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評価試験と考察

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 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，当該手法を適用した問合せ処理機構の試作機をパブリッククラウド上に構築して最大 16 個の処理インスタンスを駆動する実験結果を示し，当該手法が正しく機能し再実行コストが削減されること，また，ストレージ律速のワークロードで実行オーバヘッドが無視できる例を実証する．

キーワード 並列データベース，耐故障性

1 はじめに

ビッグデータの応用は意思決定支援や知識発見をはじめとして多岐に渡る．その需要に牽引され，分析対象として蓄積されるデータの量は増大の傾向が見られ，近年では数十〜数百ペタバイト級の容量を擁する分析環境が構築される例も珍しくない．例えば，2013 年の時点で Facebook は 300PB を超えるデータウェアハウスの構築を報告しており [1]，現在に至るまで Uber [2] や Netflix [3] など様々なクラウドスケールの事業者が大規模分析環境の運用に関する知見を公開している．さらに，IoT 機器類のセンサ情報の蓄積が本格化している近年の状況 [4] を踏まえると，この傾向は今後も加速してゆくことが予想される．

大規模データベースには，多数の計算機（ノード）を並列に駆動し，更に演算間のパイプライン並列性の享受も図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．しかし，その構成ノードの多さから，問合せの実行中にいずれかのノードに予期せぬ故障が発生する確率が無視できない．故障に対する自明な対処法としては，正常に動作中のノードの実行状態を全て破棄して，問合せ処理を最初から再実行するものが考えられる．しかし，対象となるデータベースの巨大化が進む現在では，再実行に伴う余剰の実行時間及び運用コストが無視できないものとなっている．

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と分析系の読出し限定の問合せを対象に，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．また，実際に実機を構築して正常な動作を確認し，再実行コストの削減，そして，ストレージ律速のワークロードの一例で実行オーバヘッドが無視できることを実証した．

なお，著者は過去にストレージ入出力追跡を用いてノード障害発生時に処理を引継ぐ手法に関する提案と実験の結果 [5] を示している．本論文はこれを基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ものである．なお，本手法の理論的な性能オーバヘッドに関する考察 [6] も発表している．

本論文の構成は以下の通りである．第 2 章では並列データベースシステムと，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．第 3 章で故障発生後も処理の継続を可能にする問合せ処理手法を提案し，終端演算の二重化，入出力・実行状態のチェックポインティング，動的障害回復の 3 つの要素に分けて詳述する．第 4 章ではパブリッククラウド上で行った試験結果と考察を示す．第 5 章でデータベースシステム一般の障害回復に関連する既存の研究を概説し，第 6 章で本論文を総括する．

2 背景：並列データベースシステムと故障

2.1 共有ストレージ構成の問合せ処理機構

並列データベースシステムは，複数ノードを並列に駆動させて，大規模なデータベース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，商用で広く用いられているものは，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では，全ての処理ノードがストレージを共有する．この構成の利点としては，ストレージ管理とノード管理を分離できる利便性，記憶領域の利用効率の高さ，また，各ノードがストレージ空間全体に読書き可能であることで動的な負荷分散が可能であるといった点などが挙げられる．

2.2 並列データベースシステムの問合せ実行方式

本節では一般的な並列データベースシステムの問合せ実行方式を説明する．本論文では図 1 に示す共有ストレージ型の構成を前提とする．すなわち，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．処理ノードは入力表を読み出し処理を施す．各処理ノードの読出し位置は動的に決定される．ストレージは単一の読出しカーソルを管理し，データを要求したノードに提供すべき範囲を決定してデータを転送する [7].

一般の並列データベースシステム問合せの実行方式を図 2a に図解する．並列データベースシステムは，処理スループットの観点から，中間的な結果を逐一ストレージに永続化せず，演算をパイプライン化して実行する．

パイプライン中には，連続するステートレスな演算に続き，終端にステートフルな演算が配置される．ステートレスな演算とは，選択や射影などのように出力が入力の小さい範囲にのみ依存する演算であり，パイプライン化による並列性の享受が容易である．ステートフルな演算とは，出力が入力の広範囲な部分に依存するものである．例えば集約演算は，それまでに読出したデータ全てに依存するのでパイプラインを分断する．後続に演算を配置しても並列性は得られないため，パイプラインの終端に配置される．

各処理ノードはストレージに保存されている表の各行を読み出し，演算パイプラインに入力する．結合や集約などを含む問合せでは，演算間でデータが再配分されることがある．例えば，図 1 に示したパイプラインは 2 つの再配分を含み，一つのノードが読み出したデータが演算を施されながら全ノードに拡散する．

パイプライン終端のステートフルな演算を経た出力データは，各ノードに分散して保存される．

2.3 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多数のサーバ部品で構成されるため，問合せ処理中にいずれかのノードが故障する確率が無視できない．

並列データベースシステムにおいて故障が発生した際，単純に代替ノードを投入し，パイプラインを再開しても一般には正しい問合せ結果は得られない．その理由としては主に次の 2 つが挙げられる．

処理中間結果の揮発性． 並列データベースシステムは処理速度の観点から処理データを主記憶上で授受するため，ノード故障の際は中間の処理結果が失われる．特に，パイプライン終端に配置されるステートフル演算の結果は，典型的には処理開始から入力したデータ全体に依存するため，復元するにはこれまでの入力を再供給する必要がある．

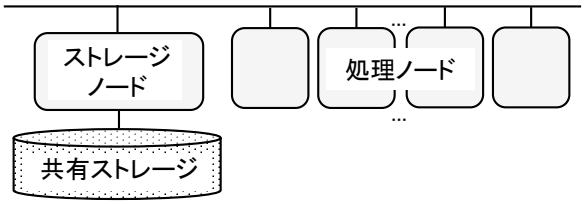


図 1: 共有ストレージ型並列データベースシステムのノード編成

障害発生時における処理進捗情報の不在． 障害発生時，各入力データがどの処理ノードに読み出され，どの演算を適用されたかに関する情報を各ノードは保持していないため，パイプライン再開後，欠落や重複のない処理を保証できない．

結果として，一般には障害が発生した際は問合せ全体を再実行する必要がある．第 1 章で述べた通り，現在はデータベースの巨大化が進んでおり，分析系の問合せには完了まで数日を要するものも珍しくない．したがって，再実行によって発生する時間コストも増大傾向にあるといえる．これにより，問合せが得られるまでの時間がユーザの許容域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 提案 : 動的障害回復を可能にする問合せ実行方式

3.1 前　　提

3.1.1 システム構成と障害モデル

図 1 のようなノード構成をもつ共有ストレージ型並列データベースシステムを前提とする．また，データパイプラインの配置は図 2 に示すようなものをであるとする．すなわち，各処理ノードは全てのパイプラインステージの演算を実行する．

処理ノードのうち 1 台の問合せ処理機構が予期せず停止し，演算の適用およびデータの授受が不可能になる障害を想定する．このとき，故障ノードの主記憶の内容が失われてもよい．

ストレージノードの障害，及びノード間ネットワークの障害は対象としない．また，想定する故障が発生した場合は，現実的な時間内でそれを検知し，全ての生存ノードに通知する手法が利用可能であると仮定する．

同時に故障する処理ノード数の上限は 1 台を想定し，障害回復中のさらなる故障は対象外とする．ただし，障害回復後であれば，対象となる障害が再度発生してもよい．

3.1.2 処理ノードの主記憶容量

ステートフル演算の処理結果，すなわちパイプラインの処理結果が主記憶に収まるか否かの予測は本来困難である．例えば，ハッシュ結合において主記憶に収まらないハッシュ表を形成する実行方式 [8,9]，およびその並列化手法 [10] はデータベースシステムの研究コミュニティにより様々に提案・利用されている．本稿においては，処理ノードが有する主記憶の容量は演算中及び演算後のデータが全て収まるものであると仮定する．

3.2 提案する問合せ実行方式

3.2.1 終端演算の二重化

2.3 節で述べた通り，一般に並列データベースシステムにおける中間的な処理結果は失われる．

提案する実行方式では，図 2b に示すようにパイプライン終端に位置するステートフルな演算を隣接するノードにおいても実行し，処理結果もノード間で冗長に保存する．これにより，単一のノード故障が発生した際にステートフル演算の結果を授受することで代替ノードに実行状態を復元でき，問合せ全体の再実行をせず，実行の継続が可能となる．

処理手順 図 2b における黄色に縁取られた番号を辿ることにより，提案手法によるデータ処理の過程を示す．橙に縁取られた番号に対応する処理は次の小節で解説する．

- （1）処理ノードの 1 つが行範囲（ここでは ID 101-200）をストレージノードを介して読み出す．
- （3）ステートレス演算の適用と再分配を経た行が，各ノードに分散する．図中では左端のノードに着目している．このとき，各ノードのステートフル演算の直前において，ある行範囲（図中では ID 101-200）に対応する処理データを待機させる．他の処理ノードでも同様に，該当行範囲に対応するデータを待機させる．
- （4）引き続き左端のノードに着目する．入力表の行範囲（ID 101-200）に対応するデータのうち，当ノードに行き届くべきデータが全て到着したら，待機させていたデータの複製を隣接ノードに送信する．他の処理ノードに対しても同様の処理を行う．
- （6）左端ノードは待機させていた処理データ（入力表の ID 100-201 に対応）を演算結果に適用する．他処理ノードに対しても同様．なお，この適用は後述する通り対応行範囲が「承認」状態に遷移してから実行する．
- （7）左端ノードに隣接し，処理データ（入力表の ID 100-201 に対応）の複製を持つノードが当データをステートフル演算結果の複製に適用する．他処理ノードに対しても同様である．

ステートレス演算を全て経た処理データを直接ステートフル演算に適用せず待機させる理由については，次小節で説明される．

3.2.2 入出力・実行状態のチェックポインティング

2.3 節で述べた通り，代替ノード投入の後，過不足なく正しい処理を行うには，故障発生時点での各ノードの処理進捗に関する情報が必要である．

提案する実行方式では，ストレージが読み出される各行範囲に対応する処理状況を「未読出」「処理中」「承認済」の 3 状態で管理する．

初期状態は「未読出」であり，処理ノードがストレージ（ノード）に入力表のデータの読み出しを要求すると，ストレージは入力表の「未読出」状態の行範囲から供給すべきものを動的に決定し，要求した処理ノードに転送する．

状態が前進する 2 つの契機は，図 2b において橙で縁取られた番号に対応する．

- （2）入力表の行範囲を処理ノードが読み出すと，初期状態「未読出」から「処理中」に遷移する．
- （5）読みだした行範囲に対して，全てのノードが（4）の操作，即ちステートフル演算前で結集したデータの隣接ノードへの複製を終えたとき,「処理中」から「承認済」に遷移する．「承認済」に遷移した行に対応するデータはパイプライン終端の演算に適用（6，7）されることが確定する．

なお，ストレージが（5）「処理中」から「承認済」への遷移を実行するために，各処理ノードは（4）の完了，即ち複製を終えた旨をストレージに通知する制御信号（以降「承認」と呼ぶ）を送信する．承認信号には複製したデータに対応する入力表の行範囲が付与されている．

従って，ストレージが行範囲に対応する処理ノードの数だけの承認信号を受信した時に,「承認済」への状態遷移を行うことができる.「承認済」の行範囲の処理結果は各ノードに冗長に保存されているので，単一のノードに障害が発生しても復元できる．したがって，ステートフル演算に適用しても結果の復元可能性が損なわれない．

ステートレス演算を全て経た処理データを直接ステートフル演算に適用せず待機させる理由は，ステートフル演算への適用を入力表の各行範囲，すなわち進捗の管理単位に基づいて原子的に行うためである．

ステートフル演算への適用を終えたデータを待機させず直接ステートフル演算に適用する場合，障害回復後に読み出されて到着するデータについて，障害前にステートフル演算に適用されていないもののみを選択する操作が必要となるが，これは一般に困難である．なぜならば，ストレージからのデータ読出し順が動的に決定すること，また，パイプライン中に複数回の再分配が存在しうることから，待機バッファへの各データの到着順は非決定的であるためである．

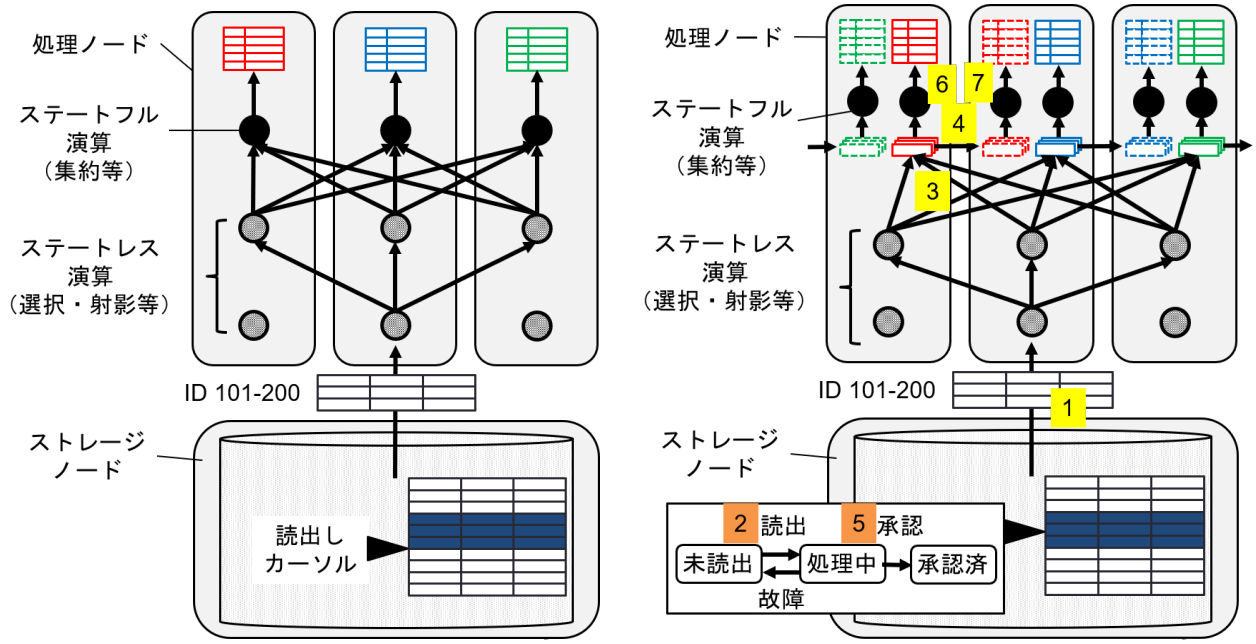
処理の継続，また，問合せ処理の正確さを同時に達成するために，提案する処理機構においてはステートフル演算の適用を連続した行範囲の単位で実行し，ストレージによる進捗状態の管理も同じ粒度で連動して行う．

入力表の行範囲の結集完了を検知するには，以下の 2 つが必要である．

- 到着するそれぞれの行がどの行範囲に相当するかを特定できる
- ある行範囲に対応する行がもう到着しないことを知ることができる

一番目の機能は，図 3a に示すように（「終端マーカ」は後述）ステートレス演算を通過する行，すなわち処理ノードに読み出され，ステートレス演算前のバッファに到着するまでの行に対して，処理中の行に対応する入力表の行範囲の識別情報を付与することで実現される．

パイプラインの始端に位置する，入力表をストレージから直接読み出す演算において識別情報を付与する方法は自明である．既にタグが付与された処理データが再分配されて更に次のステートレスな演算に適用される場合も，そのステートレス性



(a) 従来型（再実行型）　　(b) 提案型

図 2: 従来型（再実行型）および提案型の並列データベースシステムの問合せ実行方式.

から，識別情報を変更なく伝搬することで引き続き入力表の行との対応をとることができる．特に，二回以上の再分配を含むクエリブロックにおいては同じ行範囲に含まれる行が全ノードから送信されるため，識別情報の付与は必須である．

二番目の機能は，各ステートレス演算が出力の送信先ノードに対し，入力表の該当範囲に対応する出力データの送信が終了したことを表す信号を送信することで実現できる（この信号を今後「終端マーカ」と呼ぶ）．

パイプライン始端のステートレス演算は，入力表の行範囲をストレージから読み出して，データの末尾に終端マーカを付与する．演算の出力を他ノードに送信する際，状態管理の単位である入力表の行範囲に対応する処理データを全て送信してから，対応する終端マーカを送信する．出力を各ノードに再分配する場合は，終端マーカを他の全処理ノードにブロードキャストする．

２回目以降の再分配では，入力表の特定の行範囲に対応するデータが全ノードから全ノードに送信される．したがって，２回目以降の再分配を受け取る演算，すなわち３段目以降の演算は，ステートレス演算かステートフル演算かに関わらず，進捗管理単位に対応するデータを全ノードから受け取る．この場合，同じ行範囲に対応する終端マーカを処理ノードの数（N とおく）だけ受信して初めて，今後該当データが送られることはないと判断できる．

ステートレス演算の場合（図 3a）は，N 個の終端マーカ群を受信して初めて，次の演算に終端マーカをブロードキャストできる．受信した N 個の終端マーカ群を一つに併合してブロードキャストすることで，次段を流れるある行範囲に対応する終端マーカの数を N^2 個に抑えることができる．なお，図の処理ノード 3 への出力のように，選択演算を通してある送信先ノードに送信すべき行が零行である場合は，終端マーカのみを送信すればよい．

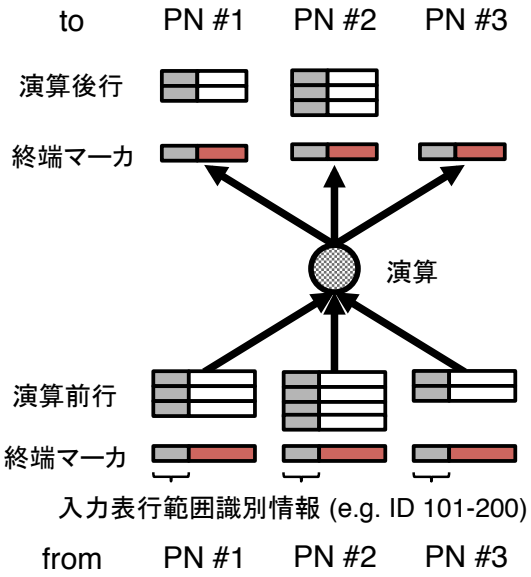
ステートフル演算の場合（図 3b）は，N 個の終端マーカ群を受信することで，進捗管理単位のデータの結集が完了したと判断でき，複製およびストレージへの承認を行うことができる．

なお，ある進捗管理単位（入力表のある行範囲）に対してある処理ノードがストレージに承認を送信した際，バッファに滞留しているデータは，他の処理ノードによる該当データへの承認が完了するまでステートフル演算に適用できない．処理ノードは，全ノードによる承認が完了したか否かを定期的にストレージに問い合わせることで，適用の可否を判断する．

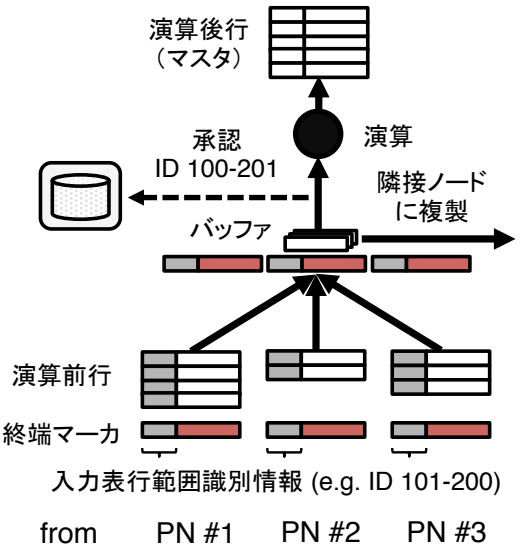
3.2.3 動的障害回復

いずれかの並列ノードに障害が発生した時は，処理パイプラインを停止し，以下のように実行状態を復旧する．

- 生存している各処理ノードは，ステートフル演算の直前のバッファに達していない処理データを全て破棄する．
- ストレージは,「未承認」状態になっている，すなわち処理ノードに読み出されパイプラインに取り込まれたが，対応する承認信号を送信していない処理ノードが一台でも存在する行範囲を「未要求」状態に戻す．これにより，処理再開後，該当行範囲を任意のノードが読み出すことができる．
- 各処理ノードは，ステートフル演算のマスタの前に保留している全てのデータのうち，対応する入力表の行範囲が「承認済」になっているものをストレージに問合せ，該当するデータをステートフル演算に適用する．該当しないデータ（「未承認」のデータ）は破棄する．
- 各処理ノードは，論理的に逆に隣接するノードのステー



(a) ステートレス演算



(b) ステートフル演算

図 3: 実行状態のチェックポインティングを可能にする処理データへのメタ情報付与．(a)(b) 各図では，入力において 2 回目以降の再分配が発生しており，進捗の管理単位である入力表行範囲（ここでは ID 100-201）に対応するデータが全ノードから到着する．”PN”は処理ノードを表す．

トフル演算の複製の前に保留している全てのデータのうち，対応する入力表の行範囲が「承認済」になっているものをストレージに問合せ，該当するデータをステートフル演算に適用する．該当しないデータ（「未承認」のデータ）は破棄する．

- 代替ノードを投入する．
- 代替ノードは論理的に隣接するノードから障害ノードの保持していたステートフル演算結果の複製を受け取る．
- 代替ノードは論理的に逆に隣接するノードから該当ノードのステートフル演算の複製を受け取る．

以上の動作を完了した後，通常の問い合わせ処理を再開することで，正しい問合せ結果が得られる．

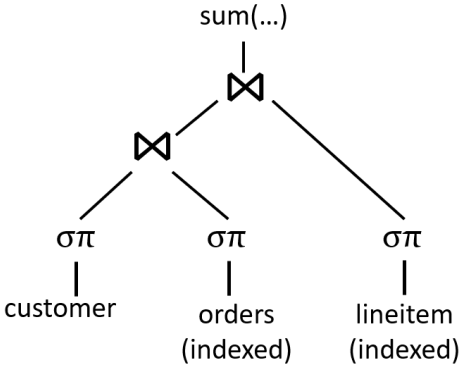
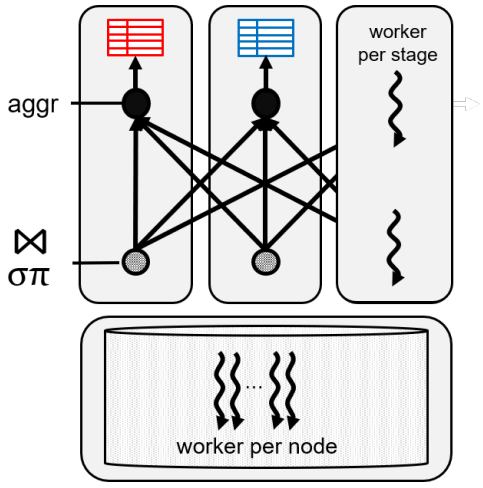
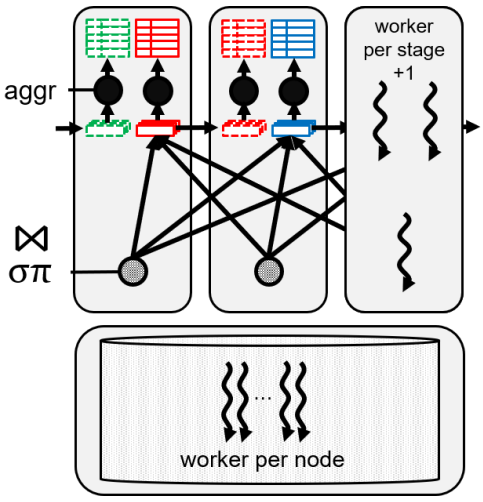


図 4: 評価用問合せの実行計画



(a) 従来型



(b) 提案型

図 5: 問合せのパイプライン構成及びスレッド配置

4 評 価 実 験

提案した処理方式（終端演算二重化，入出力・実行状態のチェックポインティング，動的障害回復）を適用しない場合とする場合の 2 種類の並列問合せ処理機構を実装し，パブリック

表 1: 実験環境の諸元

	処理ノード	ストレージノード
Instance	t2.medium	i3.xlarge
CPU	2 VCPUs	4 VCPUs
Memory	4 GB	16 GB
Storage	8 GB	950 GB
	(General purpose)	(Instance store)
OS	Amazon Linux 2	Amazon Linux 2

クラウド上に構築した．

4.1 実験環境

Amazon Web Services の提供するパブリッククラウド環境において，表 1 に諸元を示すノード群を図 1 に示す通りに相互接続した．

4.2 評価用問合せと進捗の追跡

Listing 1: 評価用問合せ

```
1  SELECT l_orderkey,
2  sum(l_extendedprice * (1 - l_discount)),
3  o_orderdate, o_shippriority
4  FROM customer, orders, lineitem
5  WHERE c_mktsegment = 'BUILDING'
6  and c_custkey = o_custkey
7  and l_orderkey = o_orderkey
8  and o_orderdate < 1995-03-15
9  and l_shipdate > 1995-03-15
10 GROUP BY l_orderkey, o_orderdate, o_shippriority
```

TPC-H ベンチマーク [11] の 3 表（customer，orders，lineitem の scale factor 100），及び orders 表の o_custkey 列と lineitem 表の l_orderkey 列に対する索引を生成し，共有ストレージに格納した．なお，各表は C 構造体の配列として，各索引は Berkeley DB [12] 18.1 の提供する B ＋木構造により作成した．

このデータセットに対し，Listing 1 に示す問合せ処理を索引結合を用いて行った．試作機に与えた実行計画は，1 つのパイプラインで構成される．問合せ木を図 4 に示す．

各処理ノードは customer 表を読み出し，選択を適用する．選択後の行の結合キーをストレージに送信し，結合行を問合せる．ストレージは索引を利用して該当キーを持つ orders 表の行を検索し，処理ノードに転送する．処理ノードは受け取った結合行に対して選択を適用する．続いて同様に，処理ノードは lineitem 表の結合対象の行をストレージから取得し，選択を適用する．射影及び結合処理の後，ハッシュベースの再分配を行い，各ノードで l_orderkey，o_orderdate，o_shippriority 行によるグループ化に基づく集約を行い，結果を蓄積する．

手法を適用する場合は，最外表である customer に対して 1000 行単位で入出力・実行状態チェックポインティングを行い，集約に対して二重化を適用した．実行パイプラインの構成の比較およびスレッド配置を図 5 に示す．手法の適用なし・ありに関わらず，ストレージノードには処理ノード数のスレッドを配置した．各スレッドは各処理ノードからのデータ読み出し及び各種制御信号を受信し処理する．処理ノードにおいては，手法を適用しない場合は各パイプラインステージに 1 スレッド，適用する場合はステートフル演算に更に 1 スレッドを追加し，複製及びチェックポインティングの動作を分担させた．

4.3 障害回復の動作挙動

処理ノードを 16 台駆動して問合せを実行し，実行開始から 720 秒の時点で 1 台のノードを停止し，その 5 秒後に代替のノードを投入した．代替ノードは処理ノードクラスタと接続を確立した後，論理的に両隣接する 2 ノードからそれぞれ約 80,000 行を約 1.4 秒で受信し，実行状態を回復した．その後は処理を継続し，900 秒以内に問合せ処理が完了し，正しい問合せ結果を得た．故障を生じさせた処理ノードの CPU 利用率，およびストレージノードの CPU 利用率とストレージの入力数とスループットを図 7 に示す．

CPU 利用率に着目すると，処理ノードは平均 1%以下程度をとっている．一方で，ストレージノードではほぼ常に 90%以上を保っており，うち 50%以上はストレージからの入力待ちであることから，明らかにストレージ読み出しが問合せ処理の律速要因になっていることが読み取れる．また，代替ノードの投入後 10 秒程度で読み出しスループットが回復していることが分かる．

4.4 障害回復時間の削減効果

16 台の処理ノードを用いて問合せ処理を実行している最中に単一ノード故障を発生させ，従来手法により最初から問合せ処理を再実行し直す場合と，提案手法によりその 5 秒後に同仕様のノードを追加投入して問合せ処理を継続させた場合の総実行時間を比較した．

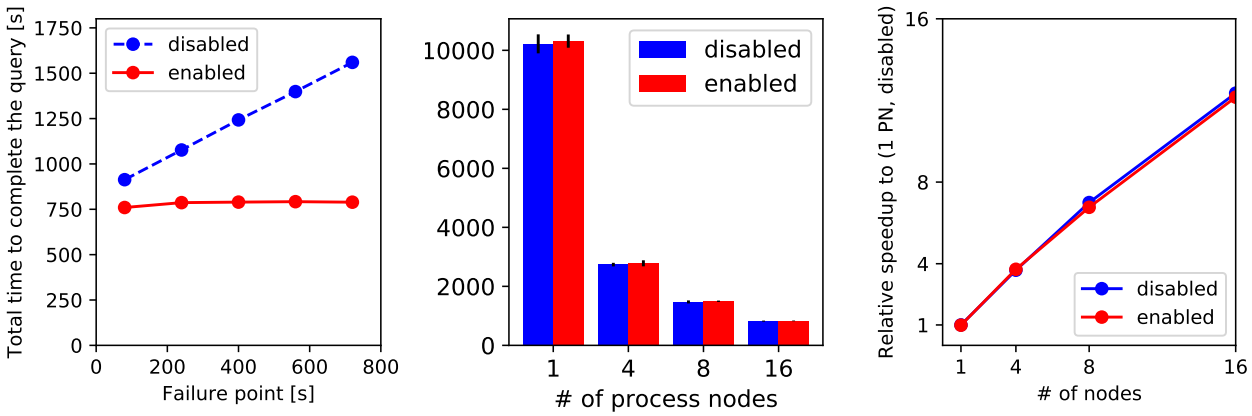
図 6a に結果を示す．従来手法では故障発生時の進捗率が高い程ペナルティが増大しているのに対して提案手法では殆どペナルティは観測されず，有効性が明らかである．例えば，故障時刻が処理開始後 720 秒後（進捗度約 90%）の場合は，総実行時間が約 45%減少している．

4.5 進捗追跡のオーバヘッド

障害を生じさせず問合せを実行した際の所要時間を，提案手法の適用なし・ありで比較した．1, 4, 8, 16 ノードにおける結果を図 6b に示す．なお，各パラメータについて 3 回ずつ測定を行い得られた平均値を測定値として採用した．提案手法の適用下において，オーバヘッド，すなわち提案手法を適用しない場合と比較した適用する場合の実行時間は最大 1.6%（ノード数 4）であった．

5 関連研究

問合せ処理の中止と再開を可能にする実行方式の研究は，自発的な中止を想定するものと予期せぬ中止を想定するものに大別される．前者には，各演算がチェックポイントを生成するこ



(a) 故障時の問合せ進捗率と実行時間の関係　(b) 進捗追跡オーバヘッド　(c) 処理ノード駆動数とスピードアップ比の関係

図 6: 耐障害型問合せ実行方式の性能特性と動的障害回復の効果．各グラフにおいて，系列”disabled”, ”enabled”はそれぞれ提案手法の適用なし・ありの結果を表す．(c) のスピードアップ比は，手法適用なし・処理ノード 1 台の実行時間を基準とする．

とで優先度の高い問合せの割り込み実行を可能にする試み [13] や，中間結果をキャッシュとして保持することで再計算を避けるもの [14] が挙げられる．本研究は後者に属するため，以下は後者を詳述する．

Hadoop をはじめとする MapReduce [15] 系の並列処理フレームワークは，問合せ処理における中間的な結果を各ジョブ毎に永続化するので，障害が発生した際は永続化した直近の状態を読み出し，実行状態を復元できる．しかし，余剰のストレージ入出力による性能オーバヘッドは並列データベースシステムと比較して無視できないものとなっている [16]．

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 [17]．ストリーミング処理を対象とする研究では，問合せ結果の正確さを犠牲にしつつ，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 [18, 19]．J. Smith は，障害回復に伴う重複データの排除を目的として，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 [20]．また，下流ノードが行番号単位で上流ノードの処理の進捗を追跡して重複を排除する手法も提案されている [21, 22]．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，ストレージ無共有構成を前提としていることから，入力表を複数ノードのストレージで冗長化し，障害発生時は生存ノードに保存された複製を代替ノードに転送する手順を必要としている．また，問合せの中間結果を処理を行わない補助的なノードに保存することから，台数に対する並列度が減少する．また，実装の詳細な議論および実機評価が未完であり，我々の提案手法にあるような処理の過不足を排除する処理ノードの協調法に関する議論が行われていない．

共有ストレージにおける動的障害回復をめざす研究の例は我々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と分析系問合せを対象に，問合せ処理パイプラインにおいて処理状態の追跡と処理結果の冗長化により，ノード故障が発生した際に他のノードが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した．

パブリッククラウド上に実装した試作機において最大 16 台の処理ノードを駆動して索引結合を実行した．ストレージ律速であることから実行オーバヘッドが無視できる程度であり，実際に再実行ペナルティが削減されることを示した．

今後は，多様な問合せを対象として評価実験の拡充を行う予定である．例えば，ハッシュ結合は今回実験の対象とした問合せとは異なり，複数のパイプラインで構成され，かつパイプライン中に複数の再分配が存在する．また，索引結合と比較して CPU 負荷が大きいため，今回観測できたものとは異なる性能特性が現れると考えられる．

文　献

[1] Dhruba Borthakur. Petabyte scale databases and storage systems at facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 1267–1268, New York, NY, USA, 2013. ACM.

[2] Reza Shiftehfar. Uber's Big Data Platform: 100+ Petabytes with Minute Latency. https://eng.uber.com/uber-big-data-platform/.

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

[4] The Internet of Things: Data from Embedded Systems Will Account for 10% of the Digital Universe by 2020 (online). https://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm.

[5] 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.

[6] 別所祐太朗, 早水悠登, 合田和生, 喜連川優. 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. 電子情報通信学会 データ工学研究会, pp. 43–48, 2019.

[7] 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, Vol. 87, No. 6, pp. 661–674, jun 2004.

[8] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, Vol. 1, No. 1, pp. 63–74, 1983.

[9] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pp. 1–8, 1984.

[10] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, pp. 110–121, 1989.

[11] The TPC-H benchmark. http://www.tpc.org/tpch/.

[12] Oracle Berkeley DB. https://www.oracle.com/database/berkeley-db/db.html.

[13] Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun Yang. Query suspend and resume. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pp. 557–568, 2007.

[14] Surajit Chaudhuri, Raghav Kaushik, Ravishankar Ramamurthy, and Abhijit Pol. Stop-and-restart style execution for long running decision support queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pp. 735–745, 2007.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, Vol. 51, No. 1, pp. 107–113, 2008.

[16] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pp. 165–178, 2009.

[17] 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.

[18] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proc. SIGMOD*, pp. 827–838. ACM, 2004.

[19] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proc. ICDE*, pp. 779–790. IEEE, 2005.

[20] Jim Smith and Paul Watson. Fault-tolerance in distributed query processing. In *Proc. IDEAS*, pp. 329–338. IEEE, 2005.

[21] Jon Olav Hauglid and Kjetil Nørvåg. Proqid: Partial restarts of queries in distributed databases. In *Proc. CIKM*, pp. 1251–1260. ACM, 2008.

[22] Binh Han, Edward Omiecinski, Leo Mark, and Ling Liu. Otpm: Failure handling in data-intensive analytical processing. In *Proc. CollaborateCom* , pp. 35–44. IEEE, 2011.

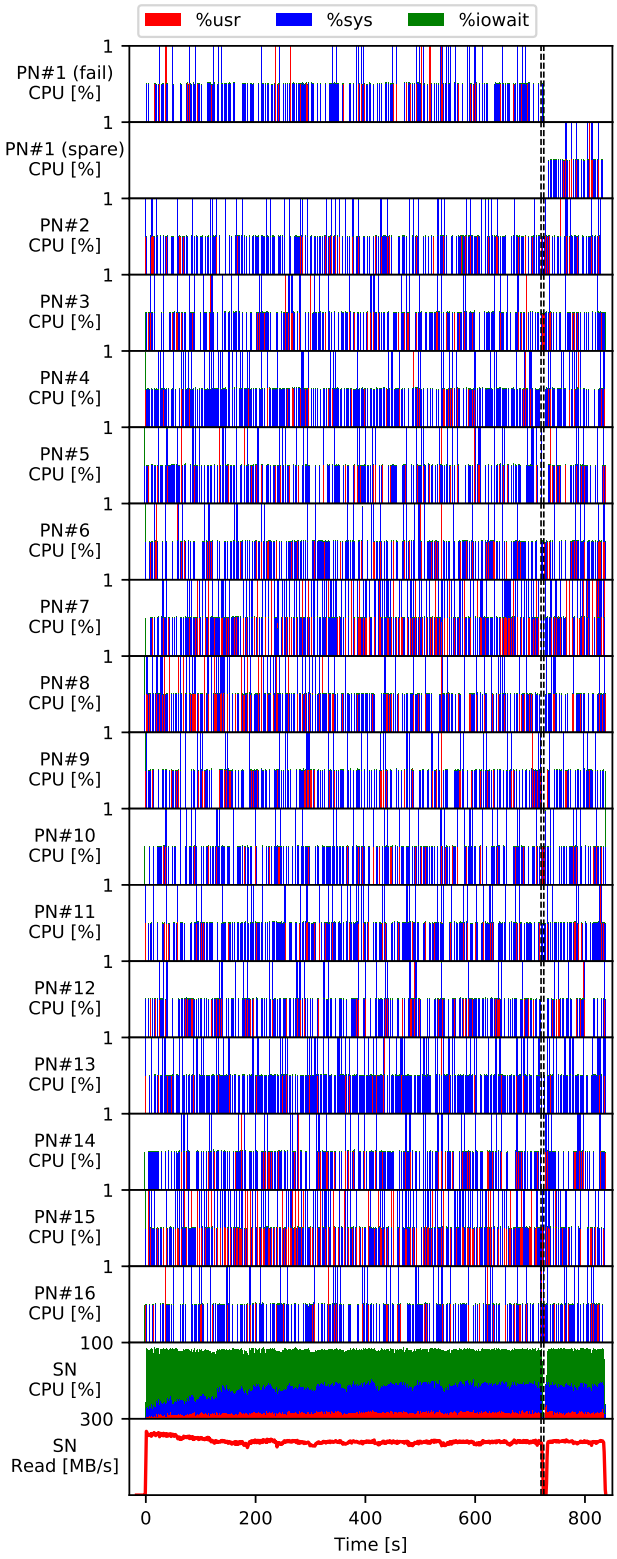


図 7: ノード故障時の動的障害回復の動作挙動．処理ノードの故障（開始 720 秒後）および代替ノードの投入（開始 725 秒後）時点を点線で示す．上段よりそれぞれ故障処理ノード，代替処理ノード，無故障 15 処理ノードの CPU 利用率，ストレージノードの CPU 利用率，ストレージノードの読出しスループットである．各ノードの CPU 利用率は，各論理 CPU 利用率の平均値として算出している．