社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

動的障害回復が可能な分析系並列データベースシステムの性能評価モデルの検討

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
†† 東京大学 生産技術研究所
††† 国立情報学研究所

あらまし 近年さらに需要が拡大する意思決定アプリケーションが取り扱うデータベースの容量は年々増加し，問合せの実行は長時間を要することがある．この類のワークロードに頻繁に用いられる並列データベースシステムは, 実行環境が多数のノードやストレージで構成されるため, 問合せ処理の実行中に障害が発生する確率が無視できない．障害の際に問合せ全体を再実行する単純な対処法には, 結果取得までの経過時間が大幅に増加する問題がある．本論文は，共有ストレージ構成の並列データベースシステムと分析系問合せを対象に，処理状態の追跡およびノード間冗長化により，ノード障害の際に他のノードがその実行状態を引き継いで処理を継続することを可能とする手法を提案する．また，手法の性能評価モデルを構築し有効性の議論を行う．

キーワード 並列データベースシステム，耐障害性，問合せ処理

A Study on Performance Model of an Analytical Parallel Dabtabase System with Dynamic Fault Tolerance

Yutaro BESSHO†, Yuto HAYAMIZU††, Kazuo GODA††, and Masaru KITSUREGAWA††,†††

† Graduate School of Information Science and Technology, The University of Tokyo
†† Institute of Industrial Science, The University of Tokyo
††† National Institute of Informatics

Abstract Driven by the ever-increasing demand of decision-making applications, database volume continues to grow, leading to prolonged query execution. Although parallel database systems are widely deployed for such analytical purposes, numerous components of nodes and storage cause failures during execution with non-negligible probabilities. The trivial approach of restarting entire queries can impose unacceptable temporal costs. In this paper, we propose a method that enables shared-storage parallel database systems to resume analytical queries by having back-up nodes take over pre-failure execution states that have been tracked and replicated among nodes. We also present several performance models of the mechanism and discuss its effect.

Key words parallel database system, fault tolerance, query processing

1. はじめに

ビッグデータの応用先は意思決定支援など多岐に渡る．その需要の増大に従い，分析用データベースに蓄積されるデータの量には増大の傾向が見られる．数十〜数百ペタバイト級容量の分析環境の構築は珍しくなく，様々な企業がその運用に関する知見を公開している [1], [2]．近年，IoT 機器類のセンサ情報が蓄積が本格化している状況を踏まえると，この傾向は今後も加速すると予想される．

大規模データベースの管理には，多数の計算機（ノード）を並列に駆動する並列データベースシステムと呼ばれる構成が広く利用されるが，問合せの実行中に予期せずいずれかのノードが故障する可能性が無視できない．ノード故障の際は，通常は正常に動作中のノードの実行状態を破棄して，問合せ処理を最初から再実行する．分析の対象となるデータベースの巨大化が進む現在では問合せ処理が数日を要することもあるため，再実行に伴う運用コストは無視できなくなっている．

本論文は，共有ストレージ構成の並列データベースシステムを対象に，ノードに障害が発生した場合に，正常なノードを以って当該問合せ処理を継続することを可能とする手法を提案

する．これは，問合せ処理の実行中に，処理の進捗の追跡と途中結果の維持を行うことで達成される．また，当該手法の性能モデルを構築し，選択・射影および結合（索引結合, ハッシュ結合）における性能を考察することで有効性を議論する．なお，著者はストレージの入出力追跡によるノード障害発生時の処理引継ぎ手法に関する提案及び実験の結果を発表している [3]．本論文は，これを基に新たにノード間でのデータ交換がある処理方式への適用や，問合せの結果にも耐障害性の保証が可能になるよう拡張する手法を検討するものである．

本論文の構成は以下の通りである．第 2 章では本論文が対象とする共有ストレージ構成の並列データベースシステムを説明し，当該構成におけるノード障害が引き起こす非効率性について述べる．次に，第 3 章で動的障害回復を可能にする問合せ進捗追跡機構を提案する．第 4 章で提案手法の性能モデルを構築し，典型的な問合せにおける適用の利得を論じる．第 5 章で関連研究を概観し，第 6 章で本論文を総括する．

2. 並列データベースシステムの故障

共有ストレージにおける負荷分散　並列データベースシステムは，複数の計算機（ノード）を並列に動作させ，大規模なデータベースシステムに対しても高速な問合せ処理を図るものである．当該システムの構成のうち，商用データベースに広く用いられているものは，主に共有ストレージ構成と無共有構成に分類できる．本論文では前者を対象として，議論を進める．

共有ストレージ構成は，複数のノード間でストレージを共有するものである．この構成の利点には，ストレージ管理とノード管理を分離できる利便性のほか，各ノードがストレージの全空間に読書き可能であることから動的な負荷分散が可能であるといった点が挙げられる．ストレージが共有されていることから，単一のデータの読込みも複数のノードが協調して並列に行うことができる．例えば，巨大な単一関係表を複数ノードで走査する場合を考える．ストレージは単一の読出しオフセットを管理し，問合せ処理の実行中に，データを要求したノードが関係表のどの部位を読み出すべきかを決定してデータを転送する [4]．処理ノードの処理能力に応じた入出力量の動的な調整によって高度な負荷分散が実現可能である．

ノード故障時の問合せ再実行の非効率性　並列データベースシステムは並列性向上を図り多数のノードから成るため，問合せ処理中に何れかのノードが故障する確率が無視できない．

一般に，並列データベースシステムは，処理の効率性の観点から，中間的な結果を逐一永続化することをせず，演算をパイプライン化して実行する．したがって，問合せ実行中にノード故障が起きた場合には，問合せ処理を最初から再実行する必要がある．この実行方式においては結果が得られるまでの時間が大幅に増加しユーザの許容範囲を超える可能性が高まる他，運用コストの増大にも繋がる．

3. 動的障害回復を可能にする問合せ進捗追跡機構

本章では，共有ストレージ構成の並列データベースシステム

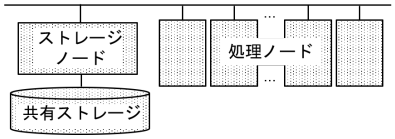


図 1: 共有ストレージ構成の並列データベースシステム

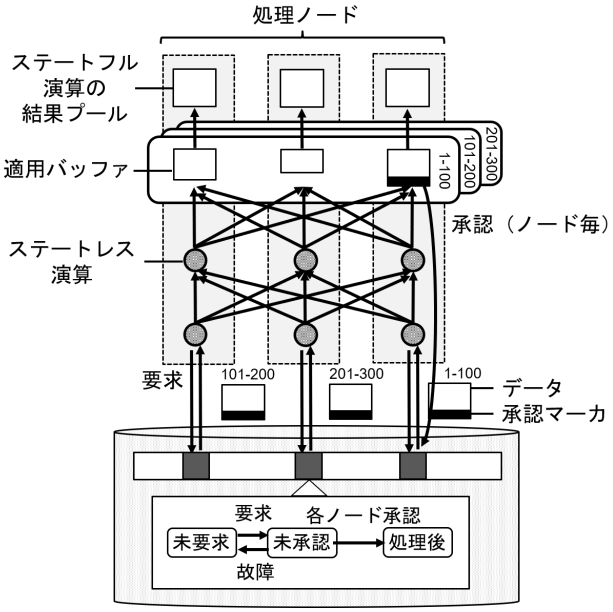


図 2: 動的障害回復を可能にする問合せ進捗追跡機構

において，問合せ実行中のノード障害の際に代替の正常なノードに処理を継続させることを可能とする手法を提案する．

システム構成および障害モデル　本論文では，並列データベースシステムの構成として図 1 に示すものを前提として議論する．ストレージは入出力を仲介するノード（ストレージノード）を挟み問合せ処理を並列に実行するノード（処理ノード）群に接続される．処理ノードは必要に応じて関係表をストレージノードを介して読出して処理を施す．以降は，処理ノードが予期せず停止する障害を想定し（ネットワーク障害等は対象外），障害検知の何らかの手法が利用可能である前提で議論を進める．

パイプライン並列性とクエリブロック　並列データベースシステムは多数の演算をパイプライン並列化することを可能な限り試みる．選択・射影などといった，出力が入力の小さい範囲にのみ依存する（ステートレスな）演算は容易にパイプライン化が可能である．一方，集約やハッシュ表形成のような，出力が入力の広範囲に依存する（ステートフルな）演算がある箇所ではパイプラインの分断が避けられない．ステートフルな演算を境界とする，パイプライン化可能な問合せ木の最大単位を今後クエリブロックと呼ぶことにする．

提案する問合せ実行方式　問合せ処理中に何れかの処理ノードが故障した際，正常動作中および代替のノードに処理を引き継ぎ正しい結果を得るには，入力表のうち二つの条件の何れにも該当しない範囲を処理ノードに割当てる必要がある：(1) 正常動作中の処理ノードが既に処理を引受けている，或いは (2) 処理結果がクエリブロックの出力に適用されることが確定している．

この範囲を特定するために，クエリブロックの演算の実行状態を，ストレージが入力表の各行を「未要求」,「未承認」,「処理後」の 3 状態に分類して管理する．

図 2 に本論文で提案する単一クエリブロックにおける問合せ処理機構を示す．クエリブロックのパイプラインには幾つかのステートレスな演算に続き，最後にステートフルな演算が配置される．処理ノードは処理中にノード間でデータを再分配する場合がある（例：並列ハッシュ結合）．

クエリブロックにおける各行の初期状態は「未要求」である．処理ノードが行が要求すると，ストレージノードは行を転送し，該当行の状態を「未要求」から「未承認」に遷移させる．

行範囲を読み出した処理ノードは，データの末尾に承認マーカと呼ばれるタグ情報を付与する．このタグ情報には，付与されている行範囲を識別できる情報が記録されており，処理ノードは，承認マーカを送信することで，該当範囲の行を今後転送しないことを受信者に通知できる．

続いて，処理ノードは読み出した行を処理する．この際，承認マーカは他の全処理ノードにブロードキャストする．受信ノードは，ある行範囲について，他の全ノードからの承認マーカを受け取って初めて，今後該当データが送付されないと判断できる．なぜならば，処理ノード間でデータの再分配が発生するとき，データは最初にストレージから読出したノードに限らず，全ノードが持ちうるためである．再分配されたデータが次の処理段階で更に再分配される場合，処理ノードは受け取った承認マーカ群を一つに併合してブロードキャストしてよい．

最後の再分配を経た行は，クエリブロックの終端のステートフルな演算に直ちに適用されず，適用バッファと呼ばれる記憶領域に一時的に配置される．ここで，処理ノードは該当行の承認マーカを他の全ノードから受信し，該当行範囲を今後受信することはないと判断すると，該当行範囲への承認をストレージへ通知する．承認した行範囲は，クエリブロック最終のステートフルな演算に適用できる．

ある行範囲の全ての処理ノードによる承認を受信したストレージノードは，該当行の状態を「処理後」に遷移させる．

いずれかの処理ノードの障害が検知されると，ストレージは故障ノードに割り当てられていた「未承認」行を「未要求」状態に戻す．これによって，故障ノードが引受けており，かつクエリブロックの処理結果として取り込まれることが未確定の処理が，その他の正常に動作している処理ノードに引き継がれる．

全ての入力行が「処理後」状態に遷移した時がクエリブロックの処理の完了である．「未承認」の行は前述の条件 (1) を満たす範囲に，「処理後」の行は条件 (2) を満たす範囲に相当するので，提案の手順に従うことで，ノード障害の発生下でも漏れと重複の無い処理の分配を達成できる．

処理状態の冗長化と障害回復　クエリブロックの末端の処理結果は各ノードに分散して蓄積される．しかし，並列データベースシステムは可能な限りデータを主記憶上で扱うため，冗長化されていなければ，障害を生じたノードに保存されていた処理結果は失われ，復元に全体再実行を要する．そこで，クエリブロックの最終の再分配の際に，分配先以外のノードにも該当データを送信し，同じ演算を行わせる．ストレージへの行の承認は，他ノードへの送信完了まで待って行うことで，承認済の結果が失われないことを保証する．

ノード障害が発生した際は，代替ノードが処理ノード群に加わり，障害ノードの処理結果の複製を生存ノードから受信し，問合せ処理を再開する．これにより，全体再実行を避けつつ，障害が発生した状態から問合せを再開できる．ただし，通常実行時のネットワークおよび CPU コストは増大する．これによる性能の影響を次の章で検討する．

表 1: 性能モデルの各パラメタ及びデフォルト値

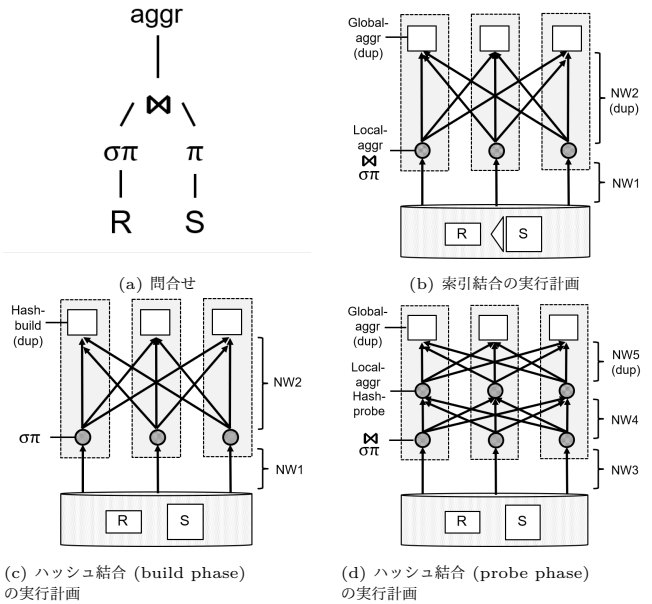
表記	説明	デフォルト値
$\|R\|$	R の行数	1×10^7
$\|S\|$	S の行数	1×10^8
r_R	R の行の長さ	100 B
r_S	S の行の長さ	100 B
r_P	射影後の R, S 各行の長さ	50 B
r_m	の長さ	100 B
s_R	R の選択率	1
j	$R1$ 行に結合する S の行数	10
a_{LA}	局所集約前行数に対する 局所集約後行数の割合	1
a_{GA}	局所集約前行数に対する 大域集約後行数の割合	1
N	処理ノード数	16
I	処理ノードが を 挿入する間隔（行）	1000
B_{st}	ストレージ帯域	1 GB/s
B_{nw}	ネットワーク帯域（処理ノード当）	1.25 GB/s
n_{CPU}	処理ノード当の並列度 （物理コア或は vCPU 数）	2
c_σ	1 行の選択処理に要する CPU 時間	1×10^{-7}s
c_{LA}	1 行の局所集約で経過する CPU 時間	1×10^{-7}s
c_{GA}	1 行の大域集約で経過する CPU 時間	5×10^{-7}s
c_{idx}	索引結合で 1 行の出力 に要する CPU 時間	1×10^{-7}s
c_{hash}	ハッシュ結合でハッシュ表 1 行追加 に要する CPU 時間	1×10^{-6}s
c_{probe}	ハッシュ結合で 1 行のプローブ に要する CPU 時間	2×10^{-6}s

4. 動的障害回復のための問合せ進捗追跡機構の性能評価モデル

本章では，前章で提案した手法の性能を考察する．典型的な問合せ処理に対して，問合せ及びハードウェアのパラメタの様々な組合わせでストレージ，ネットワーク，CPU の律速関係を定量化することで，適用による利得の多寡を論じる．

図 3a に示す 2 表 R, S の結合を考える．処理方式として，索引結合とハッシュ結合の 2 通りを考える．ただし，ハッシュ結合においてハッシュ表は R から作られ、かつ処理ノードの主記憶に収まるとする．

問合せ及び実行環境に関するパラメタの表記とデフォルト値を表 1 に示す．ハードウェア構成は図 1 を前提とする．CPU



(a) 問合せ　(b) 索引結合の実行計画

(c) ハッシュ結合 (build phase) の実行計画　(d) ハッシュ結合 (probe phase) の実行計画

図 3: 対象の問合せの問合せ木及び実行計画．ネットワーク転送（矢印）及び演算（丸印）に (dup) と注釈がある場合，提案手法では冗長化される．

時間に関する値は，同様の処理を模したマイクロベンチマークを作成し，Amazon Web Services の提供する EC2 t2.medium 仮想マシン上で実行した結果に基づいている(注1)．

各クエリブロックにおいては，処理スループットの経時的変化はないものとする．従って，ストレージ，ネットワーク，CPU それぞれの資源のコストのみを考慮した（今後これをストレージ・ネットワーク・CPU 律速の処理時間と呼ぶ）実行時間 T_{st}, T_{nw}, T_{cpu} を算出した時，最大値を取る要素がシステムを律速するので，クエリブロックの実行時間は

$$T = max(T_{st}, T_{nw}, T_{cpu}) \tag{1}$$

として得られる．

ネットワーク及びストレージの転送機構が律速する時には，帯域が律速する場合と時間当りの入出力数（IOPS）が律速する場合がある．本稿では，入出力を可能な限り粗い粒度で行う実装を前提とし，IOPS で律速することはないと仮定する．このとき，T_{st}, T_{nw} は，クエリブロック開始から完了までの転送量 L_{st}, L_{nw} を算出すれば

$$T_{st} = \frac{L_{st}}{B_{st}}, T_{nw} = \frac{L_{nw}}{NB_{nw}} \tag{2}$$

と求められる．

次に，提案手法における障害回復，すなわち実行状態の回復に必要な時間 $T_{restore}$ は，当該クエリブロックで冗長化されるデータの総量 L_{repl}[B] をネットワークで代替ノードに送信する際の経過時間であるから，バックアップ保持を 1 台の処理ノードが受け持つとすると，故障時のクエリブロックの進捗度（経過時間により算出）を $0 < \alpha < 1$ として以下のようにモデル化できる．

$$T_{restore}(\alpha) = \alpha \frac{L_{repl}}{B_{nw}} \tag{3}$$

とモデル化できる．

4.1 索引結合

ストレージ律速の実行時間 T_{st} は，読み出すデータの総量

$$L_{st} = |R|(r_R + s_R j r_S) \tag{4}$$

から式 2 で求められる．ただし，索引の読出しによる帯域圧迫は無視している．ネットワーク律速の実行時間 T_{nw} を求めるには総転送量 L_{nw} が必要であり，図 3b の 2 つの転送段階 NW1, NW2 における転送量をそれぞれ L_{nw1}, L_{nw2} として

$$L_{nw} = L_{nw1} + L_{nw2} \tag{5}$$

$$L_{nw1} = |R|(r_R + s_R j r_S) \tag{6}$$

$$L_{nw2} = \begin{cases} |R| s_R j a_{LA} r_P & (w/o) \\ 2|R|(s_R j a_{LA} r_P + I^{-1} N^2 r_m) & (w/) \end{cases} \tag{7}$$

と求まる．T_{nw2} は手法適用による転送データ冗長化とマーカ転送を考慮して場合分けしている（w/o，$w/$ はそれぞれ適用なし・ありの場合を表す）．これと式 2 から T_{nw} が求まる．CPU 律速の処理時間は，各処理の CPU 時間と，提案手法による大域集約処理の冗長化を考慮して

$$T_{cpu} = \begin{cases} \frac{|R|(c_\sigma + s_R j c_{idx} + c_{LA} + c_{GA})}{n_{CPU} N} & (w/o) \\ \frac{|R|(c_\sigma + s_R j c_{idx} + c_{LA} + 2c_{GA})}{n_{CPU} N} & (w/) \end{cases} \tag{8}$$

と算出される．なお，提案手法適用の際に冗長化されるデータ量は集約の結果プールの容量であり，以下の通りである．

$$L_{repl} = |R| s_R j a_{GA} r_P \tag{9}$$

4.2 ハッシュ結合

ハッシュ結合は，R から主記憶上にハッシュ表を構築するクエリブロック (build phase) と，S の行でハッシュ表を引き結合行を探索するクエリブロック (probe phase) に二分される，

4.2.1 Build phase

ストレージ律速の実行時間 T_{st} は，

$$L_{st} = |R| r_R \tag{10}$$

と式 2 で算出できる．図 3c のネットワーク転送段階 NW1, NW2 の転送量を各々 L_{nw1}, L_{nw2} とすると，T_{nw} は

$$L_{nw} = L_{nw1} + L_{nw2} \tag{11}$$

$$L_{nw1} = |R| r_R \tag{12}$$

$$L_{nw2} = \begin{cases} |R| s_R r_P & (w/o) \\ 2|R|(s_R r_P + I^{-1} N^2 r_m) & (w/) \end{cases} \tag{13}$$

と式 2 から求められる．CPU 律速の処理時間は，提案手法でハッシュ表の構築を冗長化することから以下のように求まる．

手法適用で冗長化されるデータ量は，ハッシュ表の容量なので以下の通りである．

(注1)：ハッシュ表は，1 つの bucket に平均 10 個の要素が単方向リストで繋がるような設定で構築し，性能を測定した．集約についても，同様のハッシュ表を用いた集約を行うことで測定を行った

$$L_{repl} = |R|s_R \tag{14}$$

4.2.2 Probe phase

ストレージ律速の場合の処理時間 T_{st} は

$$L_{st} = |S|r_S \tag{15}$$

と 2 から，T_{nw} は，図 3d に示した 3 つの転送段階 NW3, NW4, NW5 における転送量を $L_{nw3}, L_{nw4}, L_{nw5}$ として，L_{nw} を以下のように求め式 2 を利用して得られる．

$$L_{nw} = L_{nw3} + L_{nw4} + L_{nw5} \tag{16}$$

$$L_{nw3} = |S|r_S \tag{17}$$

$$L_{nw4} = \begin{cases} |S|r_P & (w/o) \\ |S|(r_P + I^{-1}N^2 r_m) & (w/) \end{cases} \tag{18}$$

$$L_{nw5} = \begin{cases} |R|s_R j a_{LA} r_P & (w/o) \\ 2(|R|s_R j a_{LA} r_P + I^{-1}N^2|S|r_m) & (w/) \end{cases} \tag{19}$$

$$\tag{20}$$

また，CPU 律速の実行時間は，提案手法では終端の全体集約が冗長化されるので次のように算出される．

手法適用で冗長化されるデータ量 L_{repl} は，集約の結果プールの量で，索引結合の場合と同一である．

4.3 パラメタと性能の関係

選択率 s_R, 結合率 j，ノード数 N, ストレージ帯域 B_{st} を変化させた時の，前章で求めた T_{st}, T_{nw}, T_{cpu} と，律速箇所を考慮（式 1）した最終的な実行時間（T）の関係を図 4（索引結合），図 5（ハッシュ結合）に示す．なお，結合率 j と表 S の行数 $|S|$ は連動して比例して変化させた．

まず索引結合については，デフォルト値ではシステム全体がストレージに律速している．選択率 s_R，結合率 j には 3 部位どれも影響を受けるが，特に結合率 j 上昇の CPU 負荷への影響が顕著である．いずれの場合もストレージに律速するので，ストレージの転送量に影響しない本手法は問合せ処理中にオーバヘッドを生じない． ネットワーク・CPU 資源はノード数 N に比例するので，実行時間とは反比例する．ノード数が少ない時はネットワーク或は CPU 律速になりうることが読み取れる．ストレージ帯域 B_{st} の変化に着目すると，B_{st} が 2GB/s すなわち 16Gbps 以内であればストレージ律速であり，手法適用による通常時の実行ペナルティはない．なお，今回無視した索引の入出力を考慮すれば，ストレージ律速の度合いは更に高まる．

次に，ハッシュ結合の場合を見る．デフォルトの値では，T_{st} と T_{cpu} がほぼ同じである．ストレージが他の箇所と大差で律速している索引結合に比べ CPU 負荷が高いことがわかる．結合率 j が小さいところでは手法適用ありの T_{cpu} が律速しているが，これは probe phase が短く，build phase の性能特性が現れているためである．提案手法では Build phase でハッシュ表形成処理を複製する影響で CPU 負荷が高まり，システム全体を律速している．また，索引結合に比べ，システム全体が CPU 或はネットワーク律速に切り替わるノード数 N の閾値が高い．ストレージ帯域 B_{st} の変化に着目すると，1GB/s すなわち 8Gbps を境目にストレージ律速か否か，即ち手法適用による通常時の実行時間ペナルティの有無が切り替わる．

4.4 障害回復時の削減効果

提案手法による動的障害回復の利得を定量化するため，次の 2 つの操作における総経過時間を比較した：問合せ処理中に一台のノードが故障したとき，従来手法により全体を再実行して問合せ処理を完遂する場合（w/o）と，提案手法により $T_{restore}$ 秒を費やし障害発生時の実行状態を回復して問合せ処理を再開し完了まで待つ場合（w/）．

図 6 に，索引結合およびハッシュ結合において, 異なる進捗度（無故障の時の問合せの所要時間に対する故障発生時刻の割合）でノード故障が発生した場合の総経過時間を示す．ただし，手法非適用・適用の場合では問合せ処理速度が異なるため，故障発生までの絶対時間も同時に示した．

通常実行時のペナルティは既に見たように索引結合では無く，ハッシュ結合の build phase ではある程度生じている．これはグラフには切片として現れている．故障時の実行状態回復ペナルティは，故障時刻が遅いほど回復すべき実行状態の容量が増えるので比例して増加する．これはグラフの傾斜として現れるが，図 6 より，このパラメタでは無視できる程度といえる．結果として，索引結合では常に，また，ハッシュ結合では故障時進捗度が 10% 以降の場合には手法を適用する方が総実行時間が短いことが読み取れる．

5. 関 連 研 究

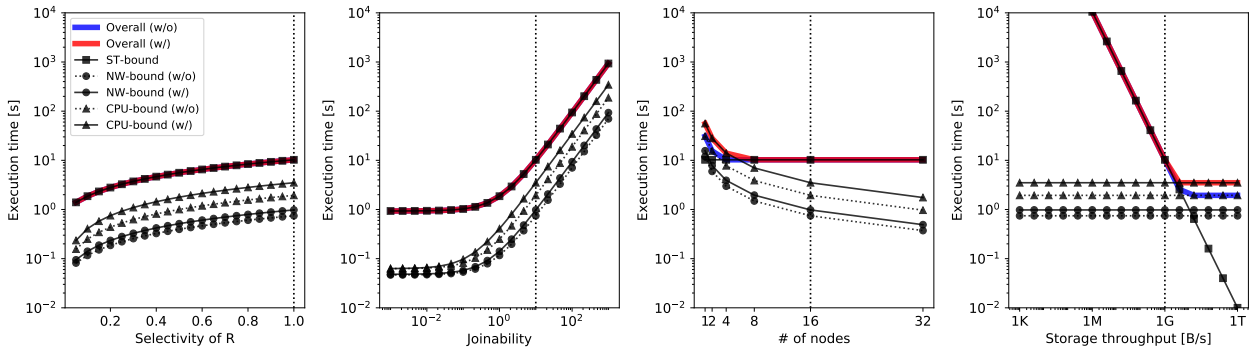
MapReduce [5] 系の並列処理フレームワーク等は処理の中間出力を毎度永続化するので，直近の永続化時点から実行状態を復元できるが，それに伴うオーバヘッドは一般に無視できない．データベースシステム一般では，複製ノードへの定期的同期を利用した障害回復が行われる [6]．ストリーミング処理が対象の研究では，問合せ結果の正確さを犠牲にし，動的かつ効率的な障害回復を実現する方式が提案されている [7], [8]．J. Smith は，障害回復に伴う重複データ排除を目的として，上流ノードによるバックアップと識別子付ログ配送手法を提案している [9]．また，下流ノードが行番号単位で上流ノードの処理進捗を追跡して重複を排除する手法も提案されている [10], [11]．特に後者は出力重複の排除だけでなく再計算コスト削減を目指しており，手法と目的が本研究に近い．しかし，共有ストレージにおける動的障害回復をめざす研究は筆者らの知る限り存在しない．

6. お わ り に

本論文では，共有ストレージ構成の並列データベースシステムを対象に，進捗追跡と処理状態冗長化により，分析系問合せ処理中のノード障害の際に代替ノードに実行状態を引継がせ処理を継続させる事を可能とする手法を提案した．また，手法の性能評価モデルを構築し有効性を議論した．今後は，本稿の議論を基に試作機を構築し，より包括的な検証を行う．

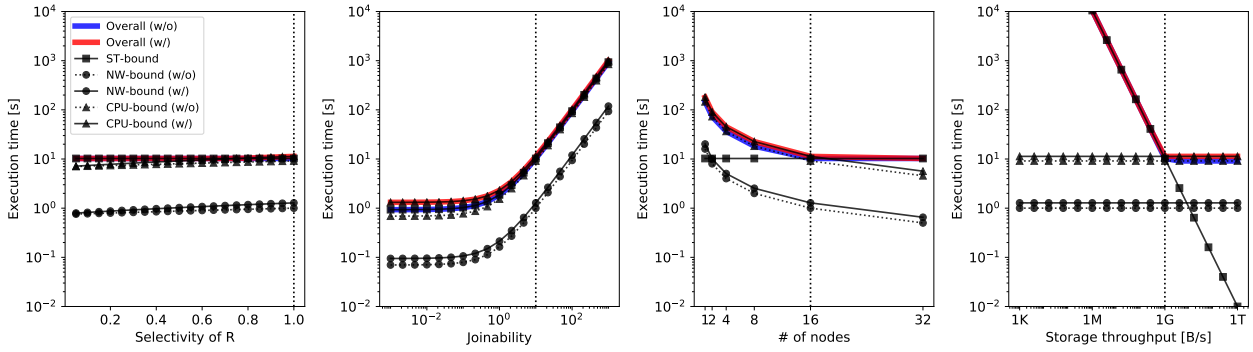
文　　献

[1] Reza Shiftehfar: “Uber’s Big Data Platform: 100+



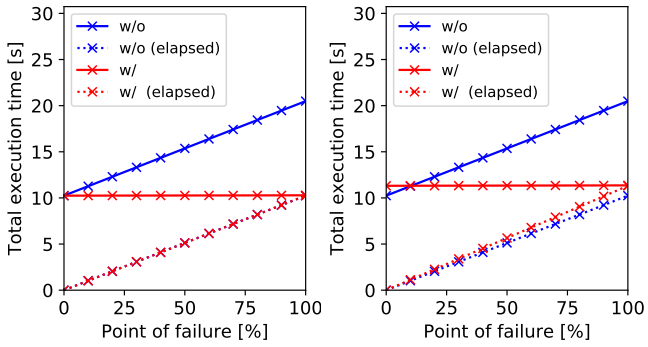
(a) 選択率 s_R を変化　(b) 結合率 j を変化　(c) ノード数 N を変化　(d) ストレージ帯域 B_{st} を変化

図 4: 各パラメタと索引結合による問合せ総実行時間の関係．"ST-bound", "NW-bound", "CPU-bound"は各々ストレージ転送，ネットワーク転送，CPU 資源のみを考慮した場合（T_{st}, T_{nw}, T_{cpu}）．"Overall"は律速箇所に合わせたシステム全体の実行時間（T）．末尾の $(w/o), (w/)$ の表記はそれぞれ提案手法適用あり・なしによる系列であることを示す．縦方向の点線は変数のデフォルト値．



(a) 選択率 s_R を変化　(b) 結合率 j を変化　(c) ノード数 N を変化　(d) ストレージ帯域 B_{st} を変化

図 5: 各パラメタとハッシュ結合による問合せ総実行時間の関係（Build phase と Probe phase の実行時間の和）．凡例については図 4 に同じ．



(a) 索引結合　(b) ハッシュ結合

図 6: 故障時の問合せ進捗率と総実行時間の関係（各パラメタはデフォルト値）．末尾に"(elapsed)"のつく系列は処理開始から障害発生までの時間を表す．

Petabytes with Minute Latency”, https://eng.uber.com/uber-big-data-platform/.

[2] Daniel Weeks: “Netflix: Integrating Spark at petabyte scale”, https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/43373.

[3] 別所祐太朗, 早水悠登, 合田和生, 喜連川優：“並列データベースシステムにおける入出力追跡による耐障害型問合せ実行方式の提案とパブリッククラウドにおける実験”, Web とデータベースに関するフォーラム 論文集, pp. 41–44 (2019).

[4] 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-I, **87**, 6, pp. 661–674 (2004).

[5] J. Dean and S. Ghemawat: “Mapreduce: simplified data processing on large clusters”, Commun. ACM, **51**, 1, pp. 107–113 (2008).

[6] S. Bartkowski, C. De Buitlear, A. Kalicki, M. Loster, M. Marczewski, A. Mosaad, J. Nelken, M. Soliman, K. Subtil, M. Vrhovnik, et al.: “High availability and disaster recovery options for DB2 for Linux, UNIX, and Windows”, IBM Redbooks (2012).

[7] M. A. Shah, J. M. Hellerstein and E. Brewer: “Highly available, fault-tolerant, parallel dataflows”, Proc. SIGMODACM, pp. 827–838 (2004).

[8] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker and S. Zdonik: “High-availability algorithms for distributed stream processing”, Proc. ICDEIEEE, pp. 779–790 (2005).

[9] J. Smith and P. Watson: “Fault-tolerance in distributed query processing”, Proc. IDEASIEEE, pp. 329–338 (2005).

[10] J. O. Hauglid and K. Nørvåg: “Proqid: Partial restarts of queries in distributed databases”, Proc. CIKM, ACM, pp. 1251–1260 (2008).

[11] B. Han, E. Omiecinski, L. Mark and L. Liu: “Otpm: Failure handling in data-intensive analytical processing”, Proc. CollaborateCom IEEE, pp. 35–44 (2011).