAM セル	AIS/FERF/Loopback
トラフィック制御機能	CBR, VBR, ABR, UBR
	遅延優先: 4 クラス / 廃棄優先: 2 クラス
	パケットレベル廃棄, UPC, シェーピング
コネクション設定制御	
設定範囲 UNI	VPI: 0 ～ 255, VCI: 0 ～ 65535
設定範囲 NNI	VPI: 0 ～ 4095, VCI: 0 ～ 65535
接続制御	PVC(固定接続)/SVC(交換接続)
接続形態	ポイント・ポイント, ポイント・マルチポイント
サーバ機能	
LAN エミュレーションサーバ	LAN Emulation Specification Version1.0
ATM ARP サーバ	RFC1483 / RFC1577
高信頼化	
冗長構成 (二重化)	プロセッサ部, スイッチユニット, ファイル
系切替	障害時の呼の引き継ぎ
ネットワーク管理機能	日立総合ネットワーク管理装置 (NETM*Cm2)
支援ツール	網設計支援 / 構成定義支援プログラムの提供
保守運用機能	障害自動検出機能, 障害部位切り分け機能
	プラグアンドプレイ

表 5.2: AN1000-20 の仕様

over ATM ドライバである [16, 21]. LAN Emulation SVC 及び Routed IP PVC のどちらも動作することを確認できたが, 実験に用いるアプリケーションでは, すべてのノードにおいて連続的にデータの送受信が行われるため, 本システムでは PVC ドライバを使用することとした.

5.1.3 ディスクドライブ

ディスクドライブには, WesternDigital 社製 WDAC32500 (Caviar32500) および Seagate 社製 ST-34371W (Barracuda 4LP) を用いた.

Caviar32500 は, IBM AT 互換 PC における標準 2 次記憶インタフェースである IDE インタフェースを持っており, OS 用のシステムディスクとして用いている. このディスクには, システムブート用のブロックや標準的な Solaris 2.5.1 のコマンド, 実行時ライブラリ, および設定ファイルなどを格納している.

一方, 高速なデータ転送が可能な Ultra-Wide SCSI インタフェースを有する Barracuda は, ディスク I/O の性能が重要になる関係データベース処理系に於けるデータベース格納用ディスクとして用いた. 表 5.3 にこのディスクの諸元を示す. このディスクでは, 最近の高速ディスクドライブによく見られる Zone Bit Recording 方式を採用しているため, トラック毎にセクタ数が異なり転送速度も変化するという特徴がある.

フォーマット容量	4.35 GB
平均セクタ数 / トラック	165 以上
トラック数	51,780
シリンダ数	5,178 (ユーザ用)
ヘッド数	10
ディスク枚数	5 (3.5 インチ)
記録方式	Zone Bit Recording, PRML (0,4,4)
内部転送レート	80 ～ 122 Mbits/sec
外部転送レート	40 MBytes/sec (同期転送時)
スピンドル回転数	7,200 RPM
平均レイテンシ	4.17 msec
バッファ容量	512 KByte
インタフェース	Ultra SCSI (ASA II, SCAM level 2)
平均トラック容量	102,500 Bytes
トラック密度	5,555 Tracks/inch
記録密度	123 Kbits/inch (ピーク)
平均アクセス時間	読み出し 9.2 msec 書き込み 10.2 msec
1 トラックシーク時間	読み出し 1.9 msec 書き込み 2.2 msec
最大フルシーク時間	読み出し 17.2 msec 書き込み 18.2 msec

表 5.3: SCSI ディスクドライブ ST-34371W の諸元

5.2 PC クラスタの基本性能 (1) — 一対一通信の性能

5.2.1 スループット

PC クラスタの基本性能として, ATM ネットワークを介した一対一通信性能を評価した. まず, *ttcp* プログラムを用いて一対一通信の実効スループットを測定した. また比較のために, 他の 2 種類の 133MHz Pentium PC 間のスループットの測定も行った. これらの PC の構成を表 5.4に示す. 以降では区別のために, 一方の 133MHz Pentium PC を "Model A", 他方の 133MHz Pentium PC を "Model B" と呼ぶ.

スループットの測定結果を図 5.3に示す. この図より以下の事柄がわかる. まず始めに, 200MHz Pentium Pro PC における一対一通信の実効スループットは非常に高い値を示している. 8 KByte から 16 KByte 程度のブロックサイズにおいて, スループットは 110 Mbps を越えている. この実験においては通信に TCP/IP over ATM を使っており, その場合プロトコル処理オーバーヘッドのためにスループットが低くなることを考えると, 155Mbps の回線の上で 110Mbps というのは驚くべき値である. これは 200MHz Pentium Pro の非常に高いパフォーマンスに依存した結果であろう.

Pentium Pro PC のスループットに比べ, 他のマシンの測定値は低くなっている. 133MHz Pentium Model A の場合, 最大実効スループットはわずか 30Mbps である. ここで実効スループットは, ホスト CPU のパワーだけでは決らないということがわかる. 何故なら 133MHz Pentium Model B においては, 90Mbps を越えるスループットが達成されているからである. これはおそらく PCI バスをドライブするマザーボードのチップセットの違いによるものであろう. Model B では Triton チップセットを使っている.

全てのモデルにおいて, メッセージブロックサイズが非常に小さい時にはスループットは

	Model A	Model B
CPU	Intel Pentium 133MHz	
Chipset	ALI	Triton
Memory	32MB	
Disk	1.2GB IDE HDD	
NIC	Interphase 5515 PCI ATM Adapter	
OS	Solaris 2.4 for x86	

表 5.4: 比較用 133MHz Pentium PC の構成

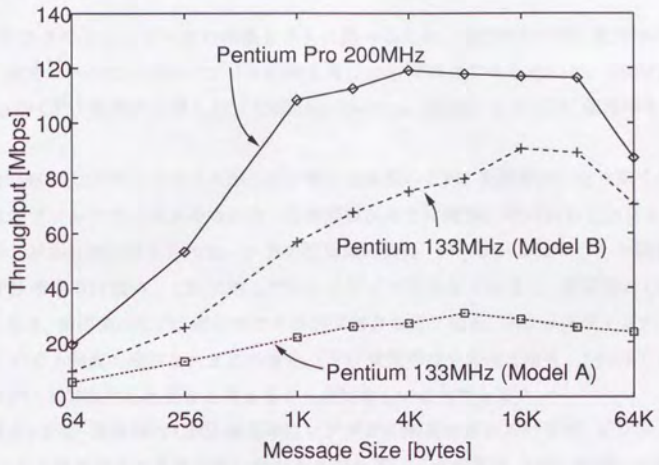


図 5.3: 一対一通信のスループット

かなり低い値になっている．これはブロックサイズが小さすぎる時には，送受信関数の呼出しが頻繁になり過ぎ，その結果パフォーマンスが低下するものと考えられる．またブロックサイズが非常に大きくなった場合にもスループットは低下する．これはバッファのオーバーフローやデータサイズと内部バスバンド幅とのミスマッチ等が起こっているためと考えられる．

5.2.2 CPU 使用率

PC クラスタのネットワークの特徴をさらに調べるため, 通信中の CPU 使用率の測定も行った. 測定ツールにはスループットの時と同じ *ttcp* プログラムを用いた. 200MHz Pentium Pro の CPU 使用率を図 5.4に, 133MHz Pentium Model A の CPU 使用率を図 5.5に示す.

図 5.4において, ブロックサイズが小さい時に送信側の CPU 使用率はかなり高くなっている. これはブロックサイズが小さいと, 送信関数が非常に頻繁に呼ばれることにより CPU 使用率が上がるものと考えられる. 一方の受信側の方は, データが少しずつしか到着しないため, CPU 使用率は低い. これに対しブロックサイズが大きくなると, 送信側の CPU 使用率は低くなり, 受信側の CPU 使用率の方が送信側より高くなる. ブロックサイズが 8K バイト - 16K バイト程度の適当な大きさの場合, CPU 使用率は十分低くなり, これはアプリケーションが PC 上で動作した場合を考えると大変好ましいことである.

一方図 5.5では, 送信側の CPU 使用率はジグザグの曲線を描いているが, ブロックサイズが 10K バイト過ぎのあたりまで高い値のままである. この結果は, CPU のデータ転送タイミングと内部バスの転送レートのミスマッチによるものと考えられる. このようにデータ送受信の際の CPU 使用率が高いと, アプリケーション実行にも影響が及び, パフォーマンスの低下が起こる可能性がある.

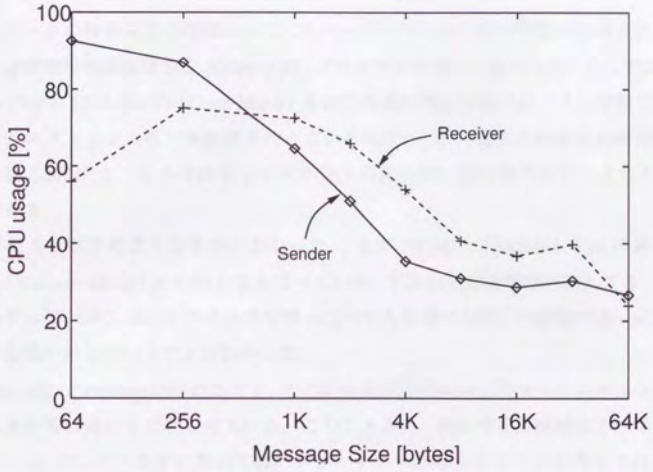


図 5.4: 200MHz Pentium Pro の CPU 使用率

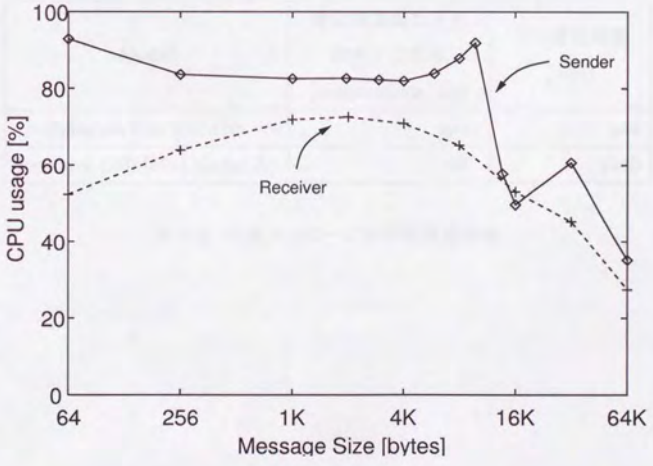


図 5.5: 133MHz Pentium Model A の CPU 使用率

5.2.3 平均遅延時間

ネットワークの特徴を示す指標として, スループットと共に遅延時間が注目される. そこで一対一通信の平均遅延時間を *Netperf*[17] プログラムを用いて測定した. ここでは 200MHz Pentium Pro と 133MHz Pentium Model A の往復遅延時間を調べた. ノード間で毎秒どれだけのリクエスト / レスポンス処理が行えるかを測定し, この値より往復遅延時間を計算した結果を図 5.5に示す. リクエスト / レスポンス処理の際に送受信するデータサイズは 1/1 バイトである.

この結果より以下のような事柄がわかった. まず 200MHz Pentium Pro の遅延時間は 133MHz Pentium Model A の約 1/3 となっている. すなわち遅延時間に関しても, たとえ同じソフトウェア, NIC, ATM スイッチを使っていても機種や CPU の種類の違いによって非常に大きな差が出るということがわかった.

また 200MHz Pentium Pro の方でも, その平均遅延時間は超並列マシンのネットワークと比較すると非常に長いものとなっている. このことから, 遅延時間に敏感なアプリケーションの場合には, PC クラスタにおいてはパフォーマンスが低下することが考えられる. しかしデータベース処理系のプログラムを始め多くのアプリケーションは, PC クラスタの高いスループットにより大きなメリットを得るものと予想される.

Model	単位時間当たりの要求 / 応答数 [transactions/sec]	平均遅延時間 [μsec]
Pentium Pro 200MHz	2234	448
Pentium 133MHz (Model A)	647	1546

表 5.5: 往復メッセージの平均遅延時間

5.3 PC クラスタの基本性能 (2) — ディスク読み出しの性能

ここでは, SCSI ディスクの読み出し性能を測定した. このディスクドライブは Zone Bit Recording を用いているため, 初めにトラック位置による読み出し速度の変化を調べた. 測定には, SCSI ディスクを raw device としてオープンし, 先頭から順に 64KBytes 単位で read システムコールを用いて読み出し, 16MBytes 毎にタイムスタンプを取ることで平均の読み出し速度を算出するという方法を採った. 測定結果を図 5.6に示す. この結果から, このディスクが 11 個のゾーンを持っていることが分かる. 特に先頭から 2GBytes 以降の領域ではゾーンが細かく分かれており, 読み出し速度も急速に低下している. これに対して前半の領域では, 2 つのゾーンが認められるものの, 読み出し速度は約 8.8 MBytes/sec と安定している. 以上から, この SCSI ディスクを用いる場合は, 可能な限り先頭に近い領域にデータを配置することが転送速度の点からも解析の容易さからも望ましいということが言える.

次に, 読み出しのブロックサイズと CPU 負荷の影響を調べる実験を行った. この実験は, Solaris 2.5.1 でサポートされている POSIX thread ライブラリを用いて 2 つのスレッドを生成し, 一方が read システムコールで読み出したデータをメモリ上のキューを介して他方のスレッドに受け渡すというプログラムを作成して行った. ここでは便宜的に前者のスレッドを producer process, 後者のスレッドを consumer process と呼ぶことにする. producer process は raw device に対して 256 MBytes のシーケンシャルリードを様々なブロックサイズを用いて行う. また, consumer process は受け取ったデータを直ちに捨ててしまう場合とメモリ上のデータのチェックサムを取ってから捨てる場合とを比較した. これらは, 実際の関係データベース処理に於ける CPU 負荷の違いを表している.

図 5.7に測定結果を示す. Zone Bit Recording の影響を調べるため, 速度の大きい先頭のゾーン (図中 "fastest zone") と, 速度の小さい後尾のゾーン (図中 "slowest zone") のそれぞれについて測定した. また, CPU 負荷の軽いものは図中 "Simple read" で表し, CPU 負荷の重いものは "With consumer process" で示した. CPU 負荷が軽い場合については, 8KBytes のブロックでも性能の低下は見られないが, ディスク I/O と CPU の重い処理とがオーバラップする場合には, システムコールのオーバヘッドにより十分なスループットが得られない. したがって, CPU 負荷を考慮に入れた時の最適なブロックサイズは 64 KBytes となる. この時スループットは 5.8 ～ 8.8 $\times 10^6$ byte/sec で与えられる.

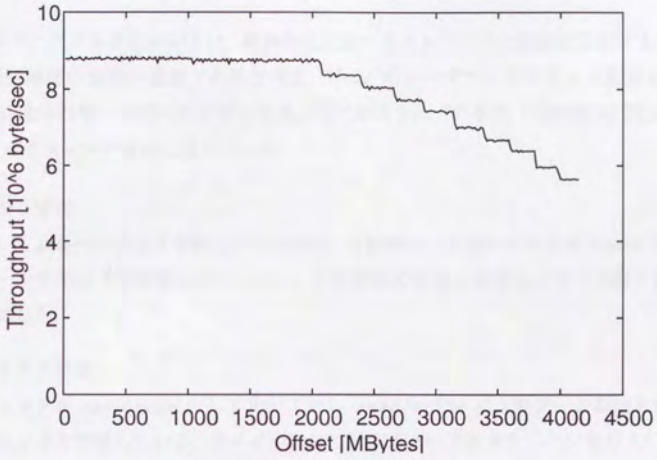


図 5.6: ディスク読み出し位置による転送速度の変化

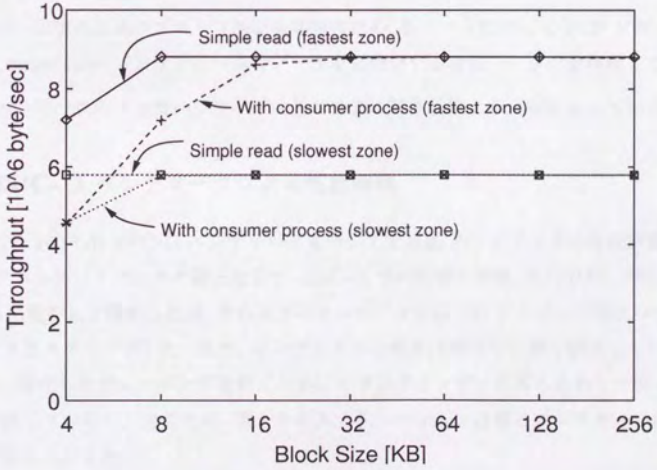


図 5.7: ディスク読み出し性能

5.4 大規模 PC クラスタにおける DBKernel の実装

大規模 PC クラスタにおいては, 最終的にはカーネルレベルでの実装を目指すものの, 可搬性と開発期間の短縮が重要であると考え, Unix のユーザプロセスとして実装を行った. DBKernel 全体は単一のデーモンプロセスとして実現されているが, 内部的には POSIX pthread によるマルチスレッド構成になっている.

- メモリ管理
  OS によるページングを防止するために, 起動時に一定量のメモリを mlock システムコールを用いて主記憶上にロックし, その領域の中から必要なメモリを割り当てるようにした.

- ディスク管理
  ディスクを raw device としてオープンし, `read/write` によりデータ転送を行う常駐スレッドを作成している. ファイル全体に対するリード要求をページ毎の I/O に変換するようにしている.

  基本性能の評価に基づき, ディスクに関する I/O サイズは 64KB としている.

- ネットワーク管理
  エラー回復のためのプロトコルが未実装のため, 各ノードに対して TCP ソケットを開き, `send/recv` インタフェースでデータを送受信する常駐スレッドを作成している.

  ネットワークのメッセージサイズは基本性能の評価結果より, 8KB としている.

5.5 TPC-D ベンチマークによる性能評価

ここでは, 100 GB TPC-D ベンチマークを用いて大規模 PC クラスタの性能評価を行う. 表 5.6にテストデータベースの諸元を示す. SDC-II での評価と同様, NATION, REGION はノード 0 に集中して格納したが, それ以外のリレーションは 100 台のノード間にハッシュを用いてデクラスタリングした. また, インデックスの効果は問合せに深く依存しているため, TPC-D に特化したチューニングを防ぐためにクラスタインデックスも含めて一切インデックスは作成していない. そのため, 問合せの入力リレーションは常にフルスキャンによりアクセスするものとした.

17 個の問合せは全て人手でコンパイル・最適化し, 実行プランを生成した. 付録 B に大規模 PC クラスタ上の 100GB TPC-D 問合せに対する実行プランを掲げる. 最適化の際には,

リレーション	タプル数	タプル長	デクラスタリング
SUPPLIER	1,000,000	208 bytes	S_Suppkey によるハッシュ分割
PART	20,000,000	176 bytes	P_Partkey によるハッシュ分割
PARTSUPP	80,000,000	220 bytes	PS_Partkey によるハッシュ分割
CUSTOMER	15,000,000	236 bytes	C_Custkey によるハッシュ分割
ORDER	150,000,000	140 bytes	O_Orderkey によるハッシュ分割
LINEITEM	600,037,902	156 bytes	L_Orderkey によるハッシュ分割
NATION	25	192 bytes	ノード 0 に集中格納
REGION	5	188 bytes	ノード 0 に集中格納

表 5.6: PC クラスタ上の 100GB TPC-D テストデータベース

結合演算と集計演算にハッシュに基づくアルゴリズムを用いるようにした．結合演算に関しては，GRACE ハッシュ結合アルゴリズムの改良版であるダイナミック Hybrid-GRACE ハッシュ結合アルゴリズム [26] を用いた．100GB データベース対する TPC-D ベンチマークにおいては各ノード毎に一つのバケットで主記憶に収まるため，スプリットフェーズ中にはバケットの書き出しは起こらない．したがって，直接ハッシュテーブルを構築した場合と比べて実行コストはほとんど変わらない．しかし，直接ハッシュテーブルを構築する方法では，タプルの選択率が不明なため，適切なテーブルサイズを選択するのは難しい．

また，集計演算に関しては，グループ化キーがリレーションの基本キーを含まない時はグループの数は多くならないと考え，処理ノード毎に独立に集計を行ってから大域的な集計を行うようにし，基本キーが含まれる時にはノード毎に異なるグループを割り当て，集計を行うようにした．

全ての問合せの実行に先立って，Query 9 をさまざまなスケジューリングにより実行した時の性能に対する影響を調べた．図 5.8に Right-deep tree による Query 9 の実行プランを示す．PART 以外のリレーションでは全てのタプルが選択されるため，ハッシュテーブルが大きくなるが，射影演算によってタプル長が短くなるため主記憶に収めることができる．また，Σ の部分は集計演算 (グループ毎の和) を表しており，各ノード毎にグループ毎のローカルな和を求め，最後にそれをマスタノード (ノード 0) に集めて全体の集計を行っている．その後の NATION との結合演算はマスタノードでローカルに行われる．NATION は非常に小さい (25 タプル) リレーションなので単一ノードでの実行が適している．

図 5.11はこの実行プランに基づいて Query 9 の実行を行っている時の CPU 使用率とディ

スクおよびネットワーク送受信の実効スループットを測定したものである. フェーズ #1 から #4 までのビルドフェーズでは, CPU 負荷は 30% 程度と軽く, ディスクのスループットも最大値の 8.8 MBytes/sec に近い値が出ている. したがって, これらのフェーズでは処理は完全にディスク I/O ボトルネックで行われていると言える. しかし, フェーズ #5 のプローブフェーズでは CPU 負荷がほぼ 100% に達し, ディスクのスループットも低下している. これは, LINEITEM に対する選択率が 100% であり, 元々負荷が高いのに加えて, 複数のプローブ処理を同時に行う必要があるために, CPU の能力が不足し, CPU ボトルネックに転じているためである.

これに対して, 図 5.9 に Left-deep tree に基づいてスケジューリングした実行プランを示す. ここでは, 選択演算によってタプル数が絞られる PART との結合演算を優先して行うようにしている. 集計演算以降の処理は Right-deep によるものと同一である. 図 5.12にこの実行プランに基づいて Query 9 の実行を行った時のトレースを示す. フェーズ #4 以外では, ディスクのスループットは最大値に達している. フェーズ #5 では ORDER による結合演算と集計演算とが同時に行われ, CPU 負荷が約 70% と高くなるが全体として処理は I/O ボトルネックである. フェーズ #4 の LINEITEM による結合演算では CPU 負荷が 100% を示しているが, ディスクのスループットは Right-deep のフェーズ #5 よりも高く, 経過時間は前者が 123 秒, 後者が 140 秒と 17 秒の差となって現れている. 他のフェーズでは読み出すリレーションのサイズで処理時間が決まるため, この 17 秒の差が結局 Left-deep と Right-deep 全体の差になっている.

図 5.10 に Bushy tree でスケジューリングした実行プラン, また図 5.13 にその実行トレースを示す. この実行プランでは, PART, SUPPLIER, および PARTSUPP 間の結合演算を Right-deep のセグメントとして実行しているが, この部分は I/O ボトルネックになっているので実行時間には変化が見られない.

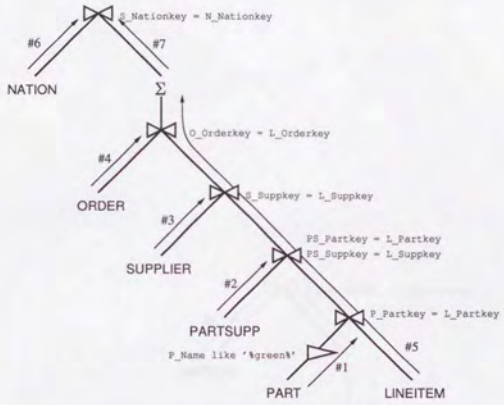


図 5.8: Right-deep tree による Query 9 の実行プラン

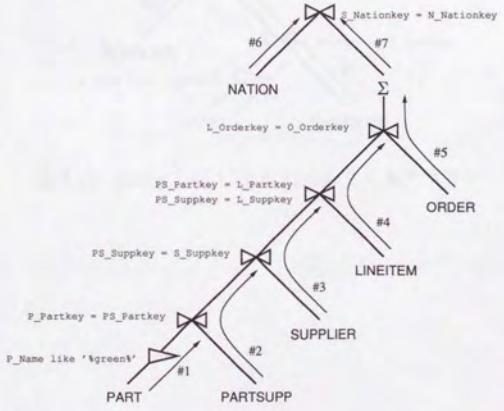


図 5.9: Left-deep tree による Query 9 の実行プラン

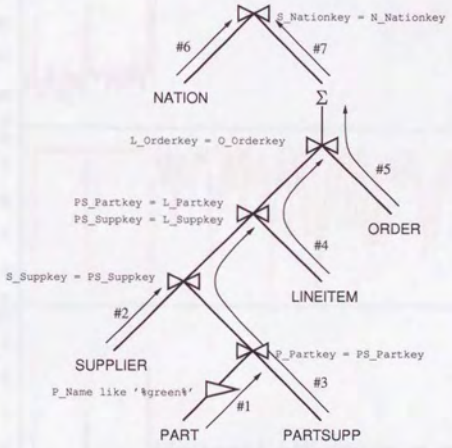


図 5.10: Bushy tree による Query 9 の実行プラン

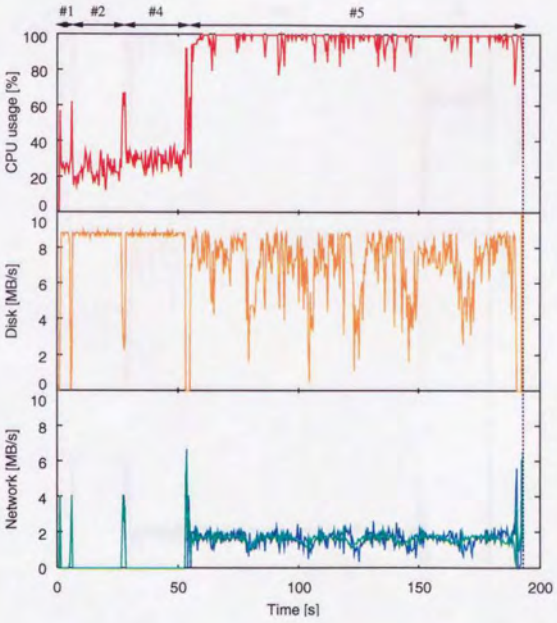


図 5.11: Right-deep tree による Query 9 の実行時トレース

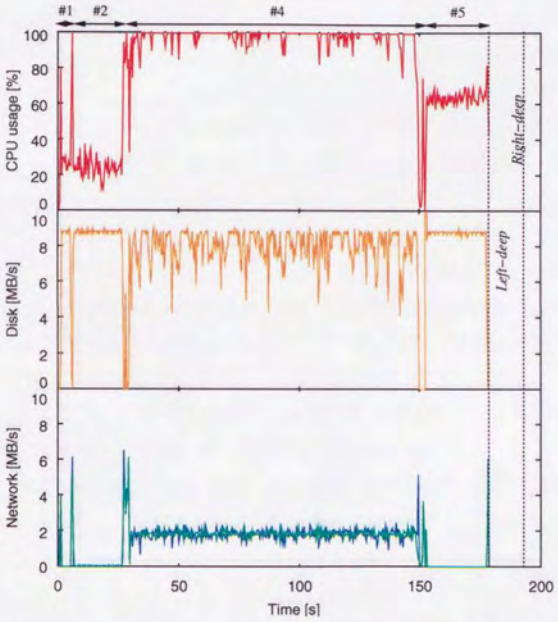


図 5.12: Left-deep tree による Query 9 の実行時トレース

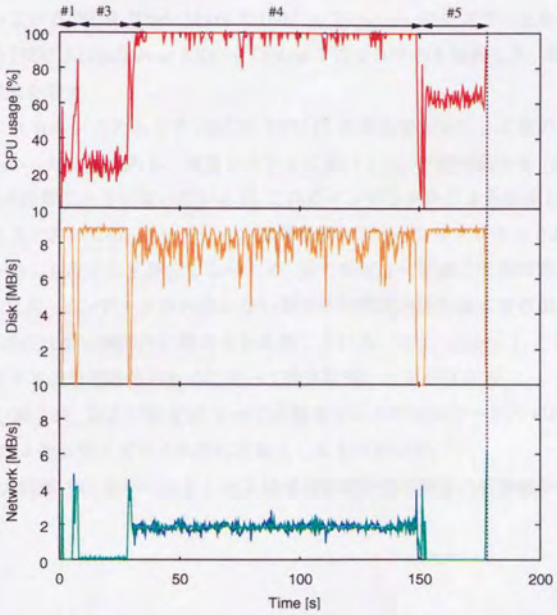


図 5.13: Bushy tree による Query 9 の実行時トレース

表 5.7 に上述の Right-deep (rd), Left-deep (ld), Bushy (b) のそれぞれの方式により大規模 PC クラスタ上で Query 9 を実行したときの性能を, 公表されている商用並列 DBMS の値とともに示す. どの実行方式においても大規模 PC クラスタの性能が優れていることが分かる. 最も高性能な商用機である NCR 5100M に比べても, 5 倍以上の性能を達成している. 一方, 大規模 PC クラスタについてはハードウェアのみのコストを示しているが, 他のシステムに比べて低コストで高い性能を得ているのは明らかである.

次に, 17 個の問合せ全体について商用システムとの比較を行う. 比較対象として, 処理性能で 1 位になっている NCR WorldMark 5100M + Teradata のシステムと価格性能比で 1 位になっている DEC AlphaServer 8400 + Oracle 7 のシステムを選択した. 表 5.8にこれらのシステムの構成を示す.

図 5.14 は, これらのシステム上で 100GB TPC-D の問合せを実行した時のそれぞれの問合せの実行にかかった時間である. 商用システムにおいては, 特定の問合せ (例えば Query 13) で実行時間が非常に小さくなっているが, これはインデックスによる効果と思われる. 大規模 PC クラスタにおいては, リレーションは常にフルスキャンでアクセスされ, また, 中間ファイルの書き出しもほとんど発生しないため, 全ての問合せを通じて処理時間がほぼ一定している. そのため, インデックスの効かない問合せの処理時間が長くなる商用システムと比べると, 全体的には短い時間内に問合せを処理している. 特に, Query 1 と 9 においては大規模 PC クラスタの性能は他のものに比べて非常に高いことが分かる.

また, 図 5.15, 図 5.16, および図 5.17 には上の結果をシステムのトータル CPU 性能, ディスク台数, およびメモリサイズでそれぞれ正規化したものを示す.

以上の結果, 大規模 PC をベースとした大規模関係問合せ処理系の有効性が強く裏付けされたといえる.

システム	実行時間 [sec]	コスト
Teradata on NCR 5100M 160 × 133MHz Pentium 20GB Main Memory 400 Disk Drives	953.3	$17M
Oracle 7 on DEC AlphaServer 8400 12 × 437MHz DECchip 21164 24GB Main Memory 84 Disk Drives	1884.9	$1.3M
Oracle 7 on SUN UE6000 24 × 167MHz UltraSPARC 5.3GB Main Memory 300 Disk Drives	2639.3	$2.1M
IBM DB2 PE on RS/6000 SP 306 96 × 112MHz PowerPC 604 24GB Main Memory 96 Disk Drives	2899.4	$3.7M
Oracle 7 on HP9000 EPS30 12 × 120MHz PA7150 3.75GB Main Memory 320 Disk Drives	7154.8	$2.2M
大規模 PC クラスタ **100 × 200MHz Pentium Pros** **6.4GB Main Memory** **100 Disk Drives**	**(rd) 193.7** **(ld) 177.2** **(b) 177.2**	**($0.5M)**

表 5.7: さまざまなシステムにおける 100 GB TPC-D Query 9 の実行時間

NCR WorldMark 5100M (running Teradata)
(処理性能 1 位 (1996/6/14))

アーキテクチャ	SE + SN (20 Nodes)
プロセッサ	160 × 133MHz Pentium
メモリ	20GB
ディスク	800GB (400 Drives)
コスト	$17M

DEC AlphaServer 8400 (running Oracle 7)
(価格性能比 1 位 (1996/11/22))

アーキテクチャ	SE
プロセッサ	12 × 437MHz DECchip 21164
メモリ	24GB
ディスク	361GB (84 Drives)
コスト	$1.3M

ATM 結合大規模 PC クラスタ (running DBKernel)
(1997/10/28)

アーキテクチャ	SN
プロセッサ	100 × 200MHz PentiumPro
メモリ	6.4GB
ディスク	430GB (100 Drives)
コスト	$0.5M (ハードウェアのみ)

表 5.8: 100 GB TPC-D 性能評価環境

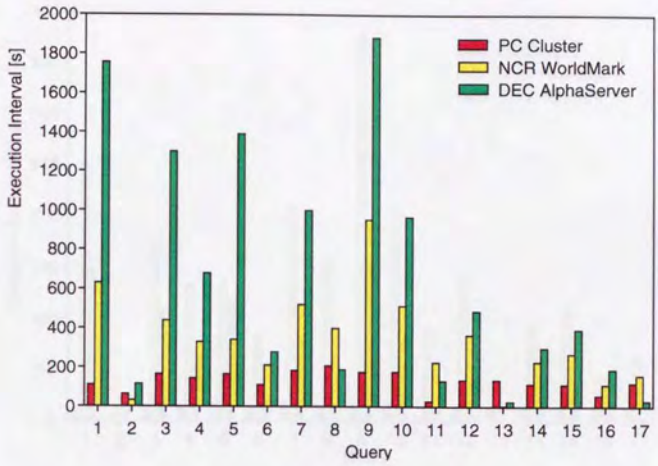


図 5.14: TPC-D による性能比較

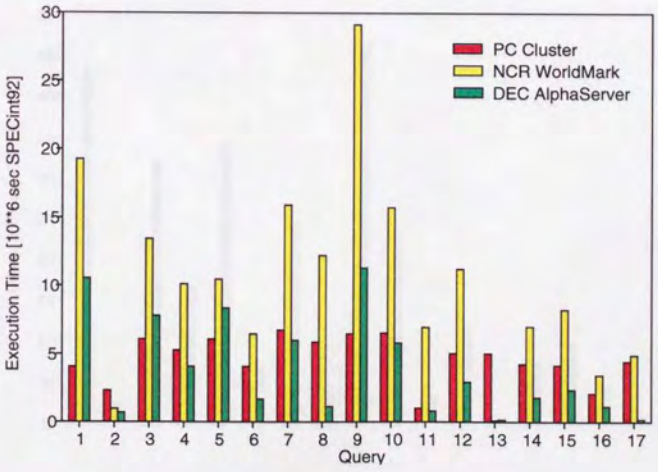


図 5.15: TPC-D による性能比較 (CPU 性能で正規化)

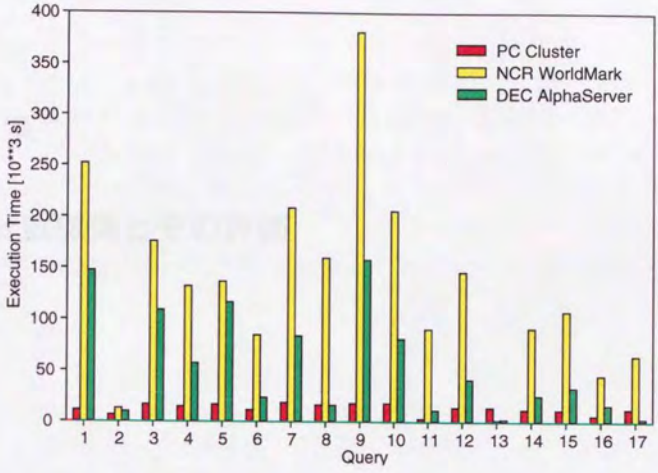


図 5.16: TPC-D による性能比較 (ディスク数で正規化)

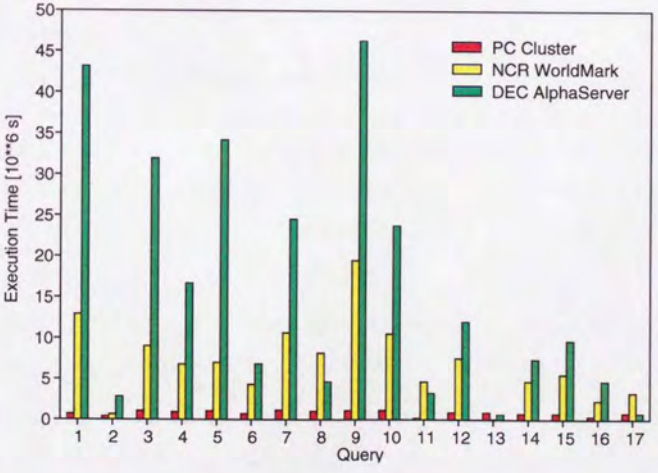


図 5.17: TPC-D による性能比較 (メモリサイズで正規化)

第 6 章

負荷分散機構とその評価

6.1 データスキューとその分類

一般に分散メモリ並列計算機環境においては, 処理ノード間の負荷の偏りによって並列性が低下するため, タスクスケジューリング等の研究が盛んに行われてきた. 並列データベースシステムにおいても, 実際のデータに内在するデータの偏り (スキュー) によってノード間の負荷が不均一になる結果, 処理速度の低下がもたらされることが知られている. データベースにおけるスキューに関しては, Walton らによって分類がなされている [35]. 最も基本的なスキューの分類としてイントリンジックスキュー (intrinsic skew) とパーティションスキュー (partition skew) が挙げられる. 以下にそれぞれのスキューについて説明する.

Intrinsic skew

属性値が均一に分散していない時に起こり, アトリビュートスキュー (AVS: attribute value skew) とも呼ばれる. このスキューはデータの特性であり, 結合演算アルゴリズムの実装方式によって影響を受けない. AVS を持つリレーション間の結合は, AVS を持たないリレーション同士の結合演算に比べ, 大きな結合率を持ち, 結合演算結果が非常に大きくなる可能性がある. この負荷を避ける並列実装は存在しないため, アルゴリズムによってノード間の負荷のバランスをとる必要がある.

Partition skew

パーティションスキューは並列実装において, タプルがノード間に不均一に分配される時に起こる. 結合演算の実装方式によってスキューの大きさが影響を受ける. パーティションスキューはイントリンジックスキューが存在しない場合にも起こりうる.

更に, パーティションスキューは, 結合演算の過程において、スキューの発生する段階に応じた分類が可能である.

- Tuple Placement Skew (TPS)

  初期状態のタプルの分布はノード間で異なる場合がある. 例えば, ユーザの指定した範囲によってリレーションが分割されている場合などが挙げられる.

- Selectivity Skew (SS)

  選択述語の選択率がノードによって異なるために起こる. 範囲分割属性に対する範囲選択を含む選択述語がこの例である.

- Redistribution Skew (RS)

  並列ハッシュ結合演算アルゴリズムのスプリットフェーズなどのように，結合演算の実行前に入力タプルを再分配する際に，各ノードに送られたデータ量がノード間で異なる場合に起こる．偶数と奇数で再分配した時の RS の例を図 6.1 に示す．

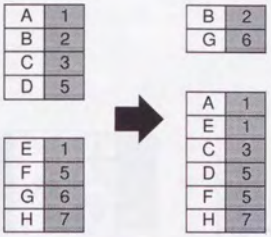


図 6.1: Redistribution skew

- Join Product Skew (JPS)

  並列ハッシュ結合演算アルゴリズムにおけるジョインフェーズにおいて，結合演算の結合率がノード間で異なる場合に発生する．これは，2 つのリレーションの組合せに対する特性であり，結合演算が行われるまでは検知できない．JPS の例を図 6.2 に示す．

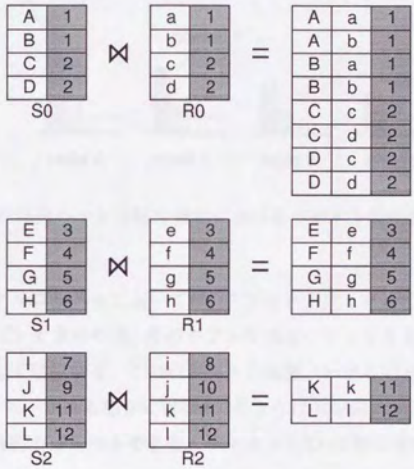


図 6.2: Join product skew

6.2 バケット分散ハッシュ結合による Redistribution Skew の解決

6.2.1 バケット分散ハッシュ結合アルゴリズム

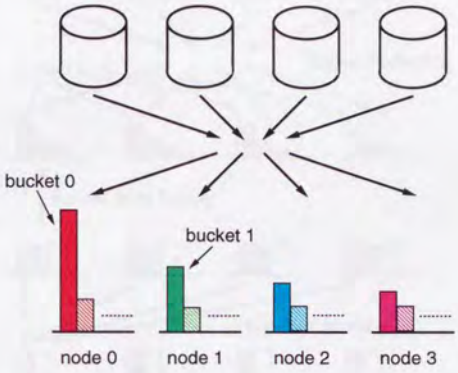


図 6.3: 並列ハッシュ結合演算におけるバケット集中格納方式

並列ハッシュ結合アルゴリズムにおいては, スプリットフェーズでタプルにハッシュ関数を適用してバケット ID を求めた後, そのタプルを当該バケットを処理すべきノードに対してネットワークを介して送信する. このバケットと処理ノードとの対応は静的に決定されているため, データスキューがある場合には図 6.3のように Redistribution Skew が発生し, 結果としてノード間の負荷のばらつきや必要なディスク I/O 回数の増加を招いてしまう.

そこで SDC-II では, Redistribution Skew による性能低下を解決するために以下の 2 つの技法を用いるバケット分散 (BS: Bucket Spreading) ハッシュ結合アルゴリズムを採用している (図 6.4).

- **動的なバケット割当て**
  バケット分割後に各バケットのサイズに基づいて処理ノードへの割当てを決定
- **バケットの分散格納**
  バケット分割中はタプルを各ノードに均等に配置

大きなバケットが生成されるのを防ぐために分割数を増やし, 小さなバケット (ファインバケット) を多数作る. ファインバケットのサイズに偏りが生じることは避けられないが,

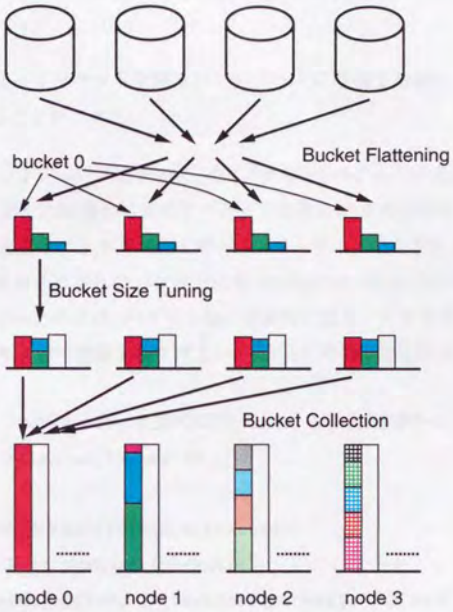


図 6.4: 並列ハッシュ結合演算におけるバケット分散格納方式

いくつかのファインバケットをまとめて 1 つのバケットを作ることでこの偏りを吸収することができる (バケットサイズチューニング). 動的なバケット割当てを採用することで, スプリットフェーズの完了後, 各ファインバケットのサイズの統計に基づいてバケットのまとめあげを行うことができる.

また, スプリットフェーズの間, ファインバケットを全てのノードにまたがって分散格納するのは以下のような理由による.

- 特定のノードにデータが集中するのを避ける.
  - ノード間で I/O 回数を均等化する.
  - バケットサイズが主記憶容量を越える, バケットオーバフローが発生した時に, その処理を並列化できる.

- 各ノードのファインバケットの分布が全体の分布と相似になるため，統計情報の取得が簡単である．
- 結合処理に先立ってバケットを割り当てのノードに収集する際に，ネットワーク上での輻輳を避けることができる．

このようにスプリットフェーズにおいて，各(ファイン)バケットが全てのノードにわたって均等に分散されるように配置する方式をバケット分散格納方式と呼び，このようなバケット分布を実現することをバケット平坦化と呼ぶ．ソフトウェア的な方法でバケット平坦化を行なうには，各ノードがそれぞれのバケットに含まれるタプルを全てのノードに対して均等に送信すればよい．そのためには，バケット毎に循環的に宛先ノードを変化させればよく，バケット毎の次の送信先を示す変数を持てばよい．図6.5にその概略を示す．

```
nextdest : array [0..N_BUCKETS-1] of 0..N_NODES-1;
for T in selected_tuples do
begin
    bucket = hash(T.join_attribute);
    send T to nextdest[bucket];
    nextdest[bucket] = (nextdest[bucket] + 1) mod N_NODES;
end
```

図 6.5: ソフトウェアによるバケット平坦化

6.2.2 不均一分布データに対する性能評価

ここでは，偏りのあるデータに対してSDC-II 上で GRACE アルゴリズムと BS (バケット分散) アルゴリズムを用いて結合演算を行った際の処理性能の比較を行う．問合せに用いたのは，以下のような単純な等結合である．

```
insert into result
select * from R, S
where R.joinkey = S.joinkey
and R.selkey < x
```

リレーション R および S は同じ属性を持ち, selkey 属性の値は均一なデータ分布とし, joinkey 属性の値が不均一に分布しているものとする. joinkey に基づいて分割した後のバケットサイズが, Zipf-like 分布にしたがって分布するものとした. また, リレーションはノード間にランダムに分割したが, 各ノード毎のタプル数は均等になるようにした.

図 6.6に処理ノード当たりのデータ量を一定にした時のスケールアップ特性を GRACE アルゴリズムと BS アルゴリズムについて測定した結果を示す. バケットサイズの分布は Zipf(0) および Zipf(1) 分布とし, 処理ノード (DPM) 数を 1 から 6 まで変化させた. DPM 内の DP 数は 4 とし, リレーション R と S のタプル数は DPM 当たり 1000000 タプル, タプル長 208 Byte, DPM 当たりのバケット数 12, selkey に対する選択率は 10%, 結合率は 100% とした.

均一なデータ分布 (Zipf(0)) に対してはどちらのアルゴリズムも良いスケールアップ特性を示している. BS アルゴリズムにはバケットを一度分散配置してから収集するためのオーバヘッドがあるが, 処理時間の増加はわずかであることが分かる. 偏りの大きい Zipf(1) 分布のデータに対しても BS アルゴリズムは同程度のスケールアップを示している. 一方, GRACE アルゴリズムの性能は DPM 数が増加するにつれて大きく低下している. この場合, バケットサイズの合計は主記憶容量より小さいが, GRACE アルゴリズムでは大きなバケットが特定の DPM に集中してしまうため, その DPM ではバケットをディスクにデステージングする必要があり, 処理時間の増加を招いてしまう. しかし, BS アルゴリズムではバケットを分散格納することでこの問題を避け, 一定の処理時間を維持している.

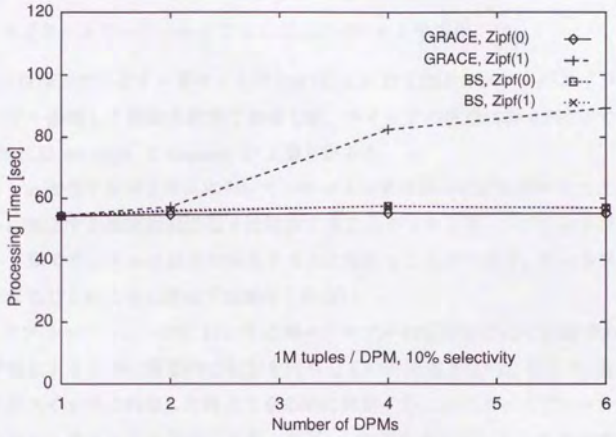


図 6.6: データスキューが存在する時のスケールアップ特性

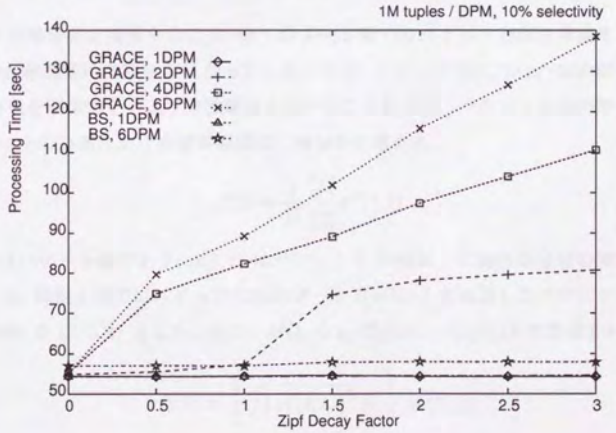


図 6.7: データスキューに対する処理時間の変化

6.3 SDC-II データネットワークによるバケット分散アルゴリズム支援

6.3.1 オメガネットワークハードウェアによるバケット平坦化

SDC-II で採用しているオメガネットワークは, 2 入力 2 出力のクロスバスイッチ素子を多段にシャッフル接続した間接多段網である [22]. スイッチの状態は図 6.8に示すように遷移し, 接続方向には straight と crossed の 2 通りがある.

並列ハッシュ結合アルゴリズムにおいてバケット分散格納方式を採用することにより, データスキューに起因する処理性能の低下は解決することができるが, スプリットフェーズにおいてはノード間でランダムな通信が発生することは防ぐことができず, ネットワークがボトルネックとなることによる性能低下は避けられない.

しかし, スプリットフェーズにおいては個々のタプルの転送先は全く任意であり, バケット分布を平坦化するために便宜的に宛先を付与しているに過ぎない. そこで, 各タプルの宛先をタプルがスイッチに到着した時点で適応的に決定することによってブロッキングを起こさないようなルーティングを実現できる. ただし, 宛先を全くランダムに決定するとバケット分布を制御できなくなってしまうので, 宛先制御の際にバケットの平坦分布を実現することを指標とする必要がある.

6.3.2 バケット平坦化ネットワークの動作原理

バケット平坦分布を実現するには, 全てのノード間でのバケット分布を考慮する必要があるが, 制御が非現実的な複雑さになってしまうため, スイッチ毎に二つの出力ポート間でのバケット分布を平坦化するよう分散制御を用いることにする. バケット分布の評価関数として, 各バケットの分布における標準偏差の二乗平均を考える.

$$f(t) = \frac{1}{B}\sum_{b=0}^{B-1}\sigma^2(b,t) \tag{6.1}$$

ここで B はバケット数であり, $\sigma(b,t)$ はバケット b の時刻 t における分布の標準偏差を表す. $\sigma(b,t)$ は, 時刻 t までにスイッチの出力ポート $i(=0,1)$ を通過したバケット b に属するタプルの総数 を $C_i(b,t)$ としたときに, $M(b,t) \equiv C_0(b,t) - C_1(b,t)$ で定義される変数 M を用いて,

$$\sigma(b,t) = \frac{1}{2}\sqrt{\{M(b,t)\}^2} = \frac{1}{2}|M(b,t)| \tag{6.2}$$

と表せる.

次に, 時刻 t_1 におけるスイッチの状態 $s(t_1)$ が図 6.8 の S0 であった時に, 新たなパケットが到着した場合を考える. 図 6.9 のように, 2 つの入力ポート $j(=0,1)$ にそれぞれパケット

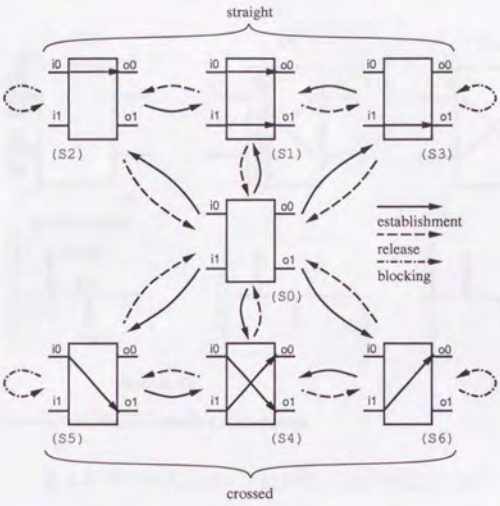


図 6.8: スイッチ素子の接続状態

b_j のタプルを含むパケットが到着したとすると, 遷移する状態は S1, S4 のいずれかであり, 状態遷移が完了する時刻を t_2, 遷移後の状態を $s(t_2)$ とすると, M は以下のように変化する.

$$\begin{aligned} M(b_0, t_2) &= M(b_0, t_1) + d(s(t_1), s(t_2)) \\ M(b_1, t_2) &= M(b_1, t_1) - d(s(t_1), s(t_2)) \\ M(b, t_2) &= M(b, t_1) \quad (b \neq b_0, b_1) \end{aligned} \tag{6.3}$$

ただし, $d(s_1, s_2)$ は状態が s_1 から s_2 に変化する時の M の変化を表し, $d(\mathrm{S0}, \mathrm{S1}) = 1$, $d(\mathrm{S0}, \mathrm{S4}) = -1$ である. このとき, f の変化は,

$$\begin{aligned} &f(t_2) - f(t_1) = \\ &\frac{1}{2B}\left[1 + d(s(t_1), s(t_2))\left\{M(b_0, t_1) - M(b_1, t_1)\right\}\right] \end{aligned} \tag{6.4}$$

で与えられる. f の増加を最小にすることによってパケット分布が平坦に近付くので, $s(t_2)$ は以下のようにして決定できる.

$$s(t_2) = \begin{cases} \mathrm{S1} & \cdots \quad M(b_0, t_1) - M(b_1, t_1) < 0 \\ \mathrm{S4} & \cdots \quad \text{otherwise.} \end{cases} \tag{6.5}$$

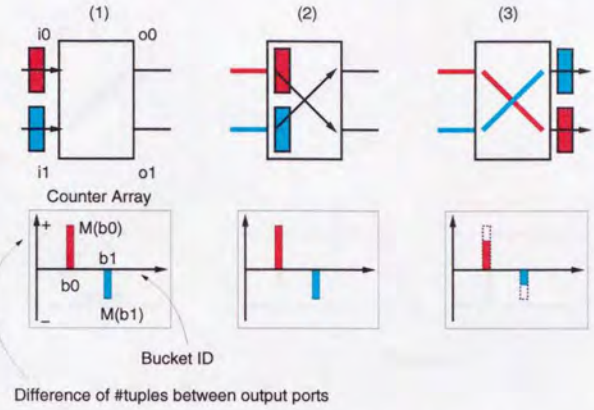


図 6.9: 2 つの入力ポートにパケットが到着した場合

以上のようにパケット平坦化ルーティングを行なう際は, パケットのタグとしてタプルの属するバケット ID を用い, そのバケット ID に対応する M の値を 2 つのポート間で比較して経路を選択する. これによって, パケットをブロックすることなく平坦なバケット分布を実現することができる.

状態 S0 において, 入力ポート j のみにパケットが到着した場合には, 前節の議論において $M(b_{1-j})$ に相当する項を 0 で置き換えればよい. すなわち, 入力ポート 0 のみに入力があった場合は,

$$s(t_2) = \begin{cases} \text{S2} & \cdots \quad M(b_0, t_1) < 0 \\ \text{S5} & \cdots \quad \text{otherwise.} \end{cases}$$

と決定される. 入力ポート 1 のみに入力があった場合も同様にして決定できる.

次に, 一方の出力ポートが使用中, つまり状態 S2, S3, S5, S6 のいずれかにおいて, 新たにパケットが到着した場合を考える. 例えば, 図 6.10のように時刻 t_1 において $s(t_1) = \text{S6}$ であり, 入力ポート 0 にバケット b_0 のタプルを含むパケットが到着したとすると, 次状態は S4 に遷移するかパケットをブロックして S6 に留まるかのどちらかである. S6 に留まった場合, $M(b_0)$ は一定であるため, 式 (6.1) の f ではブロックのペナルティを表すことができない. そこで, 式 (6.5) における $M(b_1)$ の値をブロックのペナルティを表すパラメータ $-T$

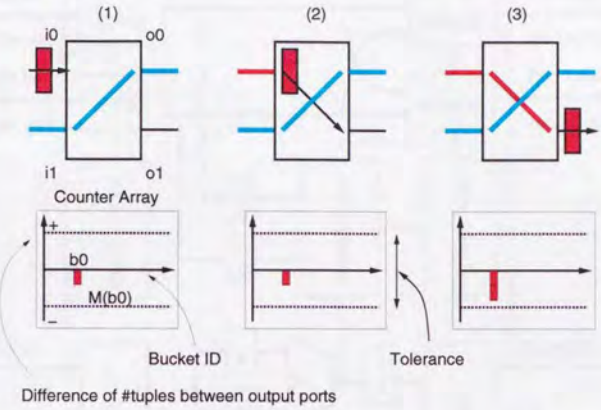


図 6.10: 一方の出力ポートが使用中にパケットが到着した場合

で置き換えた式を用いることにし,

$$s(t_2) = \begin{cases} \text{S6} & \cdots \quad M(b_0, t_1) + T < 0 \\ \text{S4} & \cdots \quad \text{otherwise.} \end{cases}$$

のように状態を決定する.

パラメータ T は許容値 (Tolerance) と呼ばれ, バケット分布の平坦度とブロック率とのトレードオフを表す. $T = 0$ の時はブロックのペナルティを 0 とみなした場合であり, 平坦度を重要視した場合に相当する. T の値を大きくするにつれ, S4 に遷移する割合が増えるため, ブロックの発生は減少する.

他の場合についても同様であり, 状態 S5 においては式 (6.5) の $M(b_0)$ を T で置き換え, 状態 S3 では $M(b_1) \to T$, 状態 S2 では $M(b_0) \to -T$ と置き換えて状態を決定する.

以上の議論では, バケット平坦化の際には全てのノードへパケットが送られることを暗黙の内に認めていた. しかし, ネットワークのパーティショニングが可能な場合, あるいはノードに障害が発生した場合には常に全てのノードが一つの演算に参加しているとは限らなくなる.

そこで, 任意数のノード間で処理が行なえるように出力ポートに重みづけをする [44]. すなわち, 処理に参加しているノードに重み 1, それ以外のノードに 0 を割り当て, 重みが 1 のノードだけにパケットが送られるようにする. スイッチでは出力ポート $i(= 0, 1)$ から到達

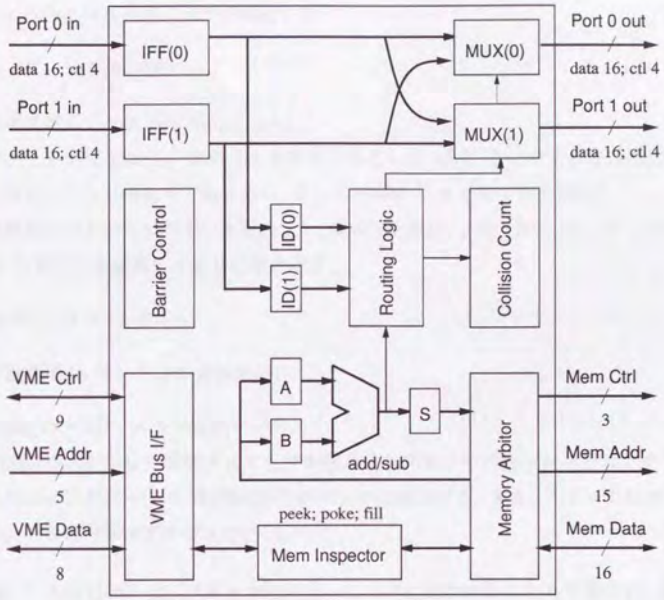


図 6.11: スイッチ素子のブロック図

できるノードについて重みの合計を取り, それをポート $(1-i)$ の重み W_{1-i} とする. そして, 式 (6.3) における $d(s_1, s_2)$ の値を $1, -1$ から $W_0, -W_1$ に変更することで, 出力ポート間の重みが異なる場合に重みの小さい方のポートからより多くのパケットが出力されるようにする.

6.3.3 パケット平坦化ネットワークの実装

スイッチ素子は, FPGA の一種である Xilinx LCA XC4010-5 (400 CLB, 10000 ゲート相当)[36] 上に実装した. 使用 CLB 数は 338, ゲート数に換算して約 6200 ゲートである. 履歴情報の配列 M はスイッチ素子に外付けの SRAM (32 KWord × 16 bit) に格納する. また, FPGA のコンフィグレーションは専用の管理プロセッサから行なう. スイッチ素子の内部は以下のようなブロックに大別される (図 6.11).

1. 入力レジスタ (IFF), マルチプレクサ (MUX)

2×2 のクロスバスイッチを構成する.

2. ヘッダレジスタ (ID)

3. 経路決定ロジック (Routing Logic)
   バケット平坦化モードでは, ID をアドレスとして $M(b)$ をメモリから読み出し, もう一方のポートが接続中であったら, さらに許容値 T をメモリから読む.
   減算器の出力から出力ポートを決定し, 経路決定後は, メモリから増分 W を読み, $M(b)+W$ を実行した結果をメモリに書き戻す.

4. 演算レジスタ (A,B)

5. 加減算器 (8 ビット 2 の補数表示)

6. VME バス I/F, メモリ監視レジスタ
   VME バスを介して管理プロセッサからスイッチ素子の内部レジスタにアクセスするためのインタフェース. 初期設定やデバッグに使用する. また, スイッチを介して外付けメモリの内容にアクセスできる.

許容値 T や増分 W は, スイッチ内部のレジスタに保存することも可能だが, 使用ゲート数を減らし, データパスを簡単にするために, 外付けメモリの中の特定アドレスを割り当てて使っている. ヘッダ中のバケット番号の領域は 13 ビット分なので, メモリの容量 32 KWord の内 8 KWord 分が履歴情報用に使用されることになる.

6.3.4 バケット平坦化ネットワークの性能評価

ここでは実験により，ハードウェアの機能を使用してバケット平坦化を行った際の性能をソフトウェアによる実現法と比較して評価する．

ネットワーク自体の性能を調べるために，送信データをプロセッサによって DM 上に生成し終えてから DNet を動作させ，その際のスイッチにおけるブロック率と受信したデータのバケット分布の平坦度を測定した．

送信データは，Zipf(1) 分布にしたがってランダムに分布するバケット ID を持ち，ハードウェアによるバケット平坦化機能を使用する際にはその値をヘッダのタグに書き込み，ソフトウェアによるバケット平坦化を行なう際にはバケット ID から決定された宛先ノードのアドレスをタグに書き込む．バケット数 32，転送ページ数 2044 ページ / ノードは全ての実験を通じて固定し，タプル長を 100 ～ 2000 Bytes の範囲から，バケット平坦化に用いる許容値の値を 0 から 10 の間から実験毎に選択した．また，ノード数を変える実験は 2 台から 6 台までの範囲で，その他の実験はノード数 4 台で実施した．

また，各スイッチにおけるブロック率は，管理プロセッサからスイッチ内部のブロックカウントレジスタを定期的に読み出して累積することで測定した．ブロック率は，全通信時間に占めるブロック時間の百分率で表す．

バケット分布の平坦度は，バケット毎のノード間分布の標準偏差の全バケットに関する平均，MSD (Mean Standard Deviation) で表す．

$$\mathrm{MSD} = \frac{1}{B}\sum_{b=0}^{B-1}\sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left\{C_{b,i} - \frac{1}{N}\sum_{j=0}^{N-1}C_{b,j}\right\}^2} \tag{6.6}$$

ここで B はバケット数，N はノード数であり，バケット b に属するタプルの内，ノード i が受け取ったタプルの数を $C_{b,i}$ で表す．MSD の値が大きいほど平坦度が悪いことを表している．

図 6.12および図 6.13に，バケット平坦化ネットワークの許容値を変化させた時のブロック率および平坦度の変化を，ソフトウェアによるバケット平坦化による結果と共に示す．図 6.12から，タプル長による影響はあまり見られず，許容値を大きくするにつれてブロック率が急速に小さくなることが分かる．許容値が 0 の時でもブロック率はソフトウェアによる平坦化よりも小さくなっているが，1 以上の値に設定することで大幅にブロックを解消できることが分かる．特に，3 以上の値にするとブロック率はほとんど 0 に近くなる．一方，許容値を大きくすることのペナルティは，結果のバケット分布の平坦度が低下することであるが，図 6.13によるとバケットサイズのばらつきは 1 タプル以内に収まっており，平坦度の低下は問題にならないといえる．

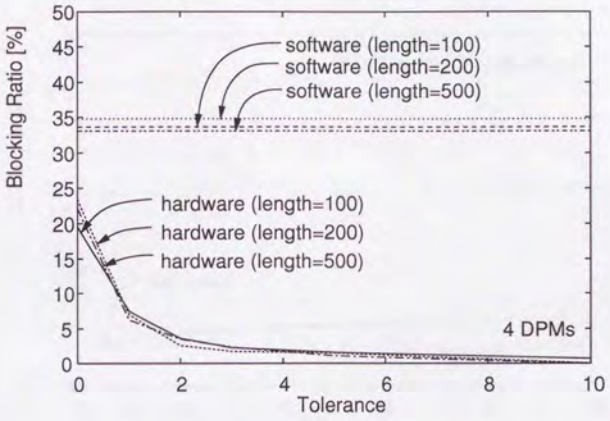


図 6.12: 許容値に対するブロック率の変化

次に, DNet 上の転送とプロセッサからのデータ生成がオーバラップして行なわれる場合について, データの生成速度を変化させてシミュレーションを行なった. データはページ単位で発生し, その発生間隔は幾何分布に従うものとした. 図 6.14は, それぞれデータの生成速度に対するスループットおよびブロック率の変化のシミュレーション結果である. この図から, ソフトウェアによる平坦化では生成速度が 10 MB/s 程度に達するとブロックの発生によってスループットが飽和するのに対し, 許容値を設定したハードウェアによる平坦化では約 20 MB/s まで入力に追随できることが分かる.

以上から, パケット平坦化機能をネットワークのハードウェアで実現することで, ソフトウェア的な実現で生じるネットワークの輻輳をほとんど 0 にすることができ, 実効的なバンド幅を倍にできることが分かった.

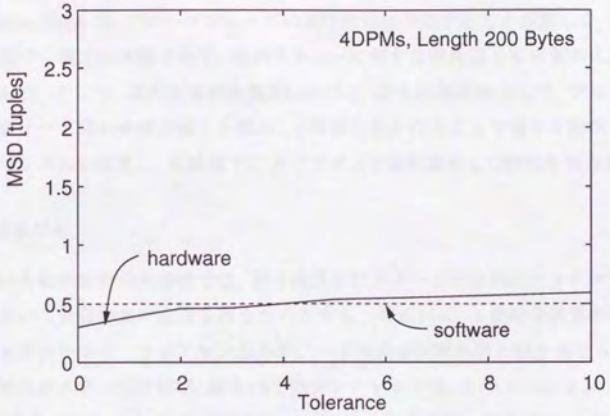


図 6.13: 許容値に対する平坦度の変化

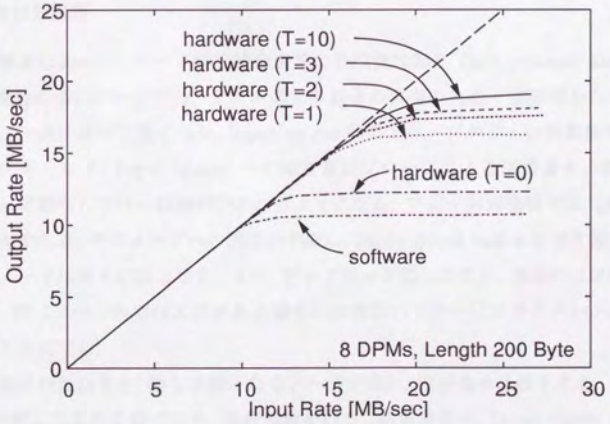


図 6.14: 入力速度の変化に対するスループットの変化

6.4 動的負荷分散による Join Product Skew の解決

Join Product Skew は, プローブフェーズの実行前には予測することが難しく, 事前のサンプリングに基づく推定は困難であり, 他のスキューに対する解決法とは本質的に異なった方法が必要である. そこで, 並列多重結合演算における JPS の解決法として, プローブフェーズ実行時に各ノード間の負荷の偏りを検出し, 負荷分散を行うことで偏りを解決する動的負荷分散アルゴリズムを採用し, 大規模 PC クラスタ上での実装および評価を試みた.

6.4.1 実装モデル

ここで用いる動的負荷分散機構では, 結合演算を行うノードとは別にフォアマンという制御ノードを用いて負荷の集中監視を行うものとする. 図 6.15に, 2 重結合演算の処理を行う場合の処理モデルを示す. フォアマン以外のノードは結合演算処理に割り当てられる. 1 つの結合演算単位をステージと呼び, 図 6.15ではリレーション A と リレーション S の結合演算 Join 1 がステージ 1, Join 1 の結果とリレーション B との結合演算 Join 2 がステージ 2 に相当する. 各々の結合演算に対する処理ステージは複数ノードにまたがり, ノードごとのステージをステージフラグメント (stage fragment : SF) と呼ぶ. 各ステージフラグメントは, ビルドフェーズで生成されたハッシュテーブルが対応する.

6.4.2 負荷分散機構

多重結合演算において, ノード間で結合演算の生成量の偏り (join product skew) が生じた場合, 生成量の多いノードでは, タプル同士の結合の負荷のため一定時間あたりのプローブ数が他のノードに比べて低くなり, Input queue 内のプローブタプルの消費量が減少する. この負荷が大きくなり, Input queue への転送量がプローブによる消費量を上回った場合, Input queue が飽和しフロー制御が行われるようになる. フロー制御機構では, Input queue buffer が飽和すると, そのノードへの送信が中断し, Input queue buffer に空き領域ができるまで送信元ノードは待ち状態となる. また, デッドロック防止のため, 後段のステージは優先度を高くし, 同じノード内では入力がある場合には後段のステージフラグメントの処理を優先して行うことにする.

フロー制御が行われると, 待ち状態となるノードが現れ, 処理効率は低下する. 負荷を分散させ, 待ち状態になるのを防ぐため, 負荷分散を行う. 負荷分散は, Input queue にたまっているタプルがある閾値以上となった時, または, 第 1 ステージで一定数のプローブの実行後に発動される. 後者は, 偏りが Input queue に蓄積する程大きくない場合に対する負荷分散

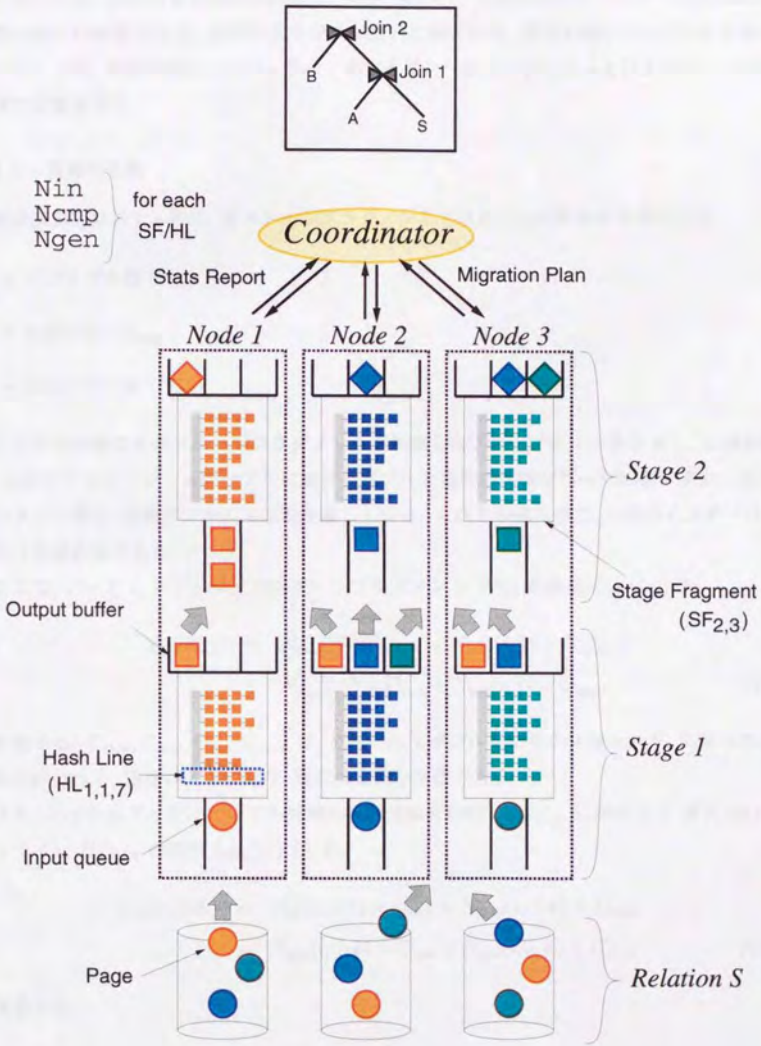


図 6.15: 動的負荷分散機構の実装モデル

のためである. 負荷分散が起動されると, まず, 各ノードの統計情報をフォアマンに収集し, 負荷の偏りの判定を行う. 負荷の偏りが検出された場合には, 負荷の高いノードから負荷の低いノードに, 負荷の高いハッシュラインのマイグレーションを行うことによりノード間の負荷の分散を行う.

6.4.3 負荷の定義

結合演算を処理する際に, 各ステージフラグメントでは以下の統計情報を保持する.

- 入力タプル数 (N_{in})
- 比較回数 (N_{cmp})
- 出力タプル数 (N_{out})

入力タプル数はそのステージフラグメントが処理したプローブタプル数を表し, 比較回数は, 入力タプルがハッシュテーブルにおけるタプルと条件の比較を行った回数を表す. また, 出力タプル数は, 結果タプルの生成数を表している. これらの値は全て, 一定のインターバルにおける統計値である.

ここで, ノード i, ステージ j のステージフラグメント $SF_{i,j}$ の負荷 $L_{SF}(i,j)$ を,

$$\begin{aligned} L_{SF}(i,j) &= N_{in}(i,j) \times C_{recv} + N_{cmp}(i,j) \times C_{cmp} \\ &+ N_{out}(i,j) \times C_{gen} + N_{out}(i,j) \times C_{send} \end{aligned} \tag{6.7}$$

と定義する. $C_{recv}, C_{cmp}, C_{gen}, C_{send}$ は, それぞれ 1 タプルあたりの受信コスト, 比較コスト, 結果生成コスト, 送信コストであり, 既に測定済みの値である.

また, ハッシュラインについても同様の統計情報を保持し, $SF_{i,j}$ における k 番目のハッシュライン $HL_{i,j,k}$ の負荷 $L_{HL}(i,j,k)$ を,

$$\begin{aligned} L_{HL}(i,j,k) &= N_{in}(i,j,k) \times C_{recv} + N_{cmp}(i,j,k) \times C_{cmp} \\ &+ N_{out}(i,j,k) \times C_{gen} + N_{out}(i,j,k) \times C_{send} \end{aligned} \tag{6.8}$$

と定義する.

6.4.4 負荷分散アルゴリズム

フォアマンにおいては以下のような処理が行われる.

1. ステージフラグメントの統計情報の収集

   各ノードから全ステージにおけるステージフラグメントの統計情報を収集する.

2. ステージフラグメントの負荷の算出

   収集された統計情報をもとに, 式 (6.7) を用いて各ステージフラグメントの負荷を算出する.

3. 負荷の偏りの判定

   負荷の偏りの判定はステージごとに行い, 偏りが検出された場合には, 以降の処理を行う.

4. ハッシュラインの統計情報の収集

   各ノードから, 負荷の偏りが検出されたステージのハッシュラインに関する統計情報を取得する.

5. ハッシュラインの負荷の算出

   式 (6.8) を用いて各ハッシュラインの負荷を算出する.

6. マイグレーションのプランニング

   ステージフラグメントの負荷が均等になるように, ハッシュラインのマイグレーションプランを立てる. このとき, 負荷の大小でハッシュラインをソートし, 負荷の大きいハッシュラインからマイグレーションプランに加えるようにする.

7. マイグレーションプランの送信

   作成されたマイグレーションプランを各ノードに送信する.

各ノードはフォアマンからマイグレーションプランを受け取ると, ハッシュラインのマイグレーションを行う. 現在プローブ中のハッシュラインのマイグレートに関しては, プローブ終了までマイグレーションを延期する. ただし, ハッシュラインマイグレーションの間にノード間の同期をとる必要はなく, 特にマイグレーションに関係の無いノードに関しては結合演算が中断するようなことはない. 以上のようにして、ハッシュラインのマイグレーションが行われる (図 6.16).

各ノードは, 各ハッシュラインのマイグレート先を登録したマイグレーションテーブルを保持しており, どのノードにどのハッシュラインが存在しているかを把握している. 全ノー

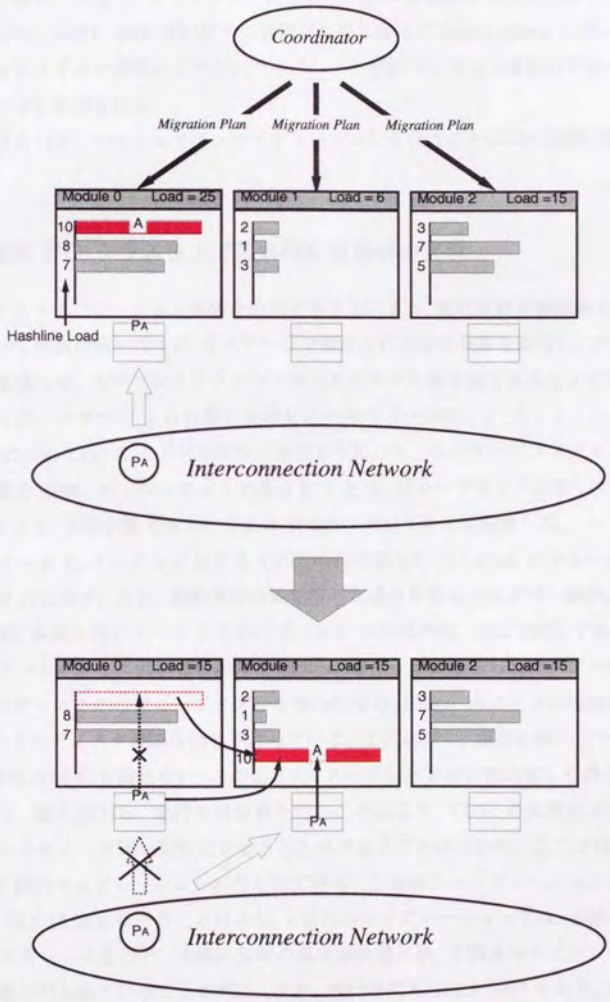


図 6.16: ハッシュラインマイグレーション

ドのマイグレーションが完了した時に, ハッシュラインのマイグレート先の登録を行う. 下位ステージの各ノードは, このマイグレーションテーブルを参照することによりタプルの送信先を決定する. しかし, 既に古い宛先に送信されてしまって Input queue に溜っているが, 対応するハッシュラインが別のノードにマイグレートされてしまった場合はプローブの際に再び他のノードに転送される.

以上のようにして、ハッシュラインマイグレーションを行うことにより負荷分散を実現する.

6.5 大規模 PC クラスタ上での負荷分散機構の実現

DBKernel にマイグレーション機構を追加することにより, 動的負荷分散機構を実装し, 10 ノードを用いて性能評価を行った. 2 ステージで構成される結合演算を使用し, プローブタプルの属性値を偏らせ, ステージフラグメント間で入力タプル数を偏らせることにより負荷の偏りを実現した. ステージ 1 の負荷の比率をノード 1 から順に, 9 : 1 : 1 : ··· : 1 とし, ステージ 2 については, 各ノードで均等となるようにした. 各ステージフラグメントのハッシュライン数を 3000, ハッシュラインの長さを 1 とし, プローブタプルに関しては, サイズを 256 Byte とし, 初期状態で各ノード毎に 500,000 タプルを分散配置した.

この時のノード 1, ノード 2 における CPU の利用率ならびに Disk のスループットの時間変化を図 6.17に示す. また, 動的負荷分散を行った場合を図 6.18に示す. 動的負荷分散を行わない場合, 負荷の高いノード 1 における CPU の利用率は, ほぼ 100% であり, ディスクのスループットが著しく低いことがみてとれる. 他ノードから送られたタプルの処理に忙しく, 新たにディスクからプローブタプルを読み出せないため, 他ノードの処理が終った時点で, 自ノードのディスクの読み出しを行っている. これに対し, 負荷の軽い ノード 2 では CPU 利用率は 50% にも満たない. このように, ノード 1 と 2 の状態は著しく異なっている.

これに対し, 図 6.18から, 動的負荷分散を行うことにより, CPU の負荷およびディスクのスループットをノード間で均衡化させることができることが分かる. ここでは, ステージ 1 において 2 回のマイグレーションが行われている. この時のマイグレーションプランを表 6.1 に示す. 図 6.18 と比較することにより, 1 回目のマイグレーションでは, 多数のハッシュラインのマイグレートを行い, 大幅に負荷の偏りが改善され, 2 回目のマイグレーションによって微調整が行われていることが判る. また, 実行時間も 109.6 [sec] となり, 動的負荷分散を行わない場合の 227.1 [sec] に対して, 大幅に短縮出来ることが判る.

以上のように, ハッシュラインマイグレーション技法を用いた動的負荷分散機構を PC クラスタ上に実装することで, JPS が存在する時の大幅な実行時間の改善を確認し, 本手法の

有効性が明らかになった.

source	# of Hash lines	
→ dest	1st	2nd
1 → 2	221	45
1 → 3	279	–
1 → 4	199	71
1 → 5	246	38
1 → 6	211	46
1 → 7	162	72
1 → 8	256	32
1 → 9	185	72
1 → 10	231	–

表 6.1: 動的負荷分散を行った場合のマイグレーションプラン

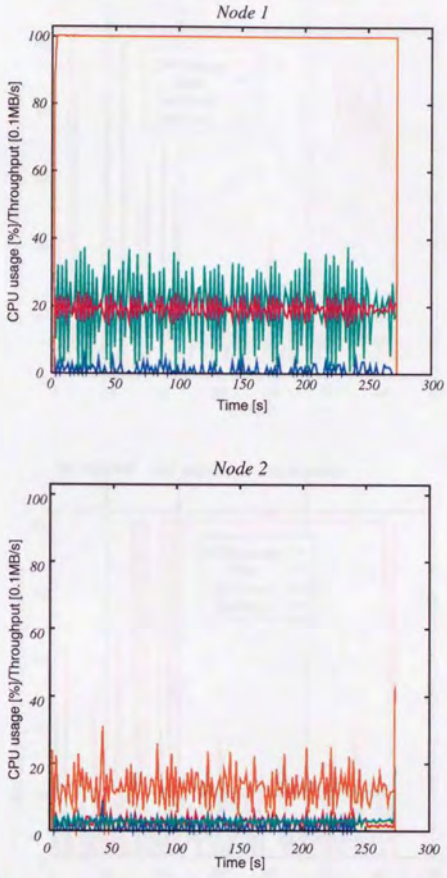


図 6.17: 動的負荷分散を行わない場合の CPU 利用率と Disk スループットの時間変化

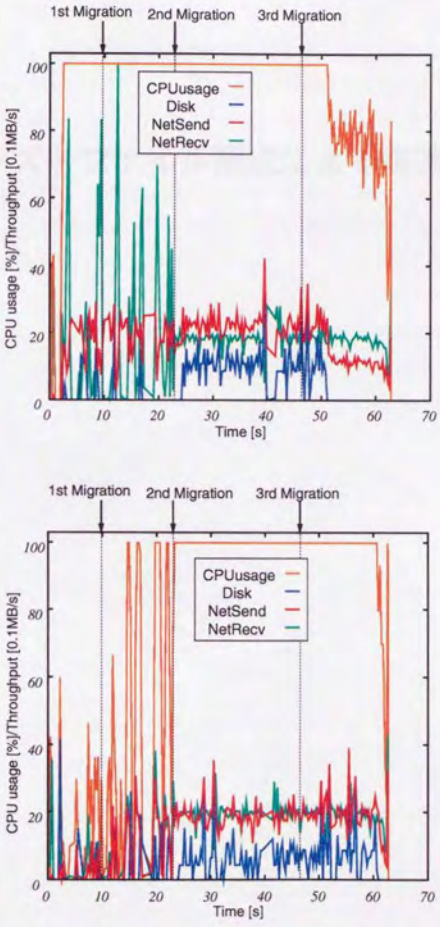


図 6.18: 動的負荷分散を行った場合の CPU 利用率と Disk スループットの時間変化

第 7 章

トランスポーズドファイル編成による高速化

7.1 TPC-D ベンチマーク問合せにおける射影演算の効果

5.5節の解析から分かるように, PC クラスタ上での TPC-D ベンチマーク問合せの実行時間は, 入力リレーションに対するディスク読み出しの時間が支配的である. しかしながら, ディスクから読まれるデータには選択演算や射影演算により捨てられる不要な部分が含まれているのが通常である. 選択演算については, 適切なインデックスの作成により, 必要なタプルのみにアクセスする事ができるようになるが, それによる I/O 量の削減の効果は問合せの選択条件や, 属性値の分布に大きく影響を受ける. 一方, 射影演算の効果はリレーションの定義と問合せの指定から容易に予測する事ができる. 例えば, 表 7.1に示すように, Query 9 においてはディスクから読み込まれたデータの内 80% 以上が捨てられることがわかる. この様な状況は, 意思決定問合せにおいては全ての属性にアクセスすることはまれであるため, 頻繁に生じるものと思われる.

そこで本章では, 不要な属性に関する余分な I/O を削減するため, リレーションの属性を互いに独立なファイルに格納するトランスポーズドファイルを物理記憶編成として採用することを検討する.

リレーション	全ての属性	必要部分のみ
SUPPLIER	208 MB	8 MB
PART	3.52 GB	160 MB
PARTSUPP	17.6 GB	1.28 GB
ORDER	21.0 GB	1.20 GB
LINEITEM	93.6 GB	21.6 GB
NATION	4.8 KB	800 B
計	136 GB	24 GB

表 7.1: 100GB TPC-D Query 9 における射影演算の効果

7.2 トランスポーズドファイル

トランスポーズドファイル (垂直分割とも呼ばれる) はリレーションを属性毎に分割し, 互いに独立なファイルに格納するものである [2, 3]. 同じタプルに属していた属性間の相関を維持するため, 図 7.1に示すように, 分割された属性のそれぞれにタプル ID (TID) を付加す

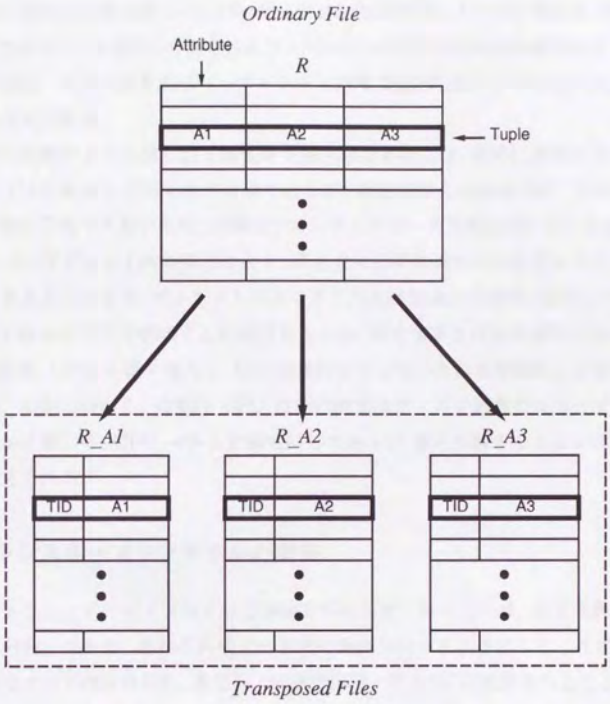


図 7.1: トランスポーズドファイル

る．したがって，元のタプルやそれに射影演算を施したものは，必要な属性を TID をキーとして結合することで復元可能である．

トランスポーズドファイルはリレーションの分割の仕方によって，

- フルトランスポーズ方式
- パーティション方式
- クラスタ方式

などの種類に分けられる．フルトランスポーズ方式は最も単純な方式であり，リレーションを完全に１つ１つの属性に分割してしまう．すなわち，分割されたリレーションには TID を

除くと 1 つの属性しか含まれない. パーティション方式ではいくつかの属性を分割せずに格納する. したがってこの場合, 分割されたリレーションには TID 以外の属性が 2 つ以上存在することになる. クラスタ方式はパーティション方式で分割したファイル間に重複した属性を持たせる方式である.

定型問合せ処理のように使われる属性が予測可能な場合には, 同時に読み出される属性を一つのファイルに格納しておくのが有効であるが, 非定型問合せ処理では, どの属性が使われるかは一般に予測できないため, 以降はフルトランスポーズ方式を用いることを前提する.

トランスポーズドファイルの採用により, 問合せに必要な属性のみをディスクから読み出すことができるようになり, ディスク入出力コストの大幅な減少が期待できる. 一方, 複数の属性を TID 結合により再構成する必要があるため, 結合演算を何回も実行しなければならない. その結果, CPU 負荷が増大し, I/O 削減によりもたらされる性能向上が制限を受けることになる. しかしながら, 近年の CPU 性能の向上はディスク装置のスループット向上に比べてはるかに著しく, CPU コストを犠牲にしても I/O 量を削減することが将来的に有利であると考えられる.

7.3 トランスポーズドファイルの実装

図 7.2 にトランスポーズドファイルの実装方式を示す. 各タプルは, まず処理ノード間に亙って水平分割してから, それぞれのノード内で垂直分割するようにした. そのため, TID の有効範囲はノード内に限られ, 異なるノードでは同一の TID が使用されることになる. システム全体で一意となる TID が必要になる時は, ノード ID とローカルな TID の組を用いる. ローカルな TID はファイルの先頭からのタプルのオフセットとみなす事ができるため, 全ての TID をディスク上に格納する必要はない. そこで, ディスクページの先頭のタプルに関する TID のみを格納し, その値からそれ以外のタプルの TID を実行時に導出する方式を採った. トランスポーズドファイルにおいては, 短い属性のそれぞれに TID が必要になるため, TID を省略する事でディスク使用量を大幅に削減する事が可能になる.

トランスポーズドファイル上での結合演算は以下の 2 つに分類できる.

- リレーション間結合
  問合せに指定された本来の結合演算.

- リレーション内結合 (TID 結合)
  分割された属性から元のタプル (あるいはそれに射影演算を施して得られるタプル) を再構成するための結合演算.

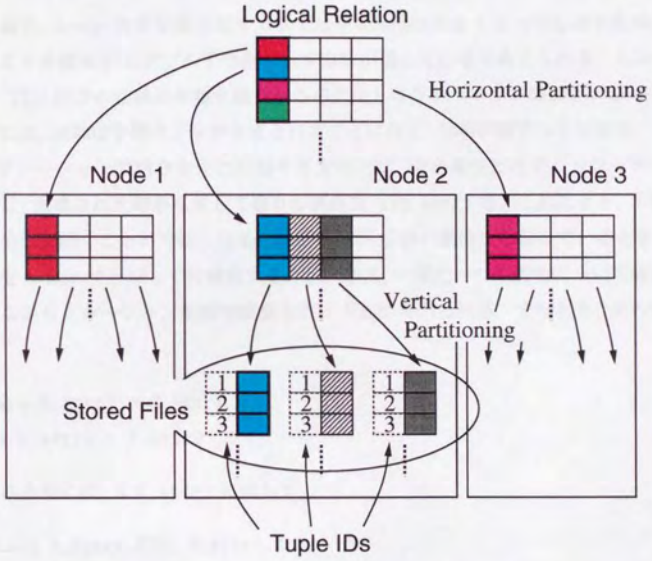


図 7.2: トランスポーズドファイルの物理レイアウト

両者のどちらを先に処理するかに応じて, 以下の 2 通りの方式が考えられる.

1. TID 結合 ⇒ リレーション間結合

2. リレーション間結合 ⇒ TID 結合

TID 結合を先に処理する方式では, 通常の射影演算を TID 結合で置き換えた形となるため, 全体の実行プランが影響を受けないという利点がある. そのため, 問合せオプティマイザにとってはトランスポーズドファイルの存在は透過的である. また, N 個の属性値を TID 結合する場合, 2-way 結合を繰り返す方法では中間結果が大きくなってしまうため, N-way マージにより各属性を 1 タプルずつ取り出す方法が適していると考えられる. しかし, 選択率が高く TID 結合の結果が多数生成されるのに比してリレーション間結合における結合率が低い時には, 無駄な中間タプルが生成されることになり, 効率が低下してしまう.

一方, リレーション間結合を先に処理する方式では, 結合属性だけでハッシュテーブルをプローブし, 生成された結果に対して残りの属性を TID 結合する. これにより, 不要な中間タプルの生成を防ぐことが可能となる. 結合キーが n 個の属性からなっているときは, 結合キーだけを n-way マージして再構成する方法に加え, 一度に一つの属性について結合条件をテストしながらリレーション間結合演算を行う方法が考えられる. すなわち, 元の結合条件が,

```
where R.Attr1 = S.Attr1
and R.Attr2 = S.Attr2
```

で与えられる時には, まず Attr1 に対して

```
select R_Attr1.TID, R_Attr1.Attr1
from R_Attr1, S_Attr1
where exists(R_Attr1.Attr1 = S_Attr1.Attr1)
```

を評価し, その結果に対して Attr2 を TID 結合することで不要な Attr1, Attr2 間の TID 結合を削除することができる.

7.4 トランスポーズドファイルを用いた性能評価

ここでは, TPC-D の問合せを例に取り, トランスポーズドファイルの問合せ処理に対する効果を示す. 図 7.3 はトランスポーズドファイル上での Query 9 の実行プランである.

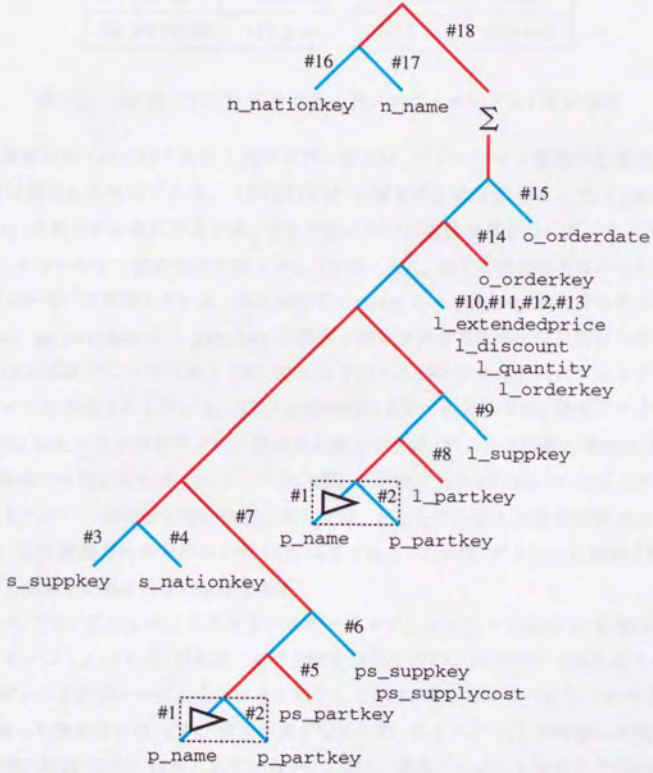


図 7.3: トランスポーズドファイル上での TPC-D Query 9 の実行プラン

	通常ファイル	トランスポーズド	
		TID 付き	TID 無し
DB 容量	131GB	173GB	131GB
Q9 I/O 量	128GB	39GB	24GB
Q9 実行時間	177.2 sec.	102.5 sec.	94.0 sec.

表 7.2: 100GB TPC-D に対するトランスポーズドファイルの効果

これは, 通常ファイルに対する図 5.10のプランを元に, リレーション間結合を優先して処理するように導出したものである. LINEITEM の結合時に結合条件として l_partkey と l_suppkey の両方が必要になるため, それぞれについて結合演算を行っている. TID 結合は細線で, リレーション間結合は太線で示している. また, 全ての結合演算はハッシュ結合アルゴリズムを用いて実装している. 図において p_name と p_partkey から作られるハッシュテーブルは, ps_partkey と l_partkey の両方と結合演算するために 2 回用いられる.

表 7.2 は大規模 PC クラスタ上での 100GB TPC-D ベンチマークに対するトランスポーズドファイルの効果を示している. TID を明示的に格納する方法では, 通常ファイル編成に比べて TID のために余分なディスク領域を必要とするが, ディスク容量の増加は 30% 以上であり, 無視できない大きさである. 一方, TID を隠蔽する方式においてはディスク容量の増加は見られない. 問合せ処理の性能に関しては, どちらの方法も大きな性能向上を達成しているが, TID 隠蔽方式の方がおよそ 10% 高速である. これは, ディスクに格納された TID を転送する処理を削減する事の効果である.

図 7.4 は, TID 隠蔽方式によるトランスポーズドファイル上で Query 9 を実行した時の実行時トレース (ノード 0) である. 実行全体を通じて CPU 利用率が 100% 近くに留まる一方で, ディスクのスループットは大きく低下しており, ボトルネックがディスク I/O から CPU に移った事が分かる. CPU 負荷が高くなるため, ディスク I/O の削減に比例した問合せ処理時間の短縮 (20%) は得られていない. しかし, 通常ファイルと比較して実行時間がおよそ 40% に短縮されており, 十分に大きな効果が得られたと言える.

図 7.4 のフェーズ #8 では, l_partkey と p_partkey との結合が行われるが, このフェーズでは読んだタプルが 100% 選択される上, 結果を元のノードに送り返すためのフィールドが付加されるため, タプル長が長くなって送出される. そのため, 処理負荷は極めて重く, ディスクの実効スループットは 1 MBytes/sec 以下にまで低下してしまう. そこで, これを改善するために, p_name と p_partkey のタプル ID 結合の結果をノード間でハッシュ分割す

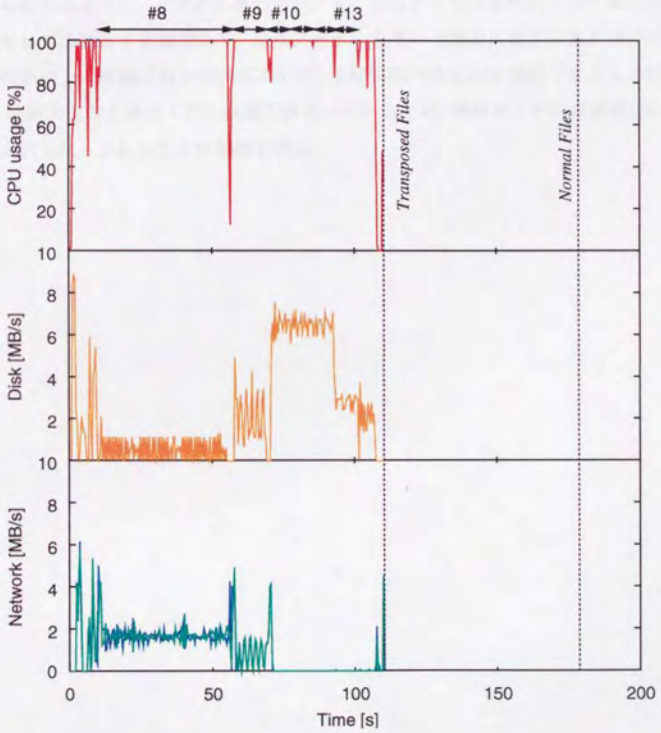


図 7.4: トランスポーズドファイル上での TPC-D Query 9 の実行時トレース

る代わりに, 全てのノードに broadcast するという方法を考えた. この方法では, p_partkey のハッシュテーブルが単一ノードのメモリに収まる必要があり, メモリの利用効率という点では好ましくないが, その結果 l_partkey をローカルに結合できるようになり, 処理負荷が軽減されることが予想される. この方式で Query 9 を実行するとフェーズ #8 の時間が大幅に減少し, さらに 30 秒程度の高速化が達成された. この時の実行時トレースを図 7.5に示す.

以上から分かるように, トランスポーズドファイルはアクセス属性の少ない意思決定問合せ処理に対しては極めて有効であり, 通常ファイルを用いた場合にはディスクデバイスの高速化しか性能向上の手段がない状況において, 2 倍程度の高速化を達成できることが分かった. また, 性能向上の上限は CPU 性能で決まっているため, 将来の CPU の高速化によりさらなる向上がもたらされることが期待される.

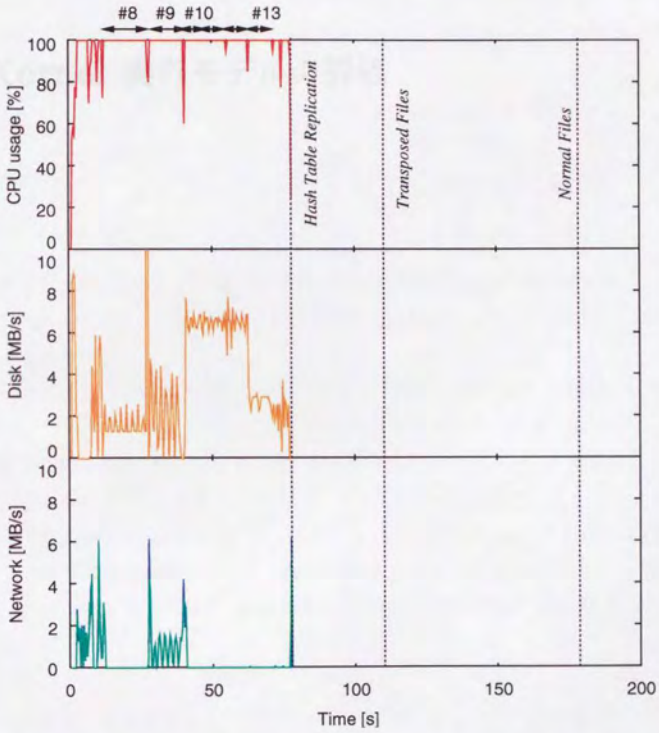


図 7.5: トランスポーズドファイル上での TPC-D Query 9 の実行時トレース (テーブル複製を行う場合)

第 8 章

DBKernel 実行モデルの評価

8.1 I/O 共有による複数問合せ実行時の性能向上

これまでの議論では, シングルユーザ環境を仮定して性能評価を行ってきたが, DBKernel はマルチユーザ環境も考慮して設計されている. マルチユーザ環境下では, 複数の問合せが同時に実行されるため, 個々の問合せ実行時間は長くなる可能性があるが, 複数の問合せが同じデータベースにアクセスする場合には, 問合せ間に共通する資源を共有することにより, 全体の処理終了時間が問合せを逐次実行した場合に比べて短縮されることが期待される. このような共有可能な資源としては,

- 入力リレーション
- ハッシュテーブル
- 問合せの中間結果

などが考えられるが, 後の 2 つに関しては問合せ実行時に検出するのは困難であり, また, 共有できる可能性も低いことから, ここでは複数問合せ間で同一リレーションの入力を共有することだけを考える.

同じデータを繰り返し読み込むのを避けるために一般的に取られている手法は, 図 8.1 に示す共有バッファ方式である. この方法は, アクセスされるデータに参照局所性が存在する場合は非常に有効な手法であるが, 意思決定問合せ処理のようにバッファ容量を越える I/O が頻繁に起こる場合には, 必要なデータがバッファから追い出されてしまうことによる性能低下が発生することもあり, 適当ではない. また, 2 つのプロセスがほぼ同時期に同一リレーションに対するアクセスを行っても, 一方のプロセスが読み込んだデータがバッファに保持されている期間内にもう一方のプロセスが同一データを読まない限りバッファの効果は存在しない.

そこで, DBKernel ではデータ駆動モデルの利点を生かし, ディスクからデータが読み込まれるごとにそのデータを必要とする複数のプロセスに渡すことでデータが共有される確率を高めている. データ駆動モデルでは, ファイルに対する I/O 要求はそのファイルを最初にオープンしたプロセスがまとめて発行し, プロセスはディスクから送られるデータを受動的に処理する (図 8.2). そのため, ディスクアクセスの順序はプロセス側は全く指示しないことになる. これによって, 2 つめのプロセスが既にオープンされているファイルをオープンした時は, 最初に到着するデータから処理を開始することになり, ファイルの先頭からの処理ではなくなる. この場合, 一度処理したデータを受け取る可能性があるため, それぞれのプロ

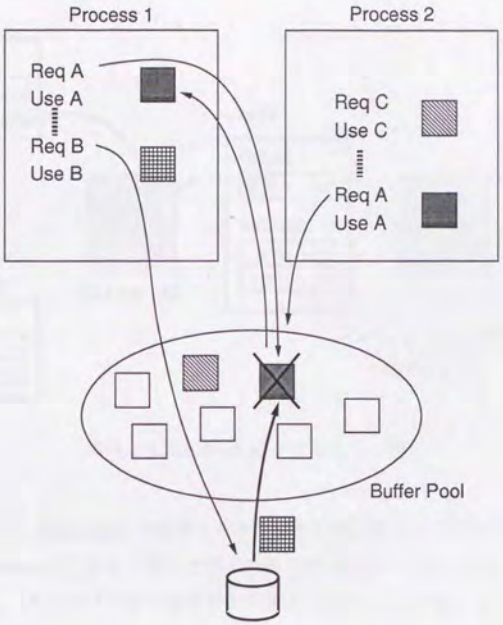


図 8.1: 共有バッファ方式

セスは既に処理したブロックに関するデータを保持しており, データの処理をスキップしたり, End-of-File の検出に用いている.

上記の方法により, 複数プロセスからの同一リレーションへのアクセスが少しでも重なれば入力の共有が起こるが, そうでない場合は異なるリレーションに対して逐次的にアクセスを行うことになる. そのため, 非常に大きなリレーションの読み込みを行っている間, 他のリレーションへのアクセスがブロックされる可能性がある. しかし, 大きなリレーションのアクセスをプリエンプトして小さなリレーションを先に読み込んでしまうことで, その後に再び大きなリレーションに対するアクセスが発行される場合にはそのリレーションが共有される可能性が高まるため, DBKernel ではリレーションの大きさに基づいたプリエンプションをサポートしている.

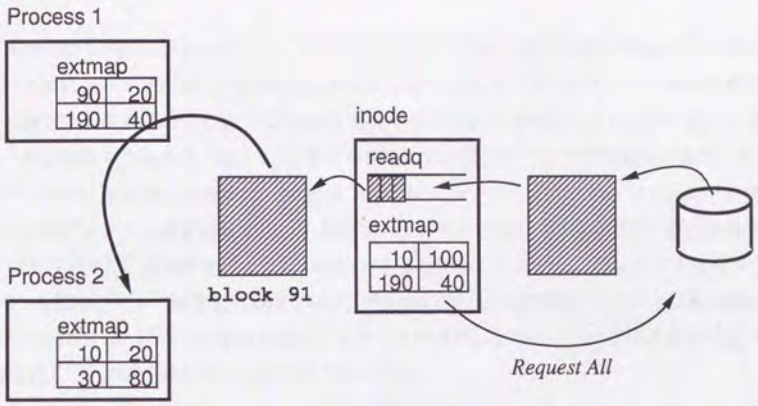


図 8.2: DBKernel における I/O 共有

既に述べたように, DBKernel ではディスクに対する非同期 I/O を実現するために専用のスレッド Disk manager によって全てのディスク I/O を行っており, Disk manager に対してキューを介してコマンドパケットを送付する事で, I/O を要求する. コマンドには以下のものが用意されている.

BIO ブロック単位の read/write.

ファイルシステムの管理情報へのアクセスや同期 I/O に用いる. Unix における buffer 構造体に類似の構造体を用いてアクセスブロック, read/write の別, 主記憶上のバッファアドレスなどを指定する.

READ_EXT 連続ブロックに対する read.

ファイル全体のフルスキャンに用いる. ファイルのディスクブロックマップ (extent 形式) とファイルに対して用意されたプリフェッチキューのアドレスを指定する. 読み込んだデータを格納するバッファは, Disk manager によって動的に割り当てられ, プリフェッチキューに挿入される.

WRITE_EXT 連続ブロックに対する write.

Delayed write で主記憶上にバッファされた連続ブロックをディスクに書き出すために用いる. ファイルのディスクブロックマップ (extent 形式) と書き出すページのリス

トを指定する.

典型的な Unix のディスクデバイスドライバにおいては, I/O 要求はブロックアドレスのシーク順にソートされる. この方法は, 全ての I/O が固定長で要求がブロック単位で発行される場合はうまく機能するが, DBKernel においては大きな連続ブロックに対するアクセスを一度に要求できるため, 他の I/O 要求を非常に長く待たせてしまう可能性がある. 特に, フリーメモリが枯渇した時には, delayed write によってメモリ上にバッファされているブロックをディスクに書き出してメモリを回収する必要があるが, READ_EXT 要求の処理中は WRITE_EXT 要求が受け付けられないとすると簡単にデッドロックに陥ってしまう. そこで, 上記のコマンド毎に優先順位を付け, プリエンプションを可能にしている. 優先順位は BIO $\geq$ WRITE_EXT $>$ READ_EXT となっているが, さらにファイルの大きさに応じて READ_EXT の優先順位が 2 つに分けられている.

ストリーム	問合せの発行順序
ストリーム 1	8, 2, 10, 14, 12, 17, 9, 3, 13, 6, 15, 11, 4, 1, 16, 5, 7
ストリーム 2	13, 15, 6, 4, 12, 14, 10, 1, 9, 17, 3, 7, 11, 5, 2, 8, 16
ストリーム 3	7, 3, 11, 17, 1, 16, 10, 9, 6, 8, 4, 2, 15, 5, 13, 12, 14
ストリーム 4	6, 11, 9, 1, 14, 15, 16, 3, 2, 12, 17, 8, 10, 7, 13, 5, 4

表 8.1: TPC-D ベンチマーク各ストリームにおける問合せの実行順序

TPC-D では, マルチユーザ環境での評価も考慮されており, 表 8.1に示す順序でそれぞれのユーザが問合せを発行するものと規定している. 4 つのストリームを同時に実行した時のディスクアクセスのトレースを図 8.3に示す. TPC-D では LINEITEM が特に大きく, これが共有されることで全体の処理時間が大幅に短縮されている. 図 8.4 と 図 8.5は, ストリーム数を変化させた時の全体の処理時間と逐次実行と比較した場合の性能向上を示している. ストリーム数を 4 より多くした場合, ハッシュテーブルを構築するためのメモリが不足するため, 4 を超えるストリームに関しては逐次的に実行するものとしている. プリエンプションを行わない場合でも I/O 共有の効果は得られているが, LINEITEM に対するプリエンプションを可能にするようにパラメータを設定することでさらなる性能向上が得られている. この場合, 逐次実行に比べて最大で 2.5 倍の性能向上が得られ, 大規模意思決定問合せ処理においては I/O 共有機構が有効に働くことを示している.

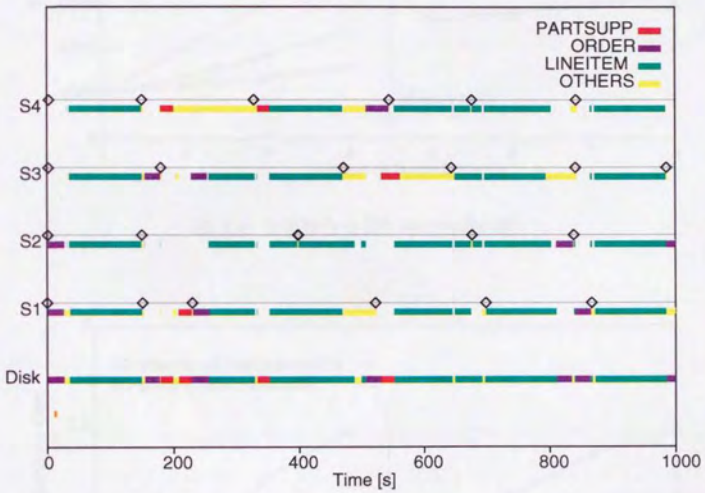


図 8.3: TPC-D 4 ストリーム同時実行時のディスクアクセストレース

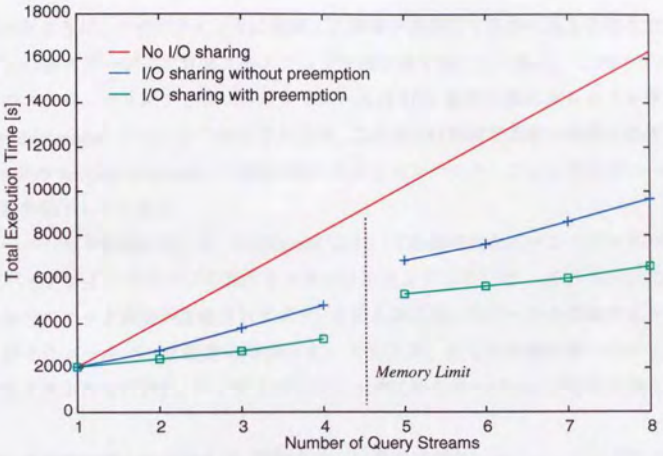


図 8.4: 複数問合せ実行時の処理時間

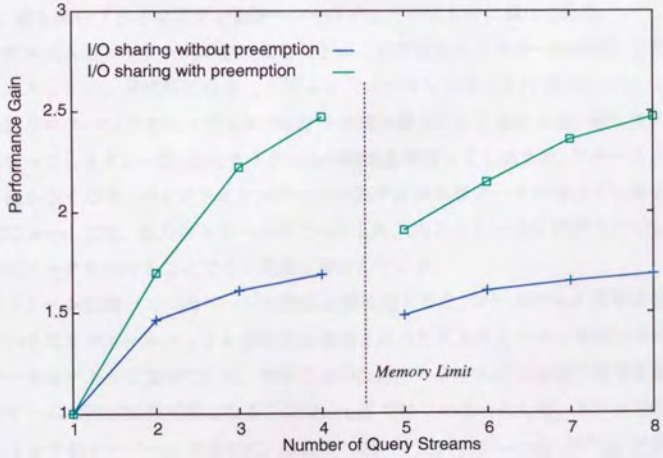


図 8.5: 複数問合せ実行時の I/O 共有による性能向上

8.2 複数演算子の単一コンテクスト内での実行による効果

既に述べたように, パイプライン上に連続した演算子を実行するのに最も自然な方法はパイプラインの各ステージにプロセスやスレッドを割り当てることである. このモデルでは, スレッドのスケジューリングとフローコントロールはI/O 操作の際にスレッドを停止 / 再開 (suspend/resume) することで達成されるが, この方法は単純で柔軟な機構を提供する一方, スレッドの suspend/resume の頻度が高くなるとコンテクストスイッチのオーバヘッドにより性能が低下してしまう.

このオーバヘッドを避けるため, DBKernel においては集中的なイベントディスパッチャを用いてパイプラインステージの実行をスケジューリングしている. 各ステージには入力に対するコールバック関数が登録されており, その入力に対してデータが到着するとディスパッチャがそのコールバック関数を呼び出す. そのため, 全ての処理は単一のディスパッチャコンテクスト内で行われ, コンテクストスイッチに伴うオーバヘッドを取り除くことができる.

このデータ駆動モデルにおいては, 複数ステージ間の適切なスケジューリングはステージ毎の入力ストリームにパイプラインの構造を反映したプライオリティをつけることで実現できる. プライオリティは, ディスクなどの一次入力側から最終出力側に向かって高くなり, ステージの途中にデータが集中しないようにしている. いくつかの入力に同時にデータが存在する時は, 最も高いプライオリティを持つコールバックがはじめに実行される.

このモデルは入力ストリームが空になることによるフローコントロールは扱うことができるが, 出力ストリームが満杯になることによるフローコントロールは扱いにくい. コールバック関数がサスペンドするとディスパッチャ自体が停止してしまうため, 他のステージの実行も止まってしまう. 一方, 出力ストリームの状態を無視してしまうと, フローコントロールは全く働かなくなり, パイプラインステージの途中に未処理データが溜ってしまうことになる. DBKernel では, 出力ストリームをフルにした入力ストリームを再びスケジュールしないようにマークを付けることでこの問題を解決している.

出力ストリームに溜っているページの数が上限に達すると, コールバック関数はサスペンドする代わりにそのコールバックを実行する原因となった入力ストリームを実行不可にしてからエラーを返すようになっている. やがてその出力ストリームに十分空きができるとその入力ストリームは再び実行可能になる. DBKernel ではこのようにして, スレッドのコンテクストスイッチのオーバヘッドなしに, 複数のステージをスケジューリングし, フローコントロールを実現している. 図 8.6 は 3 段のパイプラインが実行されている時のシステムのスナップショットである.

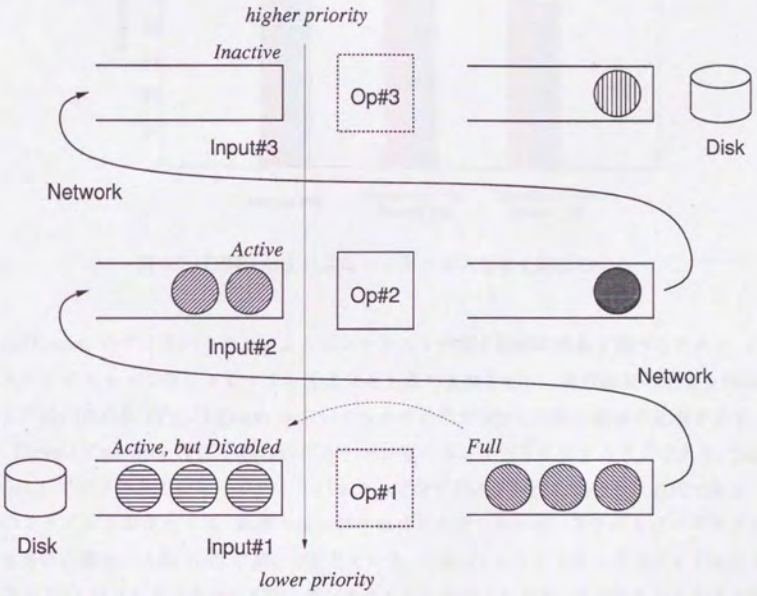


図 8.6: DBKernel のディスパッチャ

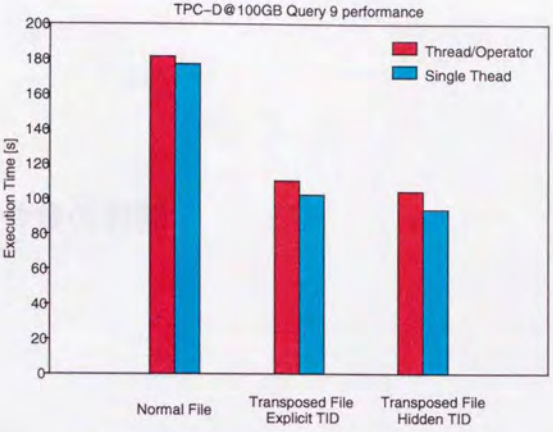


図 8.7: DBKernel によるコンテクスト切替え抑制の効果

DBKernel のディスパッチャによるコンテクスト切替え抑制の効果を調べるために, パイプラインのステージ毎にスレッドを生成する方式の実装を行い, 実行時間の比較を行った. 図 8.7 に, 100GB TPC-D Query 9 をいくつかの条件で実行した時の両者の比較を示す. 図の Thread/Operator で示されるのがステージ毎にスレッドを生成する方式であり, Single Thread で示されるのが単一のディスパッチャで全てのステージを実行する方式である. 通常のファイル上の実行では, 両者の違いはそれほど目立たないが, トランスポーズドファイルを用いた場合には約 10% の違いが生じている. これは, トランスポーズドファイルにより処理が I/O ボトルネックから CPU ボトルネックに変わったため, コンテクストスイッチのオーバヘッドが顕在化したためと考えられる.

第 9 章

まとめと今後の展開

9.1 本論文のまとめ

本研究では, 高速な大規模データベース解析を可能にする並列関係データベース処理系を構築し, 標準ハードウェアから成る大規模 PC クラスタを主対象として実装と性能評価を行った.

まず, SDC プロジェクトの最終的な評価を行い, 得られた成果を継承・発展可能なものにするために, さまざまな問合せの実行や異なるプラットフォームへの移植を前提として SDC システムソフトウェアの再設計を行い, データ駆動と Function shipping モデルを特徴とする並列問合せ処理系 DBKernel として提示した. そして, DBKernel を SDC-II 上に実装し, 業界標準ベンチマーク TPC-D (1GB/10GB) を用いた性能評価を行うことにより, 最先端の商用機と比較した SDC-II の位置づけを明らかにした. SDC-II はハードウェアの性能に比して高い性能を達成していたものの, 開発に多大な時間を要したため, 絶対性能では大きな格差が生じている事が分かった.

次に, 100 台の PC と 1 台の ATM スイッチからなる大規模 PC クラスタを構築し, 標準的な Unix 上で動作する DBKernel の実装を行った. また, 100GB の TPC-D ベンチマークによる性能評価の結果, 最も高性能な既存商用システムに対しても最高で 5 倍以上の性能を達成していることが分かった. これにより, PC クラスタの有効性が単に性能対価格比だけに留まらず絶対性能においても充分存在すること, 問合せ処理系の改善の効果が大きいことを示すことができた.

現実のデータにおいては分布に偏り (スキュー) が存在するため, 台数効果を維持するために不可欠となる負荷分散機構について論じた. SDC では Redistribution Skew に属するスキューをアルゴリズムとハードウェアの協調により解決している. PC クラスタにおいてはハードウェアによる解決法は採らないが, より高度な問題である Join Product Skew を処理するための実行時負荷分散機構を実装・評価し, その有効性を示した.

また, さらなる性能向上を達成すべく, 物理ファイル編成方式にトランスポーズドファイルを適用した. トランスポーズドファイルは, リレーションを属性毎に独立したファイルに格納する技法であり, 多数の属性から成るリレーションに対して一部の属性に関する解析を行う意思決定問合せにおいては, I/O 量の削減によるディスクボトルネックの解消という効果が期待できる. しかし, 分離された属性を再び結合する処理が増えるため, 結合演算の効率が重要になる. PC クラスタ上でトランスポーズドファイル編成を 100GB TPC-D ベンチマークの問合せに対して適用する事により, 通常ファイル編成に比べて 2 倍程度の性能向上が達成された. この結果, トランザンクションを中心とする従来のデータベース応用ではそのオーバヘッドの大きさから利用されなかったトランスポーズドファイルは, 意思決定問合

せに対しては極めて有効である事が明らかになった.

DBKernel はマルチユーザ環境にも対応しており, 複数の問合せが同じデータベースに対して発行された時には, 同一ファイルへのアクセスを実行時に検出し, 問合せ間で I/O 処理を共有する機構により性能向上を図っている. また, ファイルの大きさに基づいてファイルアクセスの優先順位を設定する事により, 応答時間の長くなる大きなファイルの読み込みが共有される確率を高める工夫を施している. TPC-D で規定された複数問合せストリームの実験により, 同時に実行される問合せ数を増やした時に全体の処理時間の増加が抑えられ, システム全体としてのスループットが非常に高い事が明らかになった. また, 多数の演算子を単一のコンテクスト内で並行して評価する方法を各演算子に実行コンテクストを割当てた場合と比較することで, コンテクストスイッチを抑制することにより, 10% 程度の性能向上が得られていることを示した.

本研究の結果, 超並列関係データベースサーバ実現のために大規模 PC クラスタが極めて有望であること, および問合せ処理系改善の効果が大きいことが明らかになった.

9.2 今後の課題

DBKernel は大規模なシステムであるため, 多くの課題が残されている. 特に, 問合せ言語処理系における DML サポートについてはほとんど手付かずである. また, セキュリティを考慮して実行コードを中間言語で表現することなども重要であろう. また, 実行コードは問合せの都度生成されるのでその効率的配布法も考慮する必要がある.

実行エンジンの部分に付いては, 異なるプラットフォームへの移植をさらに進める必要がある. 特に, SMP クラスタでのノード内の並列性の抽出や, カーネルレベルでの実装による性能向上などは興味深い.

最後に, DBKernel で無視していた更新処理やインデックスを用いたアクセスなどがどのくらい DBKernel と親和性を持っているかを評価することも今後の重大な課題であろう.

謝辞

本研究を進めるに当たっては, 多くの方々の御指導, 御協力を賜わりました.

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わり, 研究を進めていく上での環境面についても多大な御配慮を頂きました. 本研究は, 二つの大規模な並列データベースシステムにおける, 実機上での性能評価を主眼とするものだったために, ややもすると現実の制約に捕らわれて大局的な目標を見失いがちでしたが, 喜連川先生の助言によって実装と研究とのバランスを取ることができました.

本研究は, SDC プロジェクトに携わった諸先輩の貴重な研究成果の上に成り立っています. 特に, 現電子技術総合研究所の平野聡氏と現三菱電機株式会社の中村稔氏が開発された初期の SDC システムソフトウェアは, 並列関係データベース処理系におけるさまざまな問題点とその対処法を把握するのに非常に参考になりました. 本研究の動機の一つには, 両氏の研究成果をより広範な環境と問題に対して適用可能にすることがありました. また, SDC-II ハードウェアの設計・製作に関しては, 富士通株式会社の小川義久氏に大変お世話になりました.

PC クラスタの構築は大規模なプロジェクトであり, 研究室や企業の多くの方々の御協力なしには成立しませんでした. 小口正人氏は PC クラスタの機種選定のための予備評価を根気良く行って下さいました. 実際のシステムが期待外れに終ることなく, 妥当な性能を達成することができたのは氏の努力の賜です. 助手の根本利弘氏は PC クラスタの設置場所に関する繁雑な事務処理を引き受けて下さいました. 日立製作所株式会社の方々には, 128 ポートという大規模な ATM スイッチに関するさまざまな問題に対して御支援を頂きました.

DBKernel の設計においては, BSD に関する 2 冊の本 [23, 24] とソースコードを頻繁に参照しました. DBKernel と BSD とで対応する概念が存在する時には, BSD の命名規則や時には実装そのものを利用させて頂きました. また, GNU や X Window System をはじめとするさまざまなフリーソフトウェアは快適な開発環境を整えるために不可欠でした. これらの有益なソフトウェアのソースコードを無償で公開して下さった作者の方々に深く感謝いたします.

助手の中野美由紀氏には並列データベース技術全般に関わるさまざまな討論を通じて貴重な助言を頂きました. Stephen Davis 氏には動的負荷分散アルゴリズムとそのシミュレーションに関する研究成果を利用させて頂きました. また, 国際学会へ論文を投稿する際に非常に熱心に英語を添削して下さいました. 安井隆宏君は発展途上の DBKernel に動的負荷分散アルゴリズムを実装するという難しい作業を見事にこなしてくれました. また, その作業を通じて DBKernel の設計や実装上の問題点を洗い出してくれました. 茂木和彦氏や根本氏は, 研究室の計算機環境の整備に多大な労力をつぎこんで下さいました.

経済的に苦労せずに研究生活を送れたのは, 日本学術振興会, 新エネルギー・産業技術総合開発機構 (NEDO), 東京大学生産技術研究所奨励会の御支援のお蔭です. 改めて感謝の意を表します.

最後に, 7 年間に及ぶ研究生活を常に温かい励ましと寛容の精神で支えてくれた久美に心から感謝します.

参考文献

[1] C. Baru, G. Fecteau, et al. An overview of DB2 parallel edition. In *Proceedings of International Conference on Management of Data*, pp. 460–462. ACM SIGMOD, 1995.

[2] D.S. Batory. On searching transposed files. *ACM Transactions on Database Systems*, Vol. 4, No. 4,, 1979.

[3] P.A. Boncz, W. Quak, and M.L. Kersten. Monet and its geographical extensions: A novel approach to high performance GIS processing. In *Proceedings of International Conference on Extending Database Technology*, pp. 147–166, 1996.

[4] F. Calino Jr. and P. Kostamaa. Exegesis of DBC/1012 and P-90. *Teradata Advanced Concepts Laboratory (TACL)*, 1992.

[5] R. Carter and J. Laroco. Commodity clusters: Performance comparison between PC's and workstations. In *Proceedings of International Symposium on High Performance Distributed Computing*, pp. 292–304. IEEE, 1995.

[6] M. Chen, M. Lo, P. Su, and H. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proceedings of International Conference on Very Large Data Bases*, pp. 15–26, 1992.

[7] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, Vol. 13, No. 6, pp. 377–387, 1970.

[8] Teradata Corp. DBC/1012 data base computer concepts & facilities. Technical Report C02-0001-05, Teradata Corp., 1988.

[9] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, 6th edition, 1995.

[10] D.J. DeWitt and R.H. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of International Conference on Very Large Data Bases*, 11th, pp. 151–164, 1985.

[11] D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna. GAMMA - A high performance dataflow database machine. In *Proceedings of International Conference on Very Large Data Bases*, 12th, pp. 228–237, 1986.

[12] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, pp. 44–62, 1990.

[13] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 2nd edition, 1994.

[14] G. Graefe, A. Linville, and L.D. Shapiro. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 6, pp. 934–944, 1994.

[15] G. Hallmark. Oracle parallel warehouse server. In *Proceedings of International Conference on Data Engineering*, pp. 314–320. IEEE, 1997.

[16] J. Heinanen. Multiprotocol encapsulation over ATM adaptation layer 5. Technical Report RFC1483, 1993.

[17] Information Networks Division Hewlett-Packard Company. Netperf: A network performance benchmark. Technical Report Revision 2.0, Hewlett-Packard Company, 1995. `http://www.cup.hp.com/netperf/NetperfPage.html`.

[18] M. Kitsuregawa, S. Hirano, M. Harada, M. Nakamura, T. Tamura, and M. Takagi. Overview of the Super Database Computer (SDC-I). *Transaction of The Institute of Electronics, Information and Communication Engineers*, Vol. E77-C, No. 7, pp. 1023–1031, 1994.

[19] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to data base machine and its architecture. *New Generation Computing*, Vol. 1, No. 1, pp. 66–74, 1983.

[20] K. Kojima, S. Torii, and S. Yoshizumi. IDP - a main storage based vector database processor. In *5th Intenational Workshop on Database Machines*, pp. 47–60, 1987.

[21] M. Laubach. Classical IP and ARP over ATM. Technical Report RFC1577, 1994.

[22] D.H. Lawrie. Access and alignment of data in an array processor. *Transactions on Computers*, Vol. C-24, No. 12, pp. 1145–1155, 1975.

[23] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[24] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[25] P. Mishra and M.H. Eich. Join processing in relational databases. *ACM Computing Surveys*, Vol. 24, No. 1, pp. 63–113, 1992.

[26] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proceedings of International Conference on Very Large Data Bases*, 14th, pp. 468–478, 1988.

[27] Red Brick Systems, Inc. Start schema processing for complex queries. White paper, Red Brick Systems, Inc., 1997.

[28] D.A. Schneider and D.J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of International Conference on Very Large Data Bases*, 16th, pp. 469–480, 1990.

[29] A. Shatdal and J.F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of International Conference on Management of Data*, pp. 104–114. ACM SIGMOD, 1995.

[30] E. Shekita, H. Young, and K.L. Tan. Multi-join optimization for symmetric multiprocessors. In *Proceedings of International Conference on Very Large Data Bases*, pp. 479–492, 1993.

[31] D Sloan. A practical implementation of the database machine - Teradata DBC/1012. In *Proceedings of Hawaii International Conference on Computer Sciences*, 25th. IEEE, 1992.

[32] T. Sterling, D. Saverese, D.J. Becker, B. Fryxell, and K. Olson. Communication overhead for space science applications on the Beowulf parallel workstation. In *Proceedings of International Symposium on High Performance Distributed Computing*, pp. 23–30, 1995.

[33] T. Tamura, M. Nakamura, M. Kitsuregawa, and Y. Ogawa. Implementation and performance evaluation of the parallel relational database server SDC-II. In *Proceedings of International Conference on Parallel Processing*, 25th, pp. I–212–I–221, 1996.

[34] Transaction Processing Performance Council (TPC)a. TPC Benchmark^{TM} D (Decision Support). Standard Specification Revision 1.1, TPC, 1995.

[35] C.B. Walton, A.G. Dale, and R.M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of International Conference on Very Large Data Bases*, 1991.

[36] Xilinx, Inc. *The Programmable Logic Data Book*. Xilinx, Inc., 1993.

[37] 井上, 速水, 福岡, 鈴木, 松永. データベースプロセッサ RINDA の設計と実現. 情報処理学会論文誌, Vol. 31, No. 3, pp. 373–380, 1990.

[38] 喜連川. 最近のデータベースプロセッサの商用化並びに研究開発の動向. 情報処理, Vol. 33, No. 12, pp. 1388–1402, 1992.

[39] 工藤, 他. データベース処理のハードウェア化. 情報処理学会研究報告, number 86 in 91-ARC-90, 1991.

[40] 松田, 東郷, 島川, 岩崎. データベース演算処理装置のアーキテクチャ. 電子情報通信学会技術研究報告, 1991.

[41] 中村. 並列関係問合せ処理の実行方式とその実装並びに性能評価に関する研究. PhD thesis, 東京大学, 1995.

[42] 中村, 平野, 田村, 喜連川, 高木. スーパーデータベースコンピュータ SDC-II におけるシステムソフトウェアの設計と実装. 電子情報通信学会論文誌, Vol. J78-D-I, No. 2, pp. 129–141, 1995.

[43] 田村, 喜連川, 高木. 並列 SQL サーバ SDC-II の TPC-D ベンチマークを用いた 性能評価. 電子情報通信学会 データ工学研究会 *DE96-31*, pp. 43–48, 1996.

[44] 田村, 中村, 喜連川, 高木. スーパーデータベースコンピュータ (SDC) のパケット平坦化ネットワークにおける縮退動作支援アルゴリズムとその評価. 情報処理学会研究報告 *ARC-95-16*, 1992.

[45] 平野. 高並列関係データベースサーバに於けるシステムソフトウェアの研究. PhD thesis, 東京大学, 1992.

[46] 伏見, 岩崎, 安東, 小宮, 樋口. 高速ハードウェアソータを用いた SQL システムの実現. pp. 1–8, 1991.

[47] 平野, 原田, 中村, 小川, 楊, 喜連川, 高木. スーパーデータベースコンピュータ SDC のアーキテクチャ. 並列処理シンポジウム *JSPP'90*, pp. 137, 1990.

付録 A

TPC-D ベンチマーク問合せ

```
-- Q1 - pricing summary report query

select L_Returnflag, L_Linestatus, sum(L_Quantity) as Sum_Qty,
       sum(L_Extendedprice) as Sum_Base_Price,
       sum(L_Extendedprice * (1 - L_Discount)) as Sum_Disc_Price,
       sum(L_Extendedprice * (1 - L_Discount) * (1 + L_Tax)) as Sum_Charge,
       avg(L_Quantity) as Avg_Qty,
       avg(L_Extendedprice) as Avg_Price,
       avg(L_Discount) as Avg_Disc,
       count(*) as Count_Order
from lineitem
where   L_Shipdate <= date '1998-12-01' - interval '90' day
group by L_Returnflag, L_Linestatus
order by L_Returnflag, L_Linestatus;
```

図 A.1: TPC-D Query 1

```
-- Q2 - minimum cost supplier query

select S_Acctbal, S_Name, N_Name, P_Partkey, P_Mfgr,
       S_Address, S_Phone, S_Comment
from part P, supplier S, partsupp PS, nation, region
where   P_Partkey = PS_Partkey
    and S_Suppkey = PS_Suppkey
    and P_Size = 15
    and P_Type like '%BRASS'
    and S_Nationkey = N_Nationkey
    and N_Regionkey = R_Regionkey
    and R_Name = 'EUROPE'
    and PS_Supplycost =
        (select min(PS_Supplycost)
         from partsupp PS1, supplier S1, nation N1, region R1
         where  P.P_Partkey = PS1.PS_Partkey
            and S1.S_Suppkey = PS1.PS_Suppkey
            and S1.S_Nationkey = N1.N_Nationkey
            and N1.N_Regionkey = R1.R_Regionkey
            and R1.R_Name = 'EUROPE'
        )
order by S_Acctbal desc, N_Name, S_Name, P_Partkey;

-- return first 100 rows
```

図 A.2: TPC-D Query 2

```
-- Q3 - shipping priority query

select L_Orderkey, sum(L_Extendedprice * (1 - L_Discount)) as Revenue,
       O_Orderdate, O_Shippriority
from customer, "order", lineitem
where   C_Mktsegment = 'BUILDING'
    and C_Custkey = O_Custkey
    and L_Orderkey = O_Orderkey
    and O_Orderdate < date '1995-03-15'
    and L_Shipdate > date '1995-03-15'
group by L_Orderkey, O_Orderdate, O_Shippriority
order by Revenue desc, O_Orderdate;

-- return first 10 rows
```

図 A.3: TPC-D Query 3

```
-- Q4 - order priority checking query

select O_Orderpriority, count(*) as Order_Count
from "order"
where   O_Orderdate >= date '1993-07-01'
    and O_Orderdate < date '1993-07-01' + interval '3' month
    and exists (select *
                from lineitem
                where   L_Orderkey = O_Orderkey
                    and L_Commitdate < L_Receiptdate)
group by O_Orderpriority
order by O_Orderpriority;
```

図 A.4: TPC-D Query 4

```
-- Q5 - local supplier volume query

select N_Name, sum(L_Extendedprice * (1 - L_Discount)) as Revenue
from customer, "order", lineitem, supplier, nation, region
where   C_Custkey = O_Custkey
    and L_Orderkey = O_Orderkey
    and L_Suppkey = S_Suppkey
    and C_Nationkey = S_Nationkey
    and S_Nationkey = N_Nationkey
    and N_Regionkey = R_Regionkey
    and R_Name = 'ASIA'
    and O_Orderdate >= date '1994-01-01'
    and O_Orderdate < date '1994-01-01' + interval '1' year
group by N_Name
order by Revenue desc;
```

図 A.5: TPC-D Query 5

```
-- Q6 - forecasting revenue change query

select sum(L_Extendedprice * L_Discount) as Revenue
from lineitem
where   L_Shipdate >= date '1994-01-01'
    and L_Shipdate < date '1994-01-01' + interval '1' year
    and L_Discount between 0.06 - 0.01 and 0.06 + 0.01
    and L_Quantity < 24;
```

図 A.6: TPC-D Query 6

```
-- Q7 - volume shipping query

select Supp_Nation, Cust_Nation, "Year", sum(Volume) as Revenue
from    (select N1.N_Name as Supp_Nation,
                N2.N_Name as Cust_Nation,
                extract(year from L_Shipdate) as "Year",
                L_Extendedprice * (1 - L_Discount) as Volume
         from supplier, lineitem, "order", customer, nation N1, nation N2
         where  S_Suppkey = L_Suppkey
            and O_Orderkey = L_Orderkey
            and C_Custkey = O_Custkey
            and S_Nationkey = N1.N_Nationkey
            and C_Naitonkey = N2.N_Nationkey
            and ((N1.N_Name = 'FRANCE' and N2.N_Name = 'GERMANY')
                 or (N1.N_Name = 'GERMANY' and N2.N_Name = 'FRANCE'))
            and L_Shipdate between date '1995-01-01' and date '1996-12-31'
        )
group by Supp_Nation, Cust_Nation, "Year"
order by Supp_Nation, Cust_nation, "Year";
```

図 A.7: TPC-D Query 7

```
-- Q8 - national market share query

select "Year", sum(case when Nation = 'BRAZIL'
                        then Volume
                        else 0
                   end) / sum(Volume) as Mkt_Share
from    (select extract(year from O_Orderdate) as "Year",
                L_Extendedprice * (1 - L_Discount) as Volume,
                N2.N_Name as Nation
        from part, supplier, lineitem, "order", customer, nation N1, nation N2,
                region
        where   P_Partkey = L_Partkey
            and S_Suppkey = L_Suppkey
            and L_Orderkey = O_Orderkey
            and O_Custkey = C_Custkey
            and C_Nationkey = N1.N_Nationkey
            and N1.N_Regionkey = R_Regionkey
            and R_Name = 'AMERICA'
            and S_Nationkey = N2.N_Nationkey
            and O_Orderdate between date '1995-01-01' and date '1996-12-31'
            and P_Type = 'ECONOMY ANODIZED STEEL')
group by "Year"
order by "Year";
```

図 A.8: TPC-D Query 8

```
-- Q9 - product type profit measure query

select Nation, "Year", sum(Amount) as Sum_Profit
from    (select N_Name as Nation, extract(year from O_Orderdate) as "Year",
         L_Extendedprice * (1 - L_Discount) - PS_Supplycost * L_Quantity
                as Amount
         from part, supplier, lineitem, partsupp, "order", nation
         where  S_Suppkey  = L_Suppkey
            and PS_Suppkey = L_Suppkey
            and PS_Partkey = L_Partkey
            and P_Partkey  = L_Partkey
            and O_Orderkey = L_Orderkey
            and S_Nationkey = N_Nationkey
            and P_Name like '%green%'
        )
group by Nation, "Year"
order by Nation, "Year" desc;
```

図 A.9: TPC-D Query 9

```
-- Q10 - returned item query

select C_Custkey, C_Name, sum(L_Extendedprice * (1 - L_Discount)) as Revenue,
        C_Acctbal, N_Name, C_Address, C_Phone, C_Comment
from customer, "order", lineitem, nation
where   C_Custkey = O_Custkey
    and L_Orderkey = O_Orderkey
    and O_Orderdate >= date '1993-10-01'
    and O_Orderdate < date '1993-10-01' + interval '3' month
    and L_Returnflag = 'R'
    and C_Nationkey = N_Nationkey
group by C_Custkey, C_Name, C_Acctbal, C_Phone, N_Name, C_Address, C_Comment
order by Revenue desc;

-- return first 20 rows
```

図 A.10: TPC-D Query 10

```
-- Q11 - important stock identification query

select PS_Partkey, sum(PS_Supplycost * PS_Availqty) as "Value"
from partsupp, supplier, nation
where   PS_Suppkey = S_Suppkey
    and S_Nationkey = N_Nationkey
    and N_Name = 'GERMANY'
group by PS_Partkey
having sum(PS_Supplycost * PS_Availqty) >
       (select sum(PS_Supplycost * PS_Availqty) * 0.001
        from partsupp, supplier, nation
        where  PS_Suppkey = S_Suppkey
           and S_Nationkey = N_Nationkey
           and N_Name = 'GERMANY'
       )
order by "Value" desc;
```

図 A.11: TPC-D Query 11

```
-- Q12 - shipping modes and order priority query

select L_Shipmode,
        sum(case when   O_Orderpriority = '1-URGENT'
                     or O_Orderpriority = '2-HIGH'
                then 1
                else 0
            end) as High_Line_Count,
        sum(case when   O_Orderpriority <> '1-URGENT'
                    and O_Orderpriority <> '2-HIGH'
                then 1
                else 0
            end) as Low_Line_Count
from "order", lineitem
where   O_Orderkey = L_Orderkey
    and L_Shipmode in ('MAIL', 'SHIP')
    and L_Commitdate < L_Receiptdate
    and L_Shipdate < L_Commitdate
    and L_Receiptdate >= date '1994-01-01'
    and L_Receiptdate < date '1994-01-01' + interval '1' year
group by L_Shipmode
order by L_Shipmode;
```

図 A.12: TPC-D Query 12

```
-- Q13 - sales clerk performance query

select "Year", sum(Revenue) as Revenue
from    (select extract(year from O_Orderdate) as "Year",
                L_Extendedprice * (1 - L_Discount) as Revenue
        from lineitem, "order"
        where   O_Orderkey = L_Orderkey
            and O_Clerk = 'Clerk#000000088'
            and L_Returnflag = 'R'
        )
group by "Year"
order by "Year";
```

図 A.13: TPC-D Query 13

```
-- Q14 - promotion effect query

select 100.00 * sum(case when P_Type like 'PROMO%'
                         then L_Extendedprice * (1 - L_Discount)
                         else 0
                    end) / sum(L_Extendedprice * (1 - L_Discount))
                as Promo_Revenue
from lineitem, part
where   L_Partkey = P_Partkey
    and L_Shipdate >= date '1995-09-01'
    and L_Shipdate < date '1995-09-01' + interval '1' month;
```

図 A.14: TPC-D Query 14

```
-- Q15 - top supplier query

create view revenue_1 (Supplier_No, Total_Revenue) as
        select L_Suppkey, sum(L_Extendedprice * (1 - L_Discount))
        from lineitem
        where   L_Shipdate >= date '1996-01-01'
            and L_Shipdate < date '1996-01-01' + interval '3' month
        group by L_Suppkey;

select S_Suppkey, S_Name, S_Address, S_Phone, Total_Revenue
from supplier, revenue_1
where   S_Suppkey = Supplier_No
    and Total_Revenue =
        (select max(Total_Revenue) from revenue_1)
order by S_Suppkey;

drop view revenue_1 cascade;
```

図 A.15: TPC-D Query 15

```
-- Q16 - part/supplier relationship query

select  P_Brand, P_Type, P_Size, count(distinct PS_Suppkey) as Supplier_Cnt
from partsupp, part
where   P_Partkey = PS_Partkey
    and P_Brand <> 'Brand#45'
    and P_Type not like 'MEDIUM POLISHED%'
    and P_Size in (49, 14, 23, 45, 19, 3, 36, 9)
    and PS_Suppkey not in
        (select S_Suppkey
         from supplier
         where S_Comment like '%Better Business Bureau%Complaints%')
group by P_Brand, P_Type, P_Size
order by Supplier_Cnt desc, P_Brand, P_Type, P_Size;
```

図 A.16: TPC-D Query 16

```
-- Q17 - small-quantity-order revenue query

select sum(L_Extendedprice) / 7.0 as Avg_Yearly
from lineitem, part
where   P_Partkey = L_Partkey
    and P_Brand = 'Brand#23'
    and P_Container = 'MED BOX'
    and L_Quantity <
        (select 0.2 * avg(L_Quantity)
        from lineitem L1
        where L1.L_Partkey = P_Partkey);

-- return first 1 row
```

図 A.17: TPC-D Query 17

付録 B

100 GB TPC-D ベンチマーク問合せ実行プラン

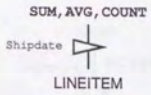


図 B.1: TPC-D Query 1 の実行プラン

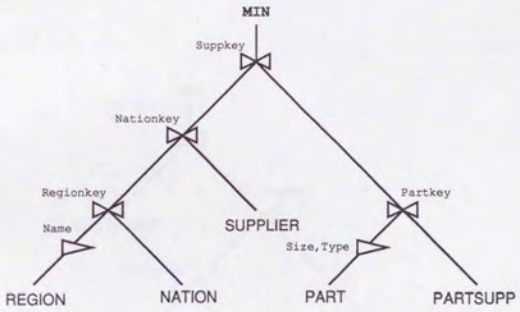


図 B.2: TPC-D Query 2 の実行プラン

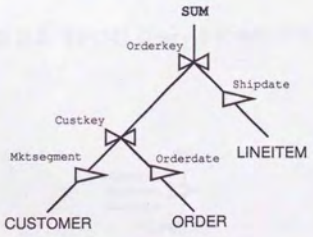


図 B.3: TPC-D Query 3 の実行プラン

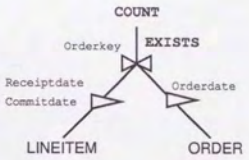


図 B.4: TPC-D Query 4 の実行プラン

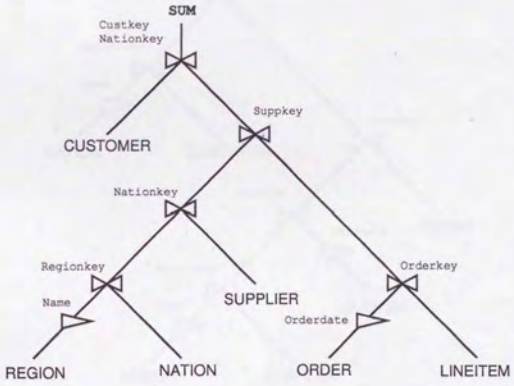


図 B.5: TPC-D Query 5 の実行プラン

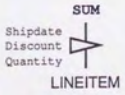


図 B.6: TPC-D Query 6 の実行プラン

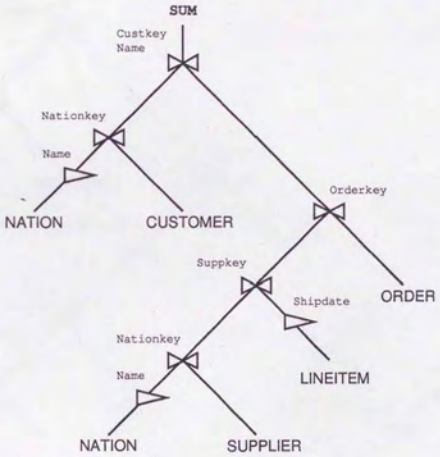


図 B.7: TPC-D Query 7 の実行プラン

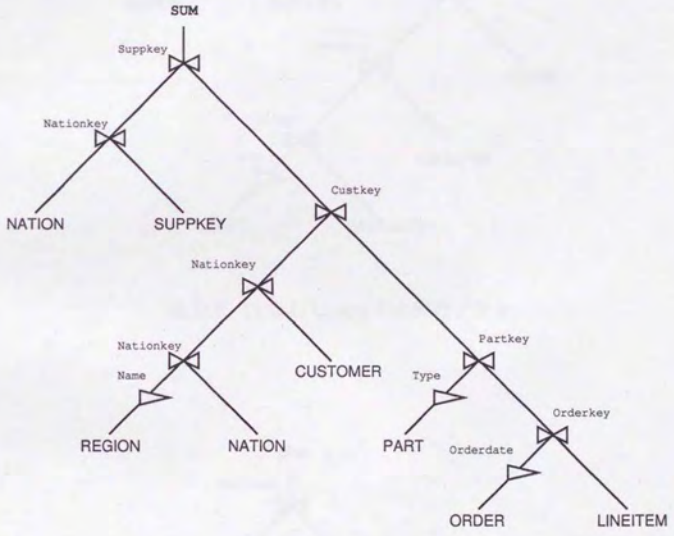


図 B.8: TPC-D Query 8 の実行プラン

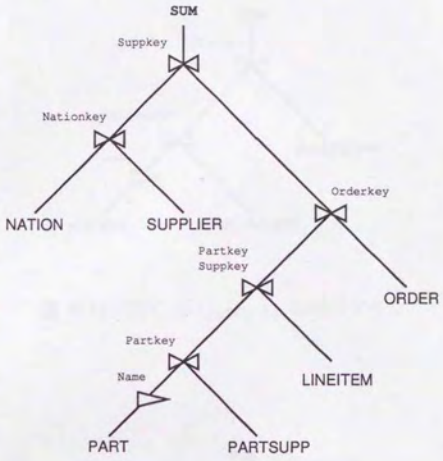


図 B.9: TPC-D Query 9 の実行プラン

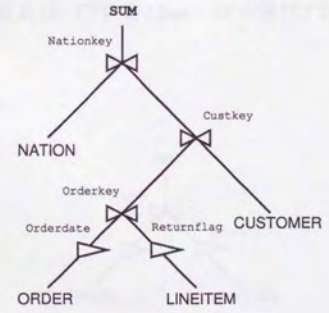


図 B.10: TPC-D Query 10 の実行プラン

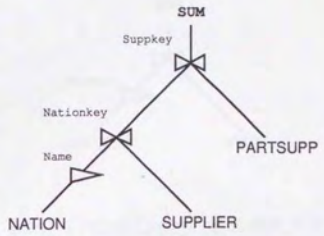


図 B.11: TPC-D Query 11 の実行プラン

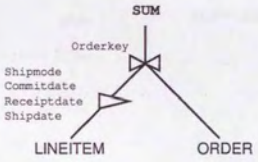


図 B.12: TPC-D Query 12 の実行プラン

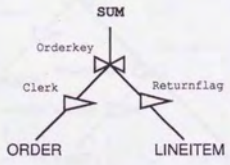


図 B.13: TPC-D Query 13 の実行プラン

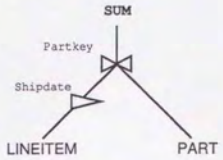


図 B.14: TPC-D Query 14 の実行プラン

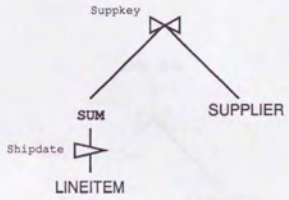


図 B.15: TPC-D Query 15 の実行プラン

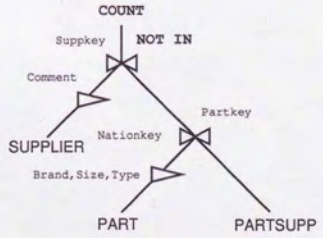


図 B.16: TPC-D Query 16 の実行プラン

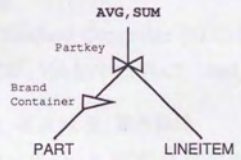


図 B.17: TPC-D Query 17 の実行プラン

付録 C

発表文献

論文誌

1. Masaru Kitsuregawa, Satoshi Hirano, Masanobu Harada, Minoru Nakamura, Takayuki Tamura, and Mikio Takagi:
   "Overview of the Super Database Computer (SDC-I)",
   電子情報通信学会英文論文誌, Vol.E77-C, No.7, 1994, pp.1023–1031.

2. 中村 稔, 平野 聡, 田村孝之, 喜連川 優, 高木幹雄:
   「スーパーデータベースコンピュータ SDC-II におけるシステムソフトウエアの設計と実装」,
   電子情報通信学会論文誌, Vol.J78-D-I, No.2, 1995, pp.129–141.

国際会議

1. Takayuki Tamura, Minoru Nakamura, Masaru Kitsuregawa, and Yoshihisa Ogawa:
   "Implementation and Performance Evaluation of the Parallel Relational Database Server SDC-II",
   In Proceedings of 25th International Conference on Parallel Processing (ICPP '96), I-212–I-221, August 1996.

2. Masato Oguchi, Takahiko Shintani, Takayuki Tamura, and Masaru Kitsuregawa:
   "Preliminary Experimental Results of a Parallel Association Rule Mining on ATM connected PC Clusters", In Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications (CODAS), pp.278–281, De-

cember 1996.

3. Takayuki Tamura and Masaru Kitsuregawa:
"Implementation and Evaluation of the Bucket Flattening Omega Network of the Parallel Relational Database Server SDC-II",
In Proceedings of the 5th International Conference on Database Systems for Advanced Applications (DASFAA '97), pp.471–480, April 1997.

4. Masato Oguchi, Takahiko Shintani, Takayuki Tamura, and Masaru Kitsuregawa:
"Characteristics of a Parallel Data Mining Application Implemented on an ATM Connected PC Cluster",
In Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe '97), April 1997.

5. Takayuki Tamura, Masato Oguchi, and Masaru Kitsuregawa:
"Parallel Database Processing on a 100 Node PC Cluster: Cases for Decision Support Query Processing and Data Mining",
In Proceedings of SC97: High Performance Networking and Computing (SuperComputing '97), November 1997.

6. Seigo Muto, Takayuki Tamura, Miyuki Nakano, and Masaru Kitsuregawa:
"Implementation and Evaluation of Parallel Relational Query Processing Using Transposed Files on Shared Memory Multiprocessors",
In Proceedings of International Symposium on Digital Media Information Base (DMIB'97), November 1997.

全国大会

1. 原田昌信, 田村孝之, 平野 聡, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ (SDC) に於ける相互結合網の設計・実装」,
情報処理学会 第 43 回全国大会, 4-273, 1991 年 10 月.

2. 田村孝之, 原田昌信, 平野 聡, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ (SDC) におけるデータネットワークの論理設計」,
情報処理学会 第 44 回全国大会, 4-207, 1992 年 3 月.

3. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ (SDC) のパケット平坦化ネットワークにおける縮退動作時の動作特性」,
情報処理学会 第 45 回全国大会, 6-189, 1992 年 10 月.

4. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ (SDC) の相互結合網におけるスイッチ素子の設計」,
電子情報通信学会 1993 年春季大会, D-145, 1993 年 3 月.

5. 中村 稔, 田村孝之, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ第二版 (SDC2) におけるシステムソフトウェアの構成」,
情報処理学会 第 47 回全国大会, 6-157, 1993 年 10 月.

6. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC2 におけるデータネットワーク系の管理方式」,
情報処理学会 第 47 回全国大会, 6-159, 1993 年 10 月.

7. 北村 学, 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC2 におけるデータネットワークの動作解析」,
情報処理学会 第 47 回全国大会, 6-161, 1993 年 10 月.

8. 北村 学, 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC2 のソフトウェアシミュレータ」,
情報処理学会 第 48 回全国大会, 4-211, 1994 年 3 月.

9. 中村 稔, 田村孝之, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ第二版 (SDC2) におけるマルチジョインの実装方式」,
情報処理学会 第 48 回全国大会, 4-213, 1994 年 3 月.

10. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC-II のデータネットワークにおけるパケッ

ト平坦化機構の実装と評価」,
情報処理学会 第 49 回全国大会, 6-7, 1994 年 9 月.

11. 中村 稔, 田村孝之, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC-II におけるライトディープ多重結合演算の評価」,
情報処理学会 第 49 回全国大会, 4-295, 1994 年 9 月.

12. 中村 稔, 田村孝之, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC-II における不均一データ分布での結合演算処理に対する性能評価」,
情報処理学会 第 50 回全国大会, 6-35, 1995 年 3 月.

13. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC-II のバケット平坦化ネットワークにおける負荷特性の評価」,
情報処理学会 第 50 回全国大会, 6-37, 1995 年 3 月.

14. 田村孝之, 喜連川 優, 高木幹雄:
「並列 SQL サーバ SDC-II におけるバッチ問合せ処理方式」,
情報処理学会 第 51 回全国大会, 4-153, 1995 年 9 月.

15. 田村孝之, 喜連川 優, 高木幹雄:
「並列 SQL サーバ SDC-II におけるトランスポーズ型ファイル編成適用の検討」,
情報処理学会 第 52 回全国大会, 4-283, 1996 年 3 月.

16. 田村孝之, 喜連川 優, 高木幹雄:
「並列 SQL サーバ SDC-II の TPC-D ベンチマークによる性能評価」,
情報処理学会 第 53 回全国大会, 3-17, 1996 年 9 月.

17. 田村孝之, 小口正人, 喜連川 優:
「ATM 結合 PC クラスタ 上での並列関係データベースサーバの構築」,
情報処理学会 第 54 回全国大会, 3-231, 1997 年 3 月.

18. 武藤精吾, 田村孝之, 中野美由紀, 喜連川 優:
「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」,
情報処理学会 第 54 回全国大会, 3-269, 1997 年 3 月.

19. 小口正人, 新谷隆彦, 田村孝之, 喜連川 優:
「ATM 結合型 PC クラスタにおける相関関係抽出」,
1997 年電子情報通信学会総合大会, 1997 年 3 月.

20. 田村孝之, 小口正人, 喜連川 優:
「大規模 PC クラスタにおける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」,
情報処理学会 第 55 回全国大会, 3-457, 1997 年 9 月.

21. 武藤精吾, 田村孝之, 中野美由紀, 喜連川 優:
「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」,
情報処理学会 第 55 回全国大会, 3-21, 1997 年 9 月.

研究会

1. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ (SDC) のバケット平坦化ネットワークにおける縮退動作支援アルゴリズムとその評価」,
情報処理学会 計算機アーキテクチャ研究会 (SWoPP '92), ARC-95-16, 1992 年 8 月.

2. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ (SDC2) におけるデータネットワーク系の実装」,
電子情報通信学会 コンピュータシステム研究会 (SWoPP '93), CPSY93-31, 1993 年 8 月.

3. 中村 稔, 田村孝之, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ第二版 (SDC2) におけるデータ流制御の評価」,
アドバンストデータベースシステムシンポジウム, pp.103–112, 1993 年 12 月.

4. 中村 稔, 田村孝之, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC2 における多重結合演算の実装と評価」,
電子情報通信学会 コンピュータシステム研究会 (SWoPP '94), CPSY94-28, 1994 年 7 月.

5. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC2 におけるデータネットワークの性能評価」,
電子情報通信学会 コンピュータシステム研究会 (SWoPP '94), CPSY94-29, 1994 年 7 月.

6. 中村 稔, 田村孝之, 喜連川 優, 高木幹雄:
「スーパーデータベースコンピュータ SDC-II における並列結合演算処理に関する性能評価」,
情報処理学会 第 100 回データベースシステム研究会, 100-10, 1994 年 10 月.

7. 田村孝之, 中村 稔, 喜連川 優, 高木幹雄:
「並列関係データベース処理を支援する相互結合網: — SDC-II におけるバケット平坦化ネットワークの実装と評価 —」,
並列処理シンポジウム JSPP'95, pp.129–136, 1995 年 5 月.

8. 田村孝之, 喜連川 優, 高木幹雄, 小川義久:
「並列 SQL サーバ SDC-II の TPC-D ベンチマークを用いた性能評価」,
電子情報通信学会 データ工学研究会, DE96-31, 1996 年 7 月.

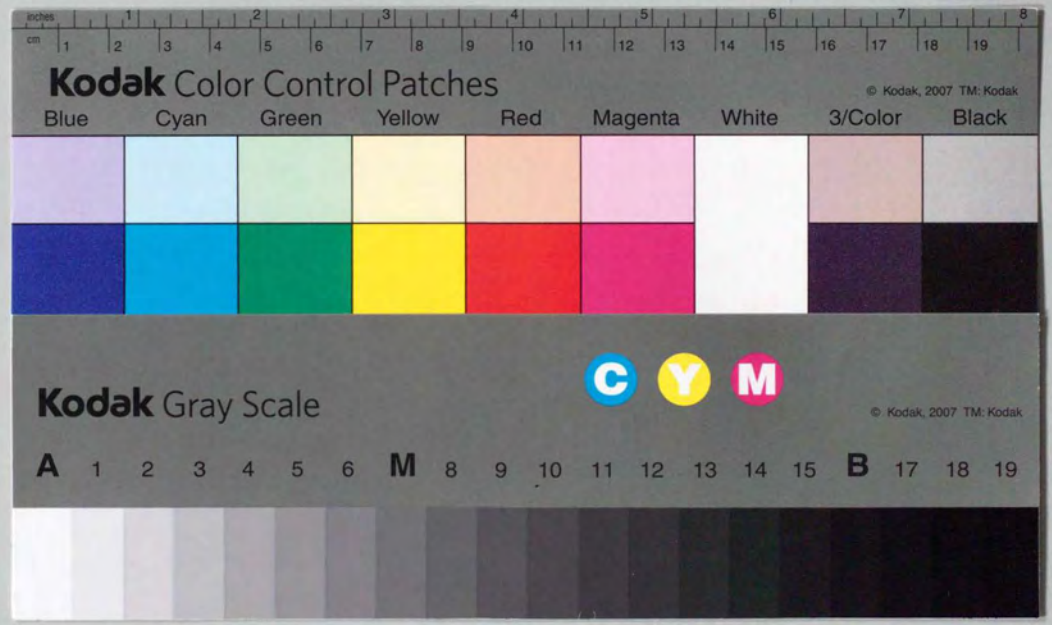
9. 小口正人, 新谷隆彦, 田村孝之, 喜連川 優:
「ATM 結合型 PC クラスタによる並列データマイニング」,
電子情報通信学会 データ工学研究会, DE96-76, 1997 年 1 月.

10. 田村孝之, 小口正人, 喜連川 優:
「ATM 結合 PC クラスタにおける並列関係問合せ処理系の設計と実装」,
電子情報通信学会 データ工学研究会, DE97, 1997 年 5 月.

11. 武藤精吾, 田村孝之, 中野美由紀, 喜連川 優:
「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い合わせ処理」,
電子情報通信学会 データ工学研究会, DE97, 1997 年 7 月.

12. 田村孝之, 小口正人, 喜連川 優:
「大規模 PC クラスタにおける並列関係問合せ実行方式とその評価」,

電子情報通信学会 人工知能と知識処理研究会 / データ工学研究会, AI97-55/DE97-88, 1997 年 12 月.


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19