並列データベース問合せ処理における動的対ノード故障耐性に関する検討

別所祐太朗† 早水 悠登†† 合田 和生†† 喜連川 優††,†††

† 東京大学 大学院情報理工学系研究科 〒 113-8656 東京都文京区本郷 7–3–1
†† 東京大学 生産技術研究所 〒 153–8505 東京都目黒区駒場 4–6–1
††† 国立情報学研究所 〒 101–8430 東京都千代田区一ツ橋 2–1–2
E-mail: †{bessho,haya,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp

あらまし データ分析アプリケーションの取り扱うデータベースの容量は年々増加し，クエリの実行は長時間に渡ることがある．このようなワークロードに対しては並列処理環境の利用が一つの有効な手法であるが，処理ノードの増加に従い障害が発生する確率が無視できなくなる恐れがある．既存の並列データベースシステムでは障害発生後にクエリ全体を再実行するのが一般的であり，ユーザに大きく時間的ペナルティを課することとなる．本論文は，共有ストレージアーキテクチャの並列データベースシステムを対象とし，クエリの進捗を追跡することにより，障害発生時に他ノードが処理を引き継ぐ手法を検討するものである．すなわち，共有ストレージ上のテーブルの並列全スキャンに関する進捗の追跡手法を提案し，実マシン上における試作実装の動作確認について述べる．

キーワード 並列データベース，耐故障性

1 はじめに

データマイニングや BI ツールによる意思決定など，データの統計的分析を行うシステムは多様な場面で利用される．日々蓄積されるデータを対象として構築されるデータベースは容量が増加する傾向にあり，近年ではペタバイトオーダーのものも珍しくない．

大規模なテーブルに対する統計的クエリ処理は長時間を要することがあるため [1]，多数のノードを利用してテーブルに対して並列に処理を行う並列データベースシステムの利用は一般的なアプローチの一つとなっている．

多数の並列ノードを擁する計算機環境においては，クエリの実行中にいずれかのノードが故障する確率が無視できなくなる可能性が高い．特に，並列データベースの処理は通常パイプライン化されているため，それまでの進捗に関わらず，ノード故障の後にクエリ全体を初めから再実行するのが一般的である．大規模なデータベースを構築することでクエリが長時間化する場合，クエリの完全な再実行に伴うユーザへの時間的ペナルティはアプリケーションの利便性を損ねる恐れがある．

ノードの故障に際してクエリを続行する研究は，ストリームデータ処理システム [2,3] やグリッド環境 [4] の他に，並列データベースシステムを対象としたものも存在する [5–8]．しかし，いずれも非共有ストレージ環境を前提としており，共有ストレージ環境を取り扱った研究は筆者の知る限りなされていない．

本論文は，共有ストレージアーキテクチャの環境において，共有ストレージ上のテーブルに対する並列の全スキャンを追跡し，故障ノードの処理を他ノードに引き継がせる手法を提案する．

本論文の構成は次の通りである．まず，第 2 章では現在主流の並列データベースアーキテクチャを紹介し，本研究が対象とする共有ストレージ方式におけるテーブル読み出し方式について説明する．続いて，第 3 章では，本稿で提案する並列全スキャンの進捗の追跡手法を，テーブルの読み出し方式別に述べる．第 4 章で初期実装の実マシン上における性能測定および動作確認の結果と考察を述べ，第 5 章で本稿を総括する．

2 並列データベースシステムと故障

本章では，本研究が対象とする共有ストレージ方式の並列データベースアーキテクチャとその特徴について説明したのち，ノード障害が発生した際の既存のシステムにおける問題点について述べる．

2.1 共有ストレージ方式と非共有ストレージ方式

現在の並列データベースシステムアーキテクチャとしては通常，非共有ストレージ (Shared nothing) 方式，あるいは共有ストレージ (Shared storage) 方式の 2 種類が用いられる．

非共有ストレージ方式においては，各ノードに独立したストレージが接続される (図 1(a))．共有ストレージ方式のように Storage Area Network (SAN) を用意する必要がなく，安価に構築が可能である．また，各ノードのデータアクセスが直接接続されたストレージに集中するため，DBMS はノード間のアクセスの衝突を調停する必要がなく，実装が単純であるという利点も存在する．

共有ストレージ方式においては，複数のノードが単一のストレージに接続される (図 1(b))．全てのノードがストレージの全空間にアクセスできるため，ストレージ領域の利用効率が高

い．また，アプリケーション開発においてストレージ上のテーブルの配置を考える必要がないという利便性や，負荷分散の柔軟性 [9] も利点として挙げられる．

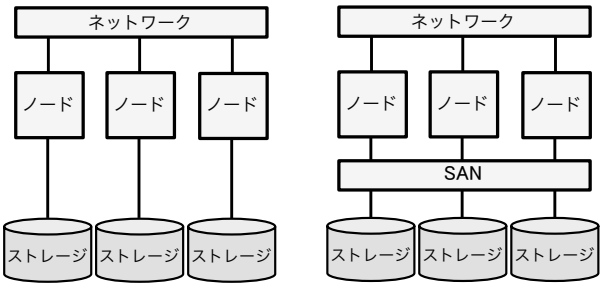
本研究は，共有ストレージ方式の並列データベースを対象とする．

2.2 共有ストレージ方式における並列読み込み

並列データベースシステムのクエリ処理においては，複数ノードが単一のファイル (ここではテーブル) を並行して読み込むことが多い．共有ストレージ方式において用いられる読み込みの方法は，分割読み込みとオンデマンド読み込みの 2 種類に大別される．

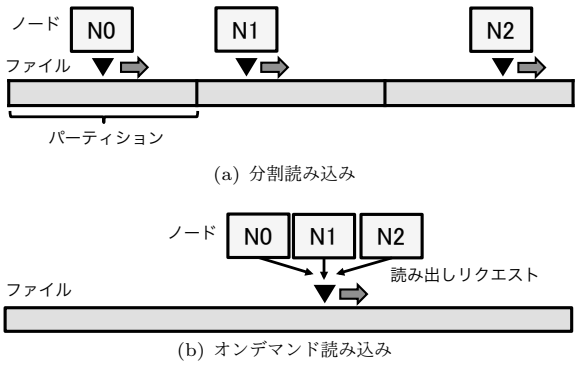
分割読み込み方式 (図 2(a)) では，ファイル全体が連続した領域の断片 (パーティション) に分割され，各処理ノードが各領域を先頭から順番に読み出す．読み出しオフセット (ファイルオフセット，あるいはテーブルのキー) は，各パーティションごとに管理される．この方式では，各処理ノードのワークロードが静的に決定されている． 非共有ストレージ方式における読み出しは，各ストレージをパーティションとみなした時，分割読み込み方式とみなすことができる．

オンデマンド読み込み方式 (図 2(b)) では，全処理ノードが読み出しオフセットを共有し，各処理ノードが発行する読み出しリクエストに応じて，未読み出しの領域がノードに与えられる．この方式では，各処理ノードが読み出す領域は動的に決定する． すなわち，試行ごとに各ノードが読み出す領域は必ずしも同じにはならない．



(a) 非共有ストレージ方式　(b) 共有ストレージ方式

図 1　並列処理データベースアーキテクチャの比較



(a) 分割読み込み

(b) オンデマンド読み込み

図 2　並列データベースシステムにおけるファイル読み出し

2.3 ノード故障による問題

並列データベースシステムは多数の処理ノードを擁するため，クエリ処理の実行中に一部のノードが故障する可能性が無視できない．故障したノードはファイルの読み出しおよび処理済データの出力を停止するので，実行を続けると得られる処理結果は誤ったものになってしまう．

一つの方法として，クエリの処理中にノードの故障が見られた場合はクエリを最初から再実行するというものがある． しかし，大規模データを取り扱う分析的クエリは処理時間が長く，再実行による時間的コストが許容できない場合がある．他の方法として，ノードが故障した際の進捗状況を始点としてクエリの実行を再開するものが考えられる．そのためには，故障したノードが処理できなかったワークロードを他の何らかのノードが正確に特定し，代替ノードに担当させる必要がある．本研究は，共有ストレージ方式においてクエリ実行の進捗状況を追跡する手法を提案するものである．

3 共有ストレージ方式における耐故障クエリ実行方式の検討

本章では，共有ストレージ環境における並列データベースシステムにおいて，クエリ実行の進捗状況を追跡する手法を検討する．これによって，ノードが故障した場合においても，他の idle なノード，あるいは事前に設定された代替ノードが故障ノードが行えなかったワークロードを特定して実行することができる．

本稿では，共有ストレージ上に保存されたテーブルを入力とする並列全スキャン処理の進捗を追跡する手法を取り扱う．読み出しの方法として 分割読み込みを適用した場合について，次に オンデマンド読み込みを適用した場合について述べる．ストリーミングされたレコード群を入力とする処理については今後の検討課題とする．

3.1 分割読み込みにおける並列全スキャンの追跡

並列データベースにおいて，共有ストレージ上のテーブル全体を分割読み込み方式でスキャンしてから，集計を行うワークロードについて考える．スキャンと集計は同じノード群で行うこととする．各ノードでは，スキャンプロセス (scanner) によってスキャンされたレコードが中間バッファを経て，下流のプロセス，ここでは集計プロセス (aggregator) に渡される．

処理対象のテーブルの各レコードの状態は次のように 3 つに分類できる．故障ノードのワークロードを他のノードに代替させるには，各レコードがどの状態であるかを把握することが必要である．

- 未要求状態 (unrequested)． どの処理ノードにも読み出しが要求されていない．
- 未承認状態 (unacknowledged)． 処理ノードに読み出しリクエストされたが，ノードに読み出され処理されていない，あるいは処理結果が下流処理ノードへの入力として確定するに至っていない．

- 承認状態 (acknowledged)． 処理が既に終了し，処理結果が下流処理ノードへの入力として確定した．

分割読み込み方式におけるレコードの要求・承認プロトコルは以下の通りである． 図 5(a) に図解する．

（1） 処理ノードがストレージコントローラにスキャン処理の開始を通知する． すると，全レコードは未要求状態から未承認状態になる． ストレージコントローラは，各パーティションの分割アドレスを設定し， その情報を処理ノードに送信する．以下，各処理ノードが (2) から (7) を繰り返す．

（2） スキャンプロセスが，パーティションの分割情報を利用して，共有ストレージにレコードのデータを要求する． 共有ストレージが，対象のレコードを処理ノードに返す．

（3） スキャンプロセスが，処理後のデータを下流処理ノード (ここでは集計プロセス) との中間バッファに送信する．

（4） ある程度の数のレコードを送信したら，集計プロセスに送信データを入力として承認することを要求する．

（5） 集計プロセスが要求されたデータを入力として取り込む．

（6） 集計プロセスがスキャンプロセスに承認を通知する．

（7） 送信したレコードの承認の通知を受け取ったスキャンプロセスが，ストレージコントローラの管理表における対象レコードの状態を未承認から承認に管理表を更新するよう通知する．

レコードの状態遷移を図 3(a) に示す．全レコードに対して状態を管理する方法例として次のようなものが考えられる．レコードは番号順に下流の処理プロセスに送られるという仮定を置くと，何番目のレコードまでが承認状態であるかを常に記録しておけば，ノードの障害発生時，代替ノードが未承認状態のレコードを順番に読み出すことで処理を再開できる．

例えば，レコード数 1000 のテーブルを 3 つのノードでスキャンする場合を考える．まず，テーブルは 3 つのパーティションに分割される．このとき，ストレージのコントローラに図 4(a) のような管理表を用意すればよい．partition head は，各パーティションの先頭のレコード番号，last ACKed は最後に承認されたレコード，すなわち，承認状態と未承認状態のレコードの境界を示している．例えば図 4(a) の状態で node id が 1 のノードが故障したら，代替ノードは 453 番目のタプルから読み出しを行えばよい．

3.2 オンデマンド読み込みにおける並列全スキャンの追跡

次に，共有ストレージ上のテーブル全体に対しオンデマンド読み込み方式でスキャン及び集計を行うワークロードについて考える．処理ノードの構成は前章と同じであるとする．

分割読み込み方式では，レコードがどのノードに読み出されるかは静的に確定し，他のノードに読み出されることは最後までない．したがって，テーブルのスキャン処理開始後は，全てのレコードには既に読み出し要求がなされたものとみなせ，未要求状態と未承認状態のみを管理すればよい．一方，レコードを読み出すノードが事前に決定しないオンデマンド読み込み方式の場合，ノードが故障したら，故障ノードが処理できなかったレコードを他のノードが読み出して処理できるようにする必要がある． そのために，故障ノードの要求したレコードのうち未承認状態のものは未要求状態に戻すようにプロトコルを設定する．レコードの状態遷移を示した図を図 3(b) に示す．

On-demand 方式におけるレコードの要求・承認プロトコルは以下の通りである． 図 5(b) に図解する．各処理ノードが以下の動作を繰り返す．

（1） 処理ノードのスキャンプロセスがレコードの読み出し要求をストレージコントローラに送信する．ストレージコントローラは管理表を参照し，未承認状態のレコードから処理ノードが読み出すべきレコードのリストを選択し，ノードに返答する．

（2） 処理ノードは返答に従い，共有ストレージに読み出し要求を送信する． 共有ストレージが，対象レコードを読み出す．

（3） 処理ノードは対象レコードの読み出し完了をストレージコントローラに通知し，対象レコードの状態を未要求から未承認に更新させる．

（4） スキャンプロセスによる処理が完了したデータを下流処理ノード (ここでは集計プロセス) との中間バッファに送信する．

（5） ある程度の数のレコードを送信したら，集計プロセスに送信データを入力として承認することを要求する．

（6） 集計プロセスが要求されたデータを入力として取り込む．

（7） 集計プロセスがスキャンプロセスに承認を通知する．

（8） スキャンプロセスがストレージコントローラに，管理表における対象レコードの状態を未承認から承認に更新するよう通知する．

テーブル上の全レコードに対して 3 状態を管理する方法として考えられる例を図 4(b) に示す．node id を処理ノードの番号として，requested ranges に各処理ノードが要求したレコードの集合を可変長セグメントのリストとして保存している． # of acks は，requested ranges のリストの先頭から数えて何個のレコードが承認状態にあるかを示す．requested list はテーブルの未承認レコードの一覧で，読み出し要求可能なレコードを示す．図では，丁度 node id が 1 の処理ノードの故障が検知された状況を例示している．この時点で，requested ranges のリストのうち承認されているのは先頭 55 個のレコード (0-55) であり，未承認のレコード (56-99，300-399) を unrequested list に返却している．また，node id が 2 の処理ノードは番号 250-299 のレコードの承認を得たため，リストから 250-299 を削除し，続いて 450-499 に対して処理および承認を要求する．

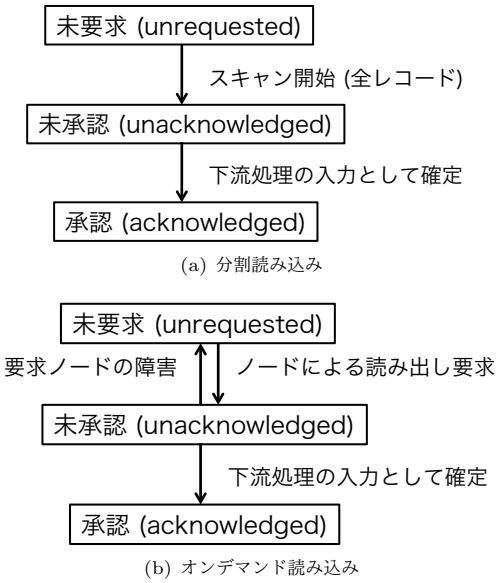
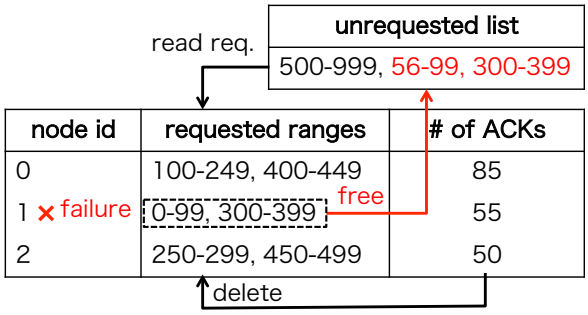


図 3　耐ノード障害性のために管理するレコードの状態遷移

node id	partition head	last ACKed
0	0	100
1	334	452
2	667	80

(a) 分割読み込み



(b) オンデマンド読み込み

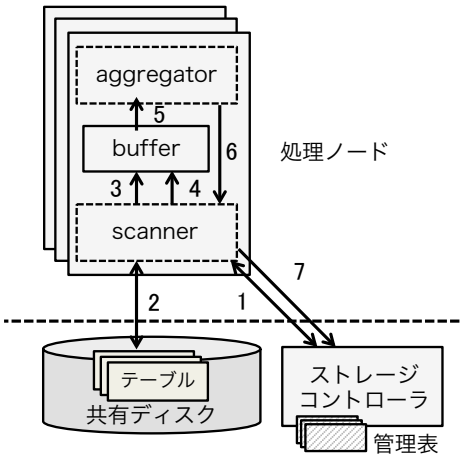
図 4　進行中の並列全スキャンにおける各レコードの状態管理の例

4　並列全スキャン進捗追跡機構の試作実装

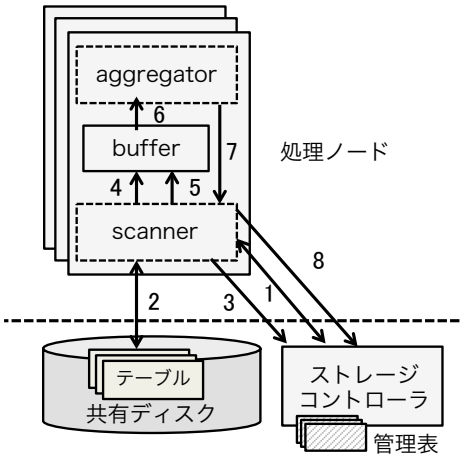
4.1　実装の概要と実験環境

複数ノードによる単一テーブルの並列全スキャンの進捗を追跡する機構の試作を実装し，実サーバ上で動作確認と性能試験を実行した．実装したのは，ストレージコントローラのための進捗管理表を管理し，テーブルの内容を送受信するプロセス (controller)，および処理ノードのスキャンプロセス (scanner) である．処理ノードにおける集計プロセスへのデータ転送等は今回の実験に含めていない．進捗管理表は，図 3(b) のように連続するレコード番号の範囲のリストとして実装した．

実サーバは同一のものを 4 台用意し，Ethernet により相互に接続した．サーバの諸元を表 1 に，接続を図 6 に示す．



(a) 分割読み込み



(b) オンデマンド読み込み

図 5　共有ストレージ上の並列全スキャンにおけるレコード状態の更新プロトコル

3 台のサーバの内，1 台をストレージノード，3 台を処理ノードとした．ストレージノードに接続されたストレージにテーブルのファイルが保存されており，処理ノード上の scanner プロセスが TCP/IP 通信を用いてテーブルを読み出す．ストレージノード controller プロセスは各処理ノードに対し 1 つずつ常駐するワーカスレッドで通信を行う．

この試作実装においては，各処理ノードがテーブルを読み出す際に次のような通信が順番に行われる．番号は図 6 に対応する．

（1） 処理ノードの scanner プロセスがストレージノードの controller プロセスにレコードを要求する． この際，要求するレコードの数を指定する．

（2） controller プロセスの対応するワーカスレッドが管理表を参照し，未要求状態のレコードから指定された数のレコードを選択して未承認状態に変化させる．要求状態にしたレコードの番号のリストを処理ノードに返答する．

（3） 処理ノードの scanner プロセスは，2. で得たレコードの番号を指定して controller プロセスにテーブルの内容の読み出しを要求する．

（4） ストレージノードの controller プロセスの対応する

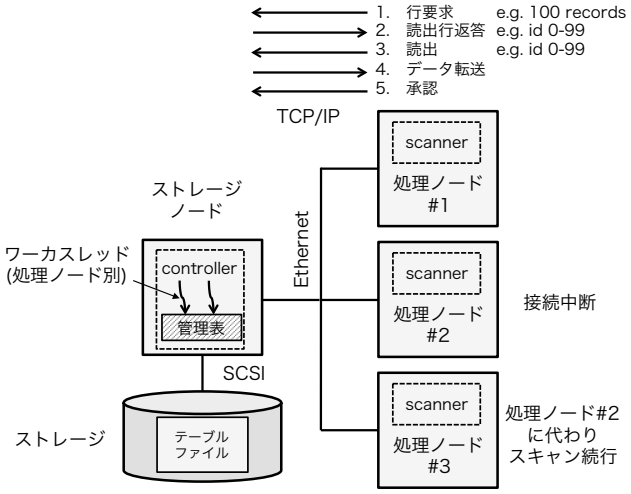


図 6　性能試験におけるサーバの接続及び通信の概要

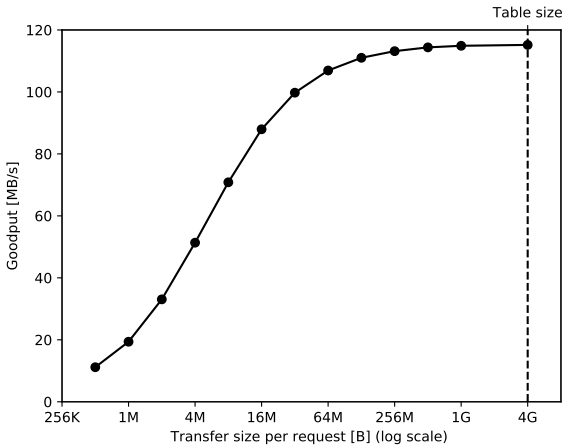


図 7　要求当のレコード転送容量と有効スループットの関係

表 1　実験用サーバ諸元
(全処理ノードとストレージノードは同一マシン)

PowerEdge R740XD	
Processor	Intel(R) Xeon(R) Gold 6132 CPU
Memory	96GB DDR4
OS	Linux 3.10.0-957.1.3.el7.x86_64
HDD conectivity	SCSI
Network Interface	NetXtreme BCM5720 Gigabit Ethernet PCIe

ワーカスレッドが，要求されたレコードをストレージから読み出し，処理ノードに送信する．

（5） 該当するレコードのデータを受信した scanner プロセスは，全て下流処理の入力として確定したものとして， 承認状態にするようストレージノードに要求する． 受け取ったワーカスレッドは該当レコードを承認状態に変化させる（すなわち，表から消去する）．

各ワーカスレッドの管理表の参照および変更は互いに干渉しないように排他制御の下にある．また，各ワーカスレッドは独立したカーソルでテーブルファイルの読み出しを行う．

この設定の下で 2 つの実験，通信オーバヘッドの測定 (4.2 章)，及び ノード障害時の動作確認 (4.3 章) を行った．

4.2　通信オーバヘッドの測定

提案したプロトコルの通信オーバヘッドを測定するために，ストレージノードと処理ノード 1 台 (#1) を用いてテーブルの全スキャン開始から完了までの時間を測定した．開始後は，ストレージノードと処理ノード#1 が前述の (1)〜(5) までの通信を全レコードの読み出しが完了するまで繰り返す．

テーブルは，64B のレコードが連続して並んだ総容量 4GB のファイルである．処理ノードが一度に要求するレコード群の容量を，512KB から，テーブルの総容量 4GB (一度の要求でテーブル全体を読み出す) まで変化させて 3 回ずつ測定を行い，最も経過時間が短かった試行を測定結果として採用した．なお，ファイル全体が OS のページキャッシュに格納された状態で実験を行った．テーブルの総容量を経過時間で割ったものを有効スループットとして算出し，プロットした結果をを図 7 に示す．

要求当たりのレコード転送容量がテーブル全体の大きさに近づくにつれて，有効スループットは 115MB/s 前後に飽和していることがわかる．ここで，性能のボトルネックはネットワークの帯域であることが推測できる．なぜならば，処理ノードは 1 台であるためストレージノードのワーカスレッドは単一であり，管理表への競合は発生していない．また，管理表のデータ構造への操作は計算量の低い単純なものである．さらに，ファイルアクセスのためのディスクアクセスは発生していない．

要求あたりの転送容量が小さくなるにつれ，4.1 章で述べた (1)，(2)，(3)，(5) の通信の回数が増え，有効スループットは減少している．例えば，32MB ずつ，すなわちテーブルを 128 回に等分して転送する場合は，テーブルを 1 回で転送する場合と比較して有効スループットが 13.4%減少している．テーブルの 1/128 の容量は，例えば 16 処理ノードで負荷分散を行う場合，各ノードが行う平均処理量の 1/16 に相当する．したがって，多ノードシステムで十分に負荷分散を図る上で，この転送容量の設定は現実的なものであるといえる．本手法において実用性を得るためには，制御を担う通信 (1)，(2) とデータ転送を担う通信 (3)，(4) を非同期に行うようにするなど，実装を改善する必要があると考えられる．

4.3　ノード障害時の動作確認

次に，テーブルの並列全読み出しの途中で処理ノードに障害が発生し，処理続行できなくなった場合に，予備の処理ノードに読み出しを続行させる動作を確認した．

ストレージノードと 3 つの処理ノード (処理ノード#1，#2，#3) を用意し，まず，2 台の処理ノード#1，#2 にスキャンを並列に実行させた．すなわち，(ストレージノード，処理ノード#1)，(ストレージノード，処理ノード#2) に 4.1 章で述べた (1)〜(5) までの通信を繰り返し行わせた．2 つの処理ノードからの最初のレコード要求はほぼ同時に行われるようにした．

なお，レコードの転送容量は 1 要求当たりつねに 4MB とし

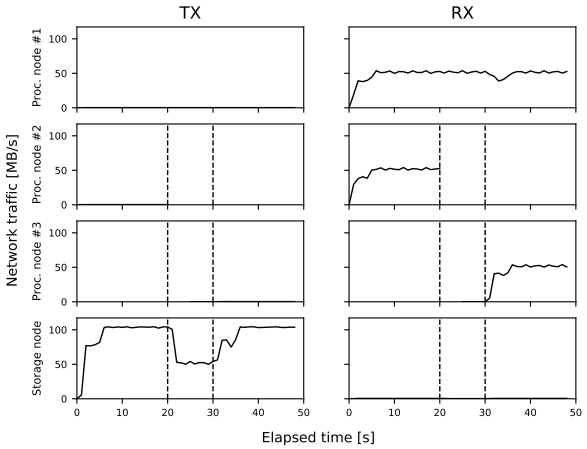


図 8　テーブルのスキャンの途中で処理ノード#2 の通信を中断し，その 10 秒後処理ノード#3 を通信に加えた際の各ノードのネットワークトラフィック量の変化．スキャン開始から 48 秒後の完了まで．上段から，処理ノード#1，2，3，ストレージノード．左列は送信量，右列は受信量．処理ノード#2 の通信中断および#3 の通信開始時刻 (それぞれ 20，30 秒後) を処理ノード#2，3，ストレージノードのグラフに点線で示す．

た．[1]

スキャン開始から 20 秒後に，処理ノード#2 の処理を停止させ，処理停止をストレージノードの controller プロセスに通知した．この時，controller プロセスは処理ノード#2 が要求していたレコードを未要求状態に戻す．続いてその 10 秒後，すなわちスキャン開始から 30 秒後に，処理ノード#3 がストレージノードと通信を確立し，処理ノード#2 に代わってレコードの読み出し通信を開始させた．結果として，テーブルの全ての行が処理ノードに読み出されたことを確認できた．

ストレージノード，および 3 台の処理ノードについて，ネットワーク入出力のスループットを経時計測したものを図 8 に示す．処理ノード#2 の通信を中断させた時より，ストレージノードの送信量が半減していることが読み取れる．次に，処理ノード#3 の処理が開始すると，ストレージノードの送信量が再び中断前の値まで上昇していることがわかる．また，ストレージノードの受信量，処理ノードの送信量はほぼ 0 であり，データ転送以外の制御用の通信はネットワーク帯域占有の観点においてはほぼ無視できるといえる．

4.4　今後の課題

本実験においては，レコードの要求のための制御通信をデータ転送と並行して行わないと性能が有意に低下することが観測されたため，実装を改善する必要がある．

また，hash join などの中間データを生成する処理に対応する場合，および，上流処理ノードからストリーミングされたテーブルを追跡する場合の手法は未検討であるため，これも本稿執筆時点での今後の課題としたい．

1：この転送容量を選択した理由は，ネットワークの利用帯域を抑え，処理ノード 2 台で通信を行ってもネットワーク帯域がボトルネックにならないようにするためである．これにより，ストレージノードにおける送信トラフィック量が半減することが観測できる．

5　おわりに

本研究は，共有ストレージアーキテクチャの並列データベースシステムにおいて，クエリの進捗をストレージコントローラで追跡させることで，障害発生時に他ノードに処理を引き継がせる手法を検討するものである．本稿では，共有ストレージ上のテーブルの並列全スキャンに関する進捗を追跡する手法を提案した．また，実マシン上における試作実装の動作確認を行い，制御通信とデータ転送の非同期化の必要などを明らかにした．

文　献

[1] David Taniar. High performance database processing. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pp. 5–6. IEEE, 2012.

[2] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 827–838. ACM, 2004.

[3] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pp. 779–790. IEEE, 2005.

[4] M Nedim Alpdemir, Arijit Mukherjee, Anastasios Gounaris, Norman W Paton, Paul Watson, Alvaro AA Fernandes, and Desmond J Fitzgerald. Ogsa-dqp: A service for distributed querying on the grid. In *International Conference on Extending Database Technology*, pp. 858–861. Springer, 2004.

[5] Stanislaw Bartkowski, Ciaran De Buitlear, Adrian Kalicki, Michael Loster, Marcin Marczewski, Anas Mosaad, Jan Nelken, Mohamed Soliman, Klaus Subtil, Marko Vrhovnik, et al. *High availability and disaster recovery options for DB2 for Linux, UNIX, and Windows*. IBM Redbooks, 2012.

[6] Jim Smith and Paul Watson. Fault-tolerance in distributed query processing. In *Database Engineering and Application Symposium, 2005. IDEAS 2005. 9th International*, pp. 329–338. IEEE, 2005.

[7] Binh Han, Edward Omiecinski, Leo Mark, and Ling Liu. Otpm: Failure handling in data-intensive analytical processing. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference on*, pp. 35–44. IEEE, 2011.

[8] Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 241–252. ACM, 2011.

[9] 合田和生, 田村孝之, 小口正人, 喜連川優. San 結合 pc クラスタにおけるストレージ仮想化機構を用いた動的負荷分散並びに動的資源調整の提案とその評価. 電子情報通信学会論文誌 D, Vol. 87, No. 6, pp. 661–674, 2004.