修士論文

Apache Hudi における ELT パイプラインのスケジューリング手法の提案と株式市場データへの適用

長谷部 陽

主指導教員 井口 寧

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

2026 年 3 月

Abstract

In a lakehouse, near real-time analytics depends on how fast continuously arriving stream data is reflected in the downstream tables. In many lakehouse environments, only the data newly appended or updated can be efficiently collected using incremental queries. However, when those incremental records are spread across many files, incremental query still has to process all those files. As a result, the execution time of ELT(Extract-Load-Transform) job that uses incremental query will increase, and the freshness of the downstream table degrades.

In this thesis, we propose Partitioned-Max-Benefit, a scheduling method that allows an ELT job to process a subset of those files at each execution when beneficial. For each execution, the scheduler enumerates a limited set of candidate subsets, estimates the staleness reduction and the execution time, and chooses the subset that maximizes marginal benefit (estimated staleness reduction per estimated execution time). Then, jobs are greedily prioritized using these maximized values, so that the cost of ELT for low-benefit files can be deferred when resources are constrained.

Assuming data analytics in a securities company, we implemented the scheduler and lakehouse using Apache Hudi and evaluated with a six-job ELT pipeline using IEX U.S. equity market data. In our experiment, the proposed method achieved a lower total staleness than Random scheduling and Max-Benefit in our setting.

目 次

図 目 次

表 目 次

第1章　はじめに

近年、リアルタイムなデータ分析の需要が高まっている。そのためには、絶えず供給されるストリームデータを過去データを含めた分析用の集計結果に反映し続ける必要がある。例えば証券会社では、株価のティックデータや板情報などの市場データを基に、市況変化に即した意思決定を行っており、この分析用の集計結果が新しいデータを反映している状態を維持することが重要である [1, 2]。

このような需要に対し、産業界ではレイクハウス [5] がデータ基盤の中核として普及している。レイクハウスでは、ストリームデータはテーブルとして管理され、複数のデータ変換処理（ELT ジョブ）が連鎖するパイプラインを通じて段階的に整形・集計される [7]。多くのレイクハウス実装では増分クエリが提供され、前回処理以降に追加・更新されたデータのみを効率的に取得できる [8]。

しかし、増分クエリは常に効率よく動作するわけではない。レイクハウスではテーブルのデータは複数のファイルに分散して保存されるため、更新対象の主キーが多数のファイルにまたがる状況では、増分クエリであっても多くのファイルを参照せざるを得ず、実行時間が増加しやすい。結果として、限られた計算資源の下では ELT ジョブの実行が滞り、下流テーブルへのデータの反映が遅れるといった課題がある。

データ鮮度の改善を目標とした既存手法として Max-Benefit[10] があるが、同手法はジョブが増分データ全量を処理する前提で marginal-benefit を評価する。したがって、増分データが多数ファイルに分散し、ファイルごとに読取り効率が大きく異なるレイクハウス環境では、一部ファイルのみの処理が有利になり得るにもかかわらず、それを選択できない可能性がある。

本研究の目的は、ストリームデータを増分処理によって取り込むレイクハウスにおいて、増分データが複数のファイルに分散して保存される状況を考慮したスケジューリングにより、読み取り効率を維持しつつデータ鮮度を向上させることである。

本論文の構成は以下の通りである。第 2 章では、本研究の背景として、レイクハウスにおける増分処理の仕組みと、増分データのファイル分散が実行時間に与える影響、および既存の ELT ジョブのスケジューリング手法の前提と限界を整理し、課題を定式化する。第 3 章では、課題への解決策として、ファイル部分集合に対して marginal-benefit を評価するスケジューリング手法 Partitioned-Max-Benefit を提案する。第 4 章では、Apache Hudi を用いた実験環境と評価方法を示し、結果と考察を述べる。第 5 章で本研究のまとめと今後の課題を述べる。

第2章　研究の背景と目的

2.1　はじめに

本章では、提案手法の理解に必要な背景を整理する。具体的には、レイクハウスにおける増分処理の仕組みと参照ファイル数が読取りコストに与える影響、データ鮮度を目的とした既存のスケジューリング手法（Max-Benefit）の前提と限界を述べ、本研究が扱う課題を明確化する。それを踏まえ、参照ファイル毎の処理効率の違いに着目した新たなスケジューリング手法の必要性を述べる。

2.2　研究の背景

レイクハウスの概要

レイクハウスは、Databricks によって提唱されたデータマネジメントアーキテクチャである。S3 や HDFS などの分散ストレージ上で、ACID トランザクション、データ更新・削除クエリ、スキーマ進化、更新履歴の管理といった従来の DWH が提供してきた機能を提供することで、スケーラブルかつ信頼性の高いデータ基盤を提供している。

本研究では、具体的な実装・評価は代表的な OSS である Apache Hudi を用いて行う。以下ではまず一般的な前提を整理し、その具体例として Hudi の仕組みを述べる。一方、分散ストレージ上のファイルはイミュータブルであり、Parquet などの列指向ファイルフォーマットではレコード単位での更新や履歴管理は想定されていない。この制約のもと、多くのレイクハウス実装では、ACID トランザクションを実現する手段として、ファイル単位のスナップショットに基づく MVCC（Multi-Version Concurrency Control）が採用されている。

MVCC を用いることで、各トランザクションはバージョン管理されたファイルの集合から一貫したスナップショットを構成できる。これにより、分散ストレージ上でも高い同時実行性とデータ整合性を両立しつつ、主キーに基づくデータ更新や履歴管理をスケーラブルに実現できる。

Apache Hudi におけるファイル管理と増分クエリの仕組み

Apache Hudi では、テーブル上のデータは file group と呼ばれる単位で管理される。file group は、テーブルのある主キー集合に対応するレコードを保持する単位であり、ファイルシステム上のパス、オブジェクトストレージ上のプリフィックスとして実装される。file group には、MVCC のため異なるコミット時点で生成された異なるバージョンのファイルが複数保存されており、各ファイルにはそのコミット時点でその file group に属する全レコードが含まれている。テーブルへのレコード追加・更新では、そのレコードの主キーに対応する file group の最新バージョンのファイルを元に、その変更を加えた新しいファイルが作成される。この仕組みにより、ひとつのファイル内には異なるコミット時刻のレコードが含まれることになり、この点が後述する増分クエリの読み取り効率に影響を与える。

各 file group およびファイルのメタデータは Apache Hudi によって管理されており、これには file group の構成、ファイルのコミット時刻、が含まれる。増分クエリ実行時における、Apache Hudi の内部処理の流れは下記に示す。

1. クエリで指定された開始コミット時刻をもとに、Apache Hudi が先述のメタデータを用いて、追加・更新されたレコードを持つ file group を特定する。
2. 各 file group について、指定された開始コミット時刻以降に生成されたファイルのうち、最新のファイルを特定する。
3. 特定されたファイルの集合のパスを、Apache Spark などの分散クエリエンジンに対して入力として渡す。
4. クエリエンジンは、各ファイルを読み取り、レコード単位でコミット時刻を判別することで、指定された開始コミット時刻以降に作成または更新されたレコードのみを抽出する。

このように Apache Hudi では、増分クエリは指定したコミット時刻以降に変更が加えられたレコードを、全ファイルの読取りを避けて効率的に取得できる。

増分クエリにおける読み取り効率の特性

従って、増分クエリには次の特性が存在することとなる。

1. ファイルごとに増分データの読み取り効率が大きく異なる

   指定した開始コミット時刻以降に多くのレコードが作成・更新されたファイルほど、1 つのファイルを読み取ることで取得できる増分データ量は多くなる。一方で、作成・更新対象となったレコード数が少ないファイルであっても、増分クエリでは同様にファイル全体の読み取り処理が必要となるため、1 つのファイルを読み取ることで取得できる増分データ量は少なくなる。

2. ファイルにおける増分データの読み取り効率が時間と共に改善される

   レコードの作成・更新が時間と共に file group に蓄積することで、同一ファイル内に含まれる増分データの割合が時間とともに増加する。さらに、ストリームデータにおける更新対象の主キー分布や新規ファイルの生成状況は時間とともに変化するため、どのファイルが効率的に増分データを提供するかは動的に変動する。

結果として、増分クエリの実行時間は、同じ増分データを読み取る場合であっても、実行時の増分データがいくつのファイルに分散して保存されているかによって大きく変動するという特性を持つ。

主キーのホットスポットに基づいたパーティション設計によって、更新時に参照されるファイル数を一時的に抑えられる場合がある。しかし、主キーの更新頻度は時間とともに変動するため、ある時点で最適なパーティション設計が将来的にも有効であるとは限らない。さらに、主キーが広範に分布するワークロードでは、偏りを利用したパーティショニングがそもそも成立せず、参照ファイル数を十分に削減することはできない。このように、書き込み方式の最適化によるアプローチには限界があるため、本研究では読み取り側での最適化を検討する。

ジョブスケジューリングに関する研究

本研究の評価指標（2.2 で後述）は陳腐度の時間積分であり、スケジューリングの実行判断の影響は将来にわたり累積する。しかしストリームデータの到着は事前に確定しないため、計測期間全体に対する最適な実行計画を求めることは一般に難しい。そこで本研究では，観測可能な情報に基づく逐次的なヒューリスティック方針を採用する。

その代表例として Max-Benefit という手法がある。Max-Benefit は、選択肢に対して目的関数に対する効果量を実行コストで割った実行効率を評価し、その値が最大となるものを優先して実行する貪欲的なアプローチである。ELT ジョブのスケジューリングにおいても Max-Benefit の適用が提案 [10] されているが、先述のレイクハウスにおける増分クエリにおけるファイル分散によるジョブの実行時間の増加・変動を加味していない。そのため、増分データ全量を不可分な単位として扱う既存の選択肢の中では、ファイルごとの読取り効率のばらつきを十分に反映できず、レイクハウス環境ではデータ鮮度の最適化が十分に達成できない可能性がある。

2.3 目的

本研究では、レイクハウス上の ELT ジョブのスケジューリング手法を提案する。各 ELT ジョブは処理としては独立に定義・実行されるが、ELT パイプライン上で

は次の点で互いに依存関係を持つ。第一に，計算資源が有限であるため、全ジョブを同時に実行することはできず、資源競合のもとで実行順序の選択が必要となる。第二に，ある ELT ジョブの出力は後続ジョブが参照するテーブルの内容を更新し、後続ジョブが各時点で処理可能なデータに影響を与える。したがって、実行順序の違いは各テーブルの更新時刻および各時点で存在するデータの状態を変化させ，結果としてパイプライン全体のデータ鮮度に差を生じさせる。

最適化目標の前提として、ある時点における任意のテーブルのデータの陳腐度を、現在時刻と各テーブルに反映されているストリームデータの最新到着時刻との差として定義する。陳腐度はデータ鮮度の逆指標であり、陳腐度が小さいほど鮮度が高い。このデータの陳腐度に関して、システム運用における時間の経過に伴う更新先テーブル群の累積値を最小化する。なお本研究では、ストリームデータの入力テーブルはポーリングにより継続的に更新され、スケジューラが実行順序で鮮度を改善する対象ではないため評価対象外とする。したがって、陳腐度の評価は入力テーブルを除く更新先テーブル群に対して行う。また、イベント発生から到着までのレイテンシや順序の乱れは扱わない。

提案においては、レイクハウス上の増分クエリで主キー分布に起因して増分データが多数のファイルに分散し、参照ファイル数およびジョブ実行時間がコミットごとに変動しやすいという増分処理特有の性質を踏まえる。その上で、増分データが多数ファイルへ分散する状況でも、限られた計算資源で下流テーブルのデータ陳腐度を低く維持できるスケジューリングを実現することを目指す。本研究の新規性は、従来テーブル単位で行われてきた ELT ジョブスケジューリングに対し、増分データを含むファイルの部分集合に対して marginal-benefit を評価し、処理対象とするファイルを実行時に選択するスケジューリング手法（Partitioned-Max-Benefit）を提案する点にある。提案手法により、増分データが多数のファイルに分散して保存されるレイクハウス環境においても、限られた計算資源で ELT パイプライン全体のデータの陳腐度を低減できることを Apache Hudi を用いた実験により確認する。

2.4 関連研究

2.4.1 ELT ジョブのスケジューリング

ELT パイプラインにおけるジョブスケジューリングは、データが分析結果に反映されるまでのレイテンシを削減することを目的として、いくつかの研究がある。特に、ストリーミングデータウェアハウスを対象とした研究では、限られた計算資源でデータ鮮度を最大化するためのスケジューリング手法が提案されている。

Golab らの研究 [10] ではデータ鮮度の逆指標としてデータの陳腐度を次のように定義している。データウェアハウス上のテーブル T_i に含まれるレコードの最新のタイムスタンプを t_i^{latest} とおく。このとき、時刻 τ におけるテーブル i のデータの陳腐度 $S_i(\tau)$ は、次式で定義している。

$$S_i(\tau) = \tau - t_i^{\text{latest}} \tag{2.1}$$

そして、システム全体のパフォーマンスを評価する性能指標 P として、各テーブルのデータの陳腐度の積分和を用いている。

$$P = \sum_{i=1}^{n} \int S_i(\tau)\, d\tau \tag{2.2}$$

彼らの提案する Max-Benefit [10] では、各ジョブの実行によって削減されるデータの陳腐度の量を推定処理コストで割った値を marginal-benefit と定義し、この値が最大となるジョブを優先的に実行する。この貪欲的なスケジューリングにより、ランダムな実行ジョブの選択と比較して、性能指標 P を最大で 13 %低減できることが示されている。

ただし、Max-Benefit では、ジョブが増分データ全量を処理対象とすることを前提に marginal-benefit を算出している。そのため、増分データが複数のファイルに分散して保存されており、増分クエリのコストがファイルによって異なるレイクハウス環境では、その一部のみを処理対象とすることで、高い marginal-benefit が得られる場合がある。しかし、Max-Benefit は、全量の前提によってこのような実行戦略を選択できず、結果として達成可能な marginal-benefit を過小評価する可能性がある。

2.5 解決すべき課題

2.5.1 ファイル集合に対する marginal-benefit の定義

レイクハウス環境において増分データが複数のファイルに分散して保存されていることを前提とし、ELT ジョブを増分データを含むファイル集合に対する処理として捉え、その marginal-benefit を定義する。

ジョブ j の入力テーブルに対して、増分データを含むファイル集合を

$$F_j = \{f_1, f_2, \ldots, f_m\}$$

とする。スケジューラは、候補として選択したファイル集合 $X \subseteq F_j$ を処理対象とする。ジョブ j が前回までに反映済みの最新到着時刻を t_j^{done} と定義する。また、ファイル集合 X に含まれる増分データの最新到着時刻を $t_j^{max}(X)$ とする。この時、X を処理したときの陳腐度の削減量は

$$G_j(X) = t_j^{max}(X) - t_j^{done} \tag{2.3}$$

となる。

候補集合 X を処理すると、対象テーブルに反映されている最新到着時刻が t_j^{done} から $t_j^{max}(X)$ に更新される。これにより、評価時刻 τ における陳腐度 $S_i(\tau)$ は $t_j^{max}(X) - t_j^{done}$ だけ減少するため、$G_j(X)$ を陳腐度の削減量として用いる。スケジューラはこの効果量を処理コストで割った $\eta_j(X)$ が大きいジョブを貪欲に実行する。

処理コストについては次式で近似する。

$$E_j(X) = a_j + b_j \cdot \sum_{f \in X} size(f) \tag{2.4}$$

ここで a_j はジョブ起動などの固定オーバーヘッド、b_j はファイルサイズに比例して増加する係数である。これらの値は固定値ではなく、利用クラスタにおける過去のジョブ実行履歴から推定した値を用いる。具体的には，複数回の実行について、入力として読取り対象となったファイル集合の合計サイズ $\sum_{f \in C_j} size(f)$ を説明変数、ジョブの実行時間を目的変数として最小二乗の線形回帰を行い、切片を a_j，傾きを b_j として推定する。

したがって，a_j, b_j は CPU 性能やストレージ I/O 帯域，ネットワーク帯域などのハードウェア性能、および、ワークロードの影響を受けて変化し得る。そのため、異なる環境に適用する場合には、同様の手順で a_j, b_j を再推定する必要がある。

これらを用い、ジョブ j の候補集合 X に対する marginal-benefit は下記のように表せる。

$$\eta_j(X) = \frac{G_j(X)}{E_j(X)} \tag{2.5}$$

2.5.2 部分集合処理による効率改善の例

例えば、あるジョブ j の増分データを含むファイル集合が $F_j = \{f_1, \ldots, f_{30}\}$（各ファイルサイズ 128 MiB）であり、前回反映済みの最新到着時刻が $t_j^{done} = 9{:}30$ であるとする。この時点で、全ファイルを処理すれば $t_j^{max}(F_j) = 9{:}55$ まで前進できる一方、特定の 3 ファイルのみを処理する部分集合 $F' \subset F_j$ を選ぶと $t_j^{max}(F') = 9{:}50$ まで前進できるとする。

このとき効果量は

$$G_j(F_j) = 9{:}55 - 9{:}30, \qquad G_j(F') = 9{:}50 - 9{:}30$$

である。一方、処理コストは 2.5.1 節で定義した近似式（式 (2.4)）を用いて、

$$E_j(F_j) = a_j + b_j \cdot (30 \times 128), \qquad E_j(F') = a_j + b_j \cdot (3 \times 128)$$

と表せる。したがって、Max-Benefit において増分データが含まれるファイル全てを処理する場合の marginal-benefit は

$$\eta_j(F_j) = \frac{G_j(F_j)}{E_j(F_j)}$$

である。一方、特定の 3 ファイルまでのファイルに含まれる増分データを対象とする場合、marginal-benefit は

$$\eta_j(F') = \frac{G_j(F')}{E_j(F')}$$

である。ここで $G_j(F') < G_j(F_j)$ であっても、スキャン量が $30 \times 128\,\mathrm{MiB}$ から $3 \times 128\,\mathrm{MiB}$ へ大きく減るため、

$$\eta_j(F') > \eta_j(F_j)$$

となり得る。すなわち、少量の増分データしか含まないファイルを一時的に除外し、短時間で大きな陳腐度削減を得られるファイルのみを処理することで、単位コスト当たりの陳腐度削減効率を高められる場合がある。

以上の議論より、本研究が対象とするレイクハウス環境においてデータの陳腐度を効率的に削減するためには、従来の ELT ジョブスケジューリング手法では満たされていない以下の要件を満たす必要がある。

- ELT ジョブをテーブル単位の不可分な処理として扱うのではなく、増分データを含むファイルをより細粒度な処理単位として扱うこと。
- 各ファイルにおける増分データ量や読取りコストの違いを反映し、ファイル集合の部分集合に対する marginal-benefit を評価すること。

次章では、これらの要件を満たすために、増分データを含むファイル集合に対して marginal-benefit を評価し、処理対象とするファイル集合を動的に選択するスケジューリング手法 Partitioned-Max-Benefit を提案する。

2.6 まとめ

本章では、レイクハウス環境における増分処理と ELT パイプラインを対象として、データ鮮度の最適化に関する背景と課題を整理した。レイクハウスでは、ファイル単位の MVCC に基づく増分処理が採用されており、テーブルからの増分データが分散するファイルの数や構成に応じて、増分クエリの処理効率や実行時間が大きく変動するという特性を持つ。既存の Max-Benefit 手法では、テーブルの増分データ全量を一括処理することを前提に marginal-benefit を評価するため、増分データが多数のファイルに分散する状況において、本来達成可能な陳腐度の削減効率が過小評価されるという課題があることを確認した。そして、テーブル単位という従来の処理単位の前提を見直し、増分データの分布や処理効率のばらつきを反映可能な、より細粒度なスケジューリング手法が必要であることを明確にした。次章では、この背景を踏まえデータの陳腐度の削減を目的に Partitioned-Max-Benefit 手法を提案する。

第3章　ファイル単位の部分集合選択によるELTジョブスケジューリング手法

3.1　はじめに

本章では、Apache Hudi 上の増分 ELT パイプラインを対象に、限られた計算資源下で更新先テーブルの陳腐度を効率よく削減するスケジューリング手法 Partitioned-Max-Benefit を提案する。提案手法は、ジョブごとの増分データを含むファイル集合から処理対象とする部分集合を選択し、その効果と実行コストの比（marginal-benefit）に基づいて実行計画を決定する点に特徴がある。

以降では、3.2 節でシステムモデルを述べた後、3.3 節で提案手法の概要を示し、3.4 節でアルゴリズムの詳細を説明する。最後に 3.5 節で実装上の入出力と設定を整理し、3.6 節で期待する効果を述べる。

3.2　システムモデル

本研究が対象とするシステムモデルでは、スケジューラは常駐プロセスとして動作し、増分データの発生をトリガとしてスケジューリングを繰り返すオンラインスケジューリングである。この処理機構のフローチャートを図 3.1 に示す。図中のスケジューラーは、Apache Hudi のメタデータ参照、実行計画の作成、および Spark ジョブの起動を担う常駐プロセスを指す。スケジューラーは Hudi メタデータから入力情報（各ジョブの増分データを含むファイル集合 F_j とそのメタデータ）を取得し、スケジューラー内部の実行計画作成アルゴリズムが実行ジョブと処理対象ファイルを決定して Spark ジョブを起動する。ジョブの内部で、変換処理を行ったデータのコミットによりメタデータが更新され、次のスケジューリングの入力となる。

なお、本研究ではスケジューラーを実装しており、その内部で動作する実行計画作成アルゴリズムが提案の核となる。

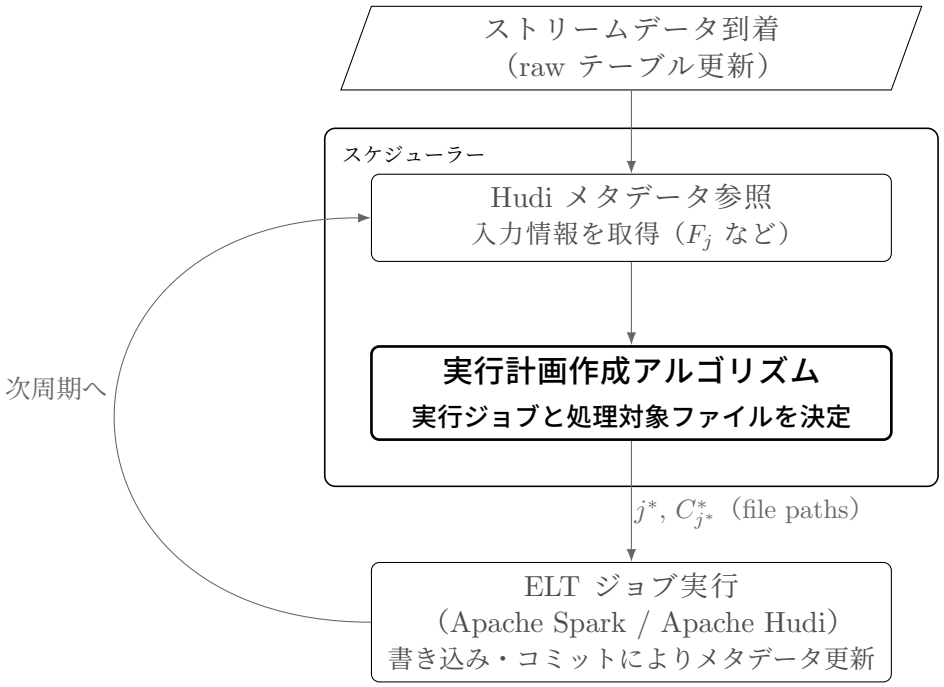

図 3.1: 本研究で扱うシステムモデルにおけるフローチャート。本研究の中でスケジューラーを実装した。実行計画作成アルゴリズムが提案の核となる。

3.3 提案手法：Partitioned-Max-Benefit

3.3.1 提案の要点

提案手法 Partitioned-Max-Benefit は、Max-Benefit におけるジョブ評価を、増分データを含むファイル集合 F_j の全量から部分集合へ拡張する点に特徴がある。Max-Benefit では各ジョブ j を増分データ全量を処理する不可分な単位として評価するのに対し、提案手法では F_j から構成される部分集合 $C \subseteq F_j$ を評価し、marginal-benefit $\eta_j(C)$ を最大化する C^*_j とその値 η^*_j を求める。ジョブ間の優先度付けは Max-Benefit と同様に η に基づく貪欲戦略を踏襲するが、用いる評価値を全量の η_j ではなく部分集合上で最大化した代表値 η^*_j に置き換える点が従来手法と異なる。

3.3.2 候補部分集合の制約

データベースとして最新到着時刻がある値まで進んだと言うためには、それ以前に到着したデータが欠落していないことが重要である。したがって本研究では、各レコードの到着時刻に基づき、ある時刻のデータが欠落していないことを反映済み時刻の整合性条件とし，この条件を満たす候補に限定する。

これを踏まえ、本研究では，ジョブ j の処理対象を取り込み上限時刻 u で表現する。すなわち，候補となる u を各未処理ファイル $f \in F_j$ の $t_j^{max}(f)$ から構成し，$U = \mathrm{sort}(\{t_j^{max}(f) \mid f \in F_j\})$ を候補集合とする。

各 $u \in U$ に対し，時刻 u までの未反映増分を含みうるファイル集合を

$$C_j(u) = \{\, f \in F_j \mid t_j^{min}(f) \leq u \,\}$$

として定義し，$C_j(u)$ を評価候補として列挙する。

このとき候補数は $|U| \leq |F_j|$ であり，ファイル部分集合の全列挙に比べて組合せ爆発を回避できる。また，u を増やすほど G_j は増加し，$C_j(u)$ に含まれるファイル数（したがってスキャン量）も増加しうるため，陳腐度削減量とコストのトレードオフを候補列挙により探索できる。

3.4 アルゴリズム詳細

スケジューリング周期における一連の処理を Algorithm1 に示す。
各スケジューリング周期において、スケジューラは投入可能ジョブ集合 J を得た上で、各ジョブ $j \in J$ について未処理ファイル集合 F_j と、各ファイルの属性 $t_j^{max}(f)$ および $size(f)$、ならびに反映済み最新到着時刻 t_j^{done} とコストモデル係数 a_j, b_j を入力として受け取る。次に、各ジョブ j について各ファイル $f \in F_j$ の $t_j^{max}(f)$ を収集し，それらを昇順にソートした候補上限時刻の集合 $U = \mathrm{sort}(\{t_j^{max}(f) \mid f \in F_j\})$ を得る。そして各 $u \in U$ を候補として，時刻 u までの未反映増分を含みうるファイル集合 $C_j(u) = \{\, f \in F_j \mid t_j^{min}(f) \leq u \,\}$ を構成して列挙する。各候補 $C_j(u)$ に対して、期待される陳腐度削減量 $G_j(C_j(u))$ と実行時間推定 $E_j(C_j(u))$ を計算し、marginal-benefit $\eta_j(C_j(u)) = G_j(C_j(u))/E_j(C_j(u))$ を求める。そしてジョブ j の marginal-benefit を最大化する候補 C_j^*、その値を η_j^* として記録する。この結果として各ジョブの代表値 η_j^* と対応する入力範囲 C_j^* が得られる。スケジューラは得られた L を用い、η_j^* の降順に資源制約の許す範囲でジョブを投入する。具体的には、空き実行枠数を $slots = K - |R|$ とし、L を η_j^* の降順にソートした列 U を作る。U の先頭から $slots$ 件の (j, C_j^*) を選んで今周期の投入集合 D とする。

3.4.1 入出力と評価量の定義

以下では、Partitioned-Max-Benefit が用いる入出力と評価量を定義する。本手法が扱う各コンポーネント間の入出力について表 3.1 で整理する。表中の取得元は情報源の種別を示すものであり、具体的な取得方法および実装上の扱いは 3.5 節で述べる。

表 3.1: Partitioned-Max-Benefit でのコンポーネント間の入出力

記号	種別	取得元	内容
F_j	入力	メタデータ	ジョブ j の増分データを含むファイル集合
$t_j^{min}(f)$	入力	ファイル統計情報	ファイル f に含まれる到着時刻の最小値
$t_j^{max}(f)$	入力	ファイル統計情報	ファイル f に含まれる到着時刻の最大値
$size(f)$	入力	メタデータ	ファイル f のサイズ
a_j, b_j	入力	スケジューラ設定	コスト推定係数
t_j^{done}	入力	スケジューラ状態	反映済み最新到着時刻
K	入力	スケジューラ設定	最大同時実行数
R	入力	スケジューラ状態	実行中ジョブ集合（空きスロット算出に用いる）
C_j^*	出力	スケジューラ	$\eta_j(C)$ を最大化するファイル集合のファイルパス
η_j^*	出力	スケジューラ	$\eta_j(C_j^*)$（実行計画作成時点で達成できるジョブ j の marginal-benefit の最大値）
D	出力	スケジューラ	今周期に投入するジョブと入力ファイル集合の組（例：$D = \{(j, C_j^*)\}$）

Algorithm 1 Partitioned-Max-Benefit における 1 周期の処理手順

Require: 投入可能ジョブ集合 J

Require: 各ジョブ j について、未処理ファイル集合 F_j と各ファイル $f \in F_j$ の $t_j^{min}(f)$, $t_j^{max}(f)$, $size(f)$, 前回反映済みの最新到着時刻 t_j^{done}, コストモデル係数 a_j, b_j

Require: 最大同時実行数 K, 実行中ジョブ集合 R

Ensure: 今周期に投入する集合 $D = \{(j, C_j^*)\}$（$|D| \le K - |R|$）

$L \leftarrow \emptyset$
for all $j \in J$ **do**
  $U \leftarrow \mathrm{sort}\big(\{\, t_j^{max}(f) \mid f \in F_j \,\}\big)$
  $\eta_j^* \leftarrow -\infty,\ C_j^* \leftarrow \emptyset$
  for all $u \in U$ **do**
    $C_j(u) \leftarrow \{\, f \in F_j \mid t_j^{min}(f) \le u \,\}$
    $G_j(C_j(u)) \leftarrow \max(0,\ u - t_j^{done})$ ▷ 陳腐度削減量
    $E_j(C_j(u)) \leftarrow a_j + b_j \cdot \sum_{f \in C_j(u)} size(f)$ ▷ 推定実行時間
    $\eta_j(C_j(u)) \leftarrow \dfrac{G_j(C_j(u))}{E_j(C_j(u))}$ ▷ marginal-benefit
    if $\eta_j(C_j(u)) > \eta_j^*$ **then**
      $\eta_j^* \leftarrow \eta_j(C_j(u)),\quad C_j^* \leftarrow C_j(u)$ ▷ ジョブ内で最良候補を更新
    end if
  end for
  $L \leftarrow L \cup \{(j, \eta_j^*, C_j^*)\}$
end for
$D \leftarrow \emptyset$
$slots \leftarrow \max(0, K - |R|)$
$Lsorted \leftarrow L$ を η_j^* の降順にソート
for all $(j, \eta_j^*, C_j^*) \in Lsorted$ **do**
  if $slots = 0$ **then**
    break
  end if
  $D \leftarrow D \cup \{(j, C_j^*)\}$
  $slots \leftarrow slots - 1$
end for
return D

3.5 Apache Hudiを用いた実装

本研究の提案では、Hudi/Spark の内部処理方式を変更するものではなく、常駐スケジューラが (i) Hudi が出力するメタデータから評価に必要な情報を構成し、(ii) 選択した部分集合 C_j^* をジョブ起動用パラメータとして Spark ジョブに渡すものである。本節では、本研究の実装で用いる入出力と設定項目を整理する。

3.5.1 スケジューラが取得するメタデータ項目

各スケジューリング時点において、スケジューラは Hudi テーブルのメタデータを参照し、各ジョブ j の増分データを含むファイル集合 F_j を構成する。本実装では、Hudi の各テーブルに対して `.hoodie` 配下に記録されるコミットメタデータ（`*.commit` ファイル中のデータ）から次の情報を取得する。

- 増分データを含むファイル集合 F_j：

  `.hoodie/*.commit` に記録された `partitionToWriteStats` に含まれる各 `WriteStat` が指すデータファイルの集合として定義する。

- 各ファイル $f \in F_j$ の $(path(f),\ size(f))$：

  同一 `WriteStat` のフィールド `path` および `fileSizeInBytes` から取得する。

また，各ファイル f に対する到着時刻の最小値･最大値 $t_j^{min}(f), t_j^{max}(f)$ は，Parquet ファイルに格納された到着時刻カラムの統計情報（min/max）から取得する。ここで参照する統計情報はファイル末尾のフッターに格納されており，本実装では実データ走査は不要となる。

一方，反映済み状態 t_j^{done} はスケジューラが保持する状態として管理する。提案手法では，各周期で選択した候補上限時刻 u までを反映するものとみなし，ジョブ完了のたびに $t_j^{done} \leftarrow u$ を更新する。

投入可能ジョブ集合 J の判定もスケジューラが行う。スケジューラは常駐プロセスとして `spark-submit` によりジョブを起動し、その子プロセスの終了を待つことで完了を検知するため、実行中か否かを内部状態として判別できる。各周期では、$F_j \neq \emptyset$ であり、かつ実行中でないジョブ j を投入可能として J を構成する。

3.5.2 Sparkアプリケーションの起動と入力パラメータ

スケジューラは、Algorithm 1 により選択された部分集合 C_j^* に含まれるファイルのパス一覧を、ジョブ起動時の入力パラメータとして渡すことで、実行時に処理対象ファイルを限定する。具体的には、スケジューラが `spark-submit` により各 ELT ジョブ（Apache Spark アプリケーション）を起動し、引数としてジョ

ブ識別子と入力ファイルパス一覧（C_j^*）を渡す。ジョブ側では引数で受け取ったパス一覧を用いて `spark.read.parquet(paths)` を実行し、当該ファイルのみを DataFrame として読み込むことで読取り対象を限定する。その後、ジョブに応じて Spark SQL による変換処理（集計・結合など）を行い、変換結果を Hudi テーブルへ書き込んでコミットする。このコミットにより `.hoodie` 配下のメタデータが更新され、次周期のスケジューラが参照する入力情報（3.5.1 節の `WriteStat` 等）となる。

3.6 期待する効果

本手法の狙いは、増分データが多数のファイルに分散する状況では、全量処理を前提にすると処理コストが増えやすく、marginal-benefit が下がる場合がある点に対して、処理対象をファイル集合の部分集合として柔軟に選べるようにすることで、marginal-benefit を高めることである。

具体的には，各ジョブ j について候補時刻 u に基づくファイル集合の中から、marginal-benefit が最大となる処理対象集合 C_j^* を選択できるため、処理コストに対して得られる G_j が小さいファイルを一時的に外し、少ない処理コストで大きく陳腐度を削減できる範囲に計算資源を集中させやすくなる。このとき、陳腐度の削減量そのものは全量処理より小さくなる場合があるが、処理コストの削減がそれ以上に効けば、結果として marginal-benefit が増加する。

また、ジョブ間の選択は各ジョブで最大化された代表値 η_j^* に基づくため、局所的に効率のよい実行が増えるだけでなく、パイプライン全体としても陳腐度を効率よく減らせるジョブから先に回す方向に実行順序が誘導される。その結果として、実効 marginal-benefit の分布が改善し、レイクハウス全体のデータの陳腐度 P の低減に寄与する可能性がある。また、部分集合選択は差分を捨てる操作ではなく、取り込み効率の低い部分を先送りすることで、後続の更新到着によって同ファイルを処理することによる陳腐度の削減量が多くなった時点でより効率的に処理される可能性がある。

3.7 まとめ

本章では、常駐スケジューラがオンラインにスケジューリングし、ELT ジョブを繰り返し起動する枠組みを前提として、ジョブ評価を増分データを含むファイル集合の部分集合へ拡張する Partitioned-Max-Benefit を提案した。到着時刻の最大値集合 U の各要素 u に対して集合 $C_j(u)$ を評価することで、反映済み時刻の整合性を保ちつつ、各スケジューリング時点で $marginal\text{-}benefit$ を最大化する処理対象 C_j^* を選択できる。次章では、提案手法の有効性を検証するための実験環境と評価方法、およびその結果を述べる。

第4章　実験・評価

4.1　はじめに

本章では、提案手法である Partitioned-Max-Benefit スケジューリングの有効性を評価するために行った実験とその結果を述べる。まず、実験環境について述べた上で、提案手法自身の評価と既存手法との比較を行う。

4.2　実験条件

本研究では、証券会社におけるシステムを想定し、Apache Hudi を用いたレイクハウス上に実装した ELT パイプラインを用いて行った。実験環境におけるストリームデータの流れと各コンポーネントの関係を図. 4.1 に示す。図に示す JOB1〜6 は、それぞれ矢印の向きに従って増分データを反映する ELT ジョブである。以降、簡単のため各ジョブは更新先テーブル名のアンダースコア（_）をハイフン（-）に置き換えた名称で表す（例：テーブル名 trade_aggr、ジョブ名 trade-aggr）。

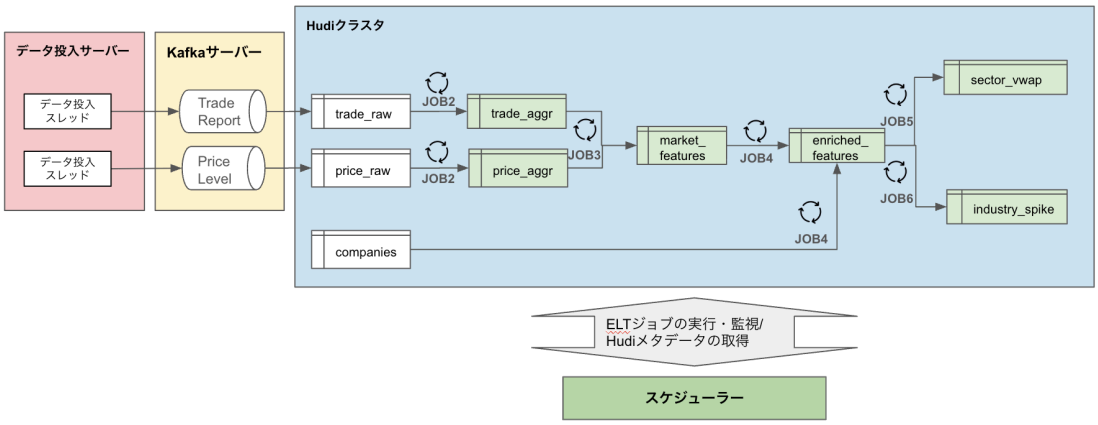

図 4.1: システム構成

本実験では、trade_aggr、price_aggr、market_features、enriched_features（以降、上流のテーブル）は高カーディナリティ属性である銘柄の頭文字に基づくパーティショニングを前提としており、更新が多数のファイルに分散しやすい条件となる。一方、industry_spike、sector_vwap（以降、下流のテーブル）はパーティショニン

グを行わない設計としたため、コミットごとに更新されるファイル数が小さくなりやすい。このように、増分データが多数のファイルに分散して保存される場合と、そうでない場合の両方を同一パイプラインに含めることで、提案手法がどの条件で部分集合選択として現れやすいかを観測できるようにした。

実験データには、米国株式市場 IEX における板情報（Price-Level）および約定情報（Trade-Report）に関するマーケットデータ [14] と、企業情報の参照のため Yahoo Finance のマーケットデータ [15] を用いた。IEX のマーケットデータはストリームとして約 3 時間で生データ合計約 18GB の約 1.6 億レコードを投入した。Yahoo Finance のデータは合計約 10MB、5257 レコードをマスタデータとしてあらかじめレイクハウスに保存した。各 ELT ジョブの 1 回あたりの実行時間は概ね 2 分〜6 分程度であり、ジョブの入力データサイズは最大でも約 2.5GB 程度であった。

実験環境および主要なソフトウェア構成・設定値を表 4.1 にまとめる。

表 4.1: 実験環境と主要設定

項目	設定値
Apache Hudi バージョン	`1.0.0`
Apache Spark バージョン	`3.4.2`
Hadoop/HDFS バージョン	`3.3.6`
Kafka バージョン	`3.6.0`
Spark executor 設定	`2GB/8executor`
Driver memory	`2GB`
並列度（同時実行数）	3
Hudi テーブル種別	`CoW`

また、実験環境として、以下の 4 要素から構成されるデータパイプラインを構築し、約 3 時間にわたってストリームデータの生成およびスケジューラの実行を行い、各種指標の計測を行った。

(1) データ投入

1 台の Linux マシン上に Apache Kafka を構築し、そのキューに別の Linux マシン 1 台から実験用のストリームデータを投入するプログラムを実装した。Apache Hudi はこのキューからストリームデータをポーリングで取得する。キューへのデータ投入は、実際のマーケットデータにおけるイベント時刻の間隔に基づいて行った。各レコードの到着時刻はキュー投入時に付与している。

(2)Apache Hudi クラスタ

9 台の Linux マシンで構成されるクラスタ環境を用いた。各ノードには Apache Hudi、Apache Spark、Apache Hadoop をインストールした。ELT ジョブは本ク

ラスタ上で実行され、Apache Hudi のライブラリを介して Apache Spark ジョブが起動される。Apache Hudi により管理される業務データは、Apache Hadoop の HDFS 上に保存される。

(3) ELT ジョブ

本実験環境は6つの ELT ジョブから構成され、price_raw テーブルおよび trade_raw テーブルから下流テーブルへ段階的にデータを変換し、反映する。各ジョブの集計内容は次のとおりである。

- **trade_aggr:** trade_raw を銘柄単位に 1 秒粒度で集計し、始値・終値・高値・安値・出来高などを算出する。
- **price_aggr:** price_raw を銘柄単位に 1 秒粒度で集計し、最良 Bid/Ask 価格、Bid/Ask 数量などを算出する。
- **market_features:** price_agg と trade_agg を（時刻、銘柄）で結合し、スプレッド等の市場特徴量を算出する。
- **enriched_features:** market_features に企業マスタを銘柄で結合し、業種、業界カラム等を付与する。
- **sector_vwap:** enriched_features を業種単位に集計し、業種別の取引状況を算出する。
- **industry_spike:** enriched_features を産業単位に集計し、出来高等の急増指標を算出する。

また、本実験では ELT ジョブは、Apache Spark 上で 1 回起動され、Hudi の増分読取り・SQL 変換・Hudi への書き出しまでを行う処理単位である。price_raw テーブルから後続テーブルへの処理には、Hudi の増分クエリを用い、前回のジョブ実行以降に追加または更新されたレコードのみを抽出した。ジョブ内のデータ変換ロジックは、集計、結合、ウィンドウ関数などの SQL を用いて実装している。

(4) スケジューラー

提案手法である Partitioned-Max-Benefit とその比較対象は、このスケジューラー内で動作する。スケジューラーは各手法の必要性に応じたメタデータの取得、実行するジョブと実行対象ファイルの選択、Apache Hudi クラスタで ELT ジョブの起動を起動する。marginal-benefit を算出するための推定実行コストを算出する際に用いる係数 a_j, b_j は、本計測に先立って前日のストリームデータを処理した際の合計入力ファイルサイズと実行時間の実績に対し線形回帰を行い推定した。以

降の評価では、この a_j, b_j を用いて第 3 章で定義した処理コスト近似を計算する。Apache Hudi クラスタ上ではスケジューリング対象とする 6 つのジョブのうち、同時に 3 つの ELT ジョブが並列で実行できるものとした。

4.3 提案手法の評価

本節では、提案手法 Partitioned-Max-Benefit の有効性を、問題提起（第 2 章）と提案内容（第 3 章）に対応づけて区分しながら評価する。具体的には、(1) 増分データのファイル分散現象の確認（4.3.1 節）、(2) Algorithm 1 の手順追従確認（4.3.2 節）、(3) 部分集合選択の発生頻度と規模、および既存手法との差（4.3.3 節以降）を順に示す。特に本研究で課題としている、増分データが多数のファイルに分散し、全量処理では処理コストが増大しやすい状況において、提案手法が既存手法より高い marginal-benefit を達成し、データの陳腐度を削減することを確認する。

4.3.1 増分データのファイル分散現象

まず、第 2 章で述べた、増分データが多数のファイルに分散して保存される現象を確認する。各テーブルへの書き込みごとに更新されたファイル数を集計し、その統計量を表 4.2 に示した。

表 4.2: コミットごとの更新ファイル数

テーブル	コミット回数	平均	最小	最大
market_features	25	26.0	26	26
price_aggr	43	26.0	25	26
trade_aggr	28	24.7	14	26
enriched_features	12	23.3	3	26
industry_spike	11	1.0	1	1
sector_vwap	11	1.0	1	1

上流テーブル（trade_aggr、price_aggr、market_features、enriched_features）では、1 コミットあたりの更新が概ね 20〜26 の多数ファイルにまたがっている。一方、下流テーブル（industry_spike、sector_vwap）では更新ファイル数は常に 1 である。これは、上流では更新対象の銘柄が多岐にわたり銘柄頭文字パーティションを横断して更新が発生するのに対し、下流は非パーティション設計のため更新先が単一ファイルに集約されやすいためである。このように、本実験には更新が多数ファイルに分散する場合と更新が少数ファイルに集約する場合の両方が含まれていることが確認できる。

次節以降では、このような実験環境において提案手法が入力として用いる増分データを含むファイル集合の構成と、候補時刻 u に対して定まる処理対象集合 $C_j(u)$ の大小が推定実行時間 E と陳腐度削減量 G に与える影響、ならびにそれらが性能指標 P に反映される様子を、集計結果の具体例で示す。

4.3.2 提案手法の追従確認

本節では、第3章で示したPartitioned-Max-Benefitの Algorithm 1の手順に沿って、実験ログから得られたスケジューラの内部状態が想定どおりに推移しているかを確認する。具体例として、trade-aggr（図4.1における JOB2）で処理対象ファイルの選別が発生した、ある1周期の内部状態を取り上げている。

1. **入力の取得**

   当該周期でスケジューラは各ジョブ j の入力テーブルに対し、未処理の増分データを含むファイル集合 F_j が列挙された。具体例として、trade-aggrが入力データを取得するtrade_rawにおいては、$|F_j| = 13$ であり、前回までに反映済みの最新到着時刻は $t_j^{done} = 22{:}34{:}38$ であった。

2. **候補時刻の列挙と集合 $C_j(u)$ の構成**

   ファイル集合 F_j に含まれる各ファイル f について最新到着時刻 $t_j^{max}(f)$ を取得し、それらを昇順にソートして候補時刻の列 $U = \{u_1, \ldots, u_m\}$ を得る。各候補時刻 u に対して，u までの未反映増分を含むファイルを $t_j^{min}(f) \le u$ を満たすものとして選別し、処理対象集合 $C_j(u) = \{\, f \in F_j \mid t_j^{min}(f) \le u \,\}$ を構成した。本周期のtrade-aggrの結果について表4.3に示した。

3. **候補集合の評価と最適集合の選択**

   表4.3より、本周期のtrade-aggrでは，全量に相当する候補（表中の $u = u_{14}$）において $G_j = 267$ 秒を得られる一方、スキャン量は249.26 MiB，推定実行時間は483.73秒であり、$\eta_j = 0.552$ であった。これに対して $u = u_8$ では、陳腐度削減量は $G_j = 117$ 秒に減少するが、スキャン量を7.28 MiBに抑えられ、推定実行時間192.40秒の結果、$\eta_j = 0.608$ となった。以上より，本周期では $\eta_j(C_j(u))$ を最大化する候補時刻を $u = u_8$ とし、$C_j^* = C_j(u_8)$ を選択した。このとき、最大の候補時刻（$u = u_{14}$）までの全量ファイルを対象とする場合と比べてスキャン量は249.26から7.28 MiBに減少（約97.1%削減）し、marginal-benefitは0.552から0.608に増加（約10.2%増加）した。これは，未反映増分の取り込み範囲（候補時刻 u）を調整し，単位処理コスト当たりの陳腐度削減効率を高めるという提案手法の意図した挙動が実際に発生していることを示している。

表 4.3: trade-aggr における候補時刻 u ごとの選別結果と評価値

index	u	スキャン量 [MiB]	推定実行時間 [秒]	G_j[秒]	$\eta = G/E$
1	22:35:00	1.06	184.90	22	0.119
2	22:35:16	1.98	186.02	38	0.204
3	22:35:30	2.84	187.05	52	0.278
4	22:35:43	3.79	188.19	65	0.345
5	22:35:56	4.62	189.18	78	0.412
6	22:36:12	5.54	190.30	94	0.494
7	22:36:23	6.30	191.21	105	0.549
8	22:36:35	7.28	192.40	117	0.608
9	22:36:50	75.13	274.44	132	0.481
10	22:37:16	136.30	347.91	158	0.454
11	22:37:49	179.18	399.35	191	0.478
12	22:38:21	208.21	434.18	223	0.514
13	22:38:42	249.26	483.73	244	0.505
14	22:42:22	249.26	483.73	267	0.552

4. **ジョブ間の優先度付けとジョブ投入**

   同一周期において、各ジョブ j について候補集合を評価し、marginal-benefit η_j が最大となる処理対象 C_j^* とその値 $\eta_j(C_j^*)$ を得た。表 4.4 に、各ジョブに関する処理結果を示す。本周期では marginal-benefit の値は trade-aggr の $\eta_j(C_j^*) = 0.608$ が最も大きく、次点は enriched-features の 0.579 であった。market-features, sector-vwap, industry-spike については増分データが存在せず、投入可能ジョブには含まれていなかった。Algorithm 1 に従い、この値に基づいて trade-aggr が選択され、表の C_j^*（読取りファイル数 8、スキャン量 7.28MiB、陳腐度削減量 117、推定実行時間 192.40 秒）を入力としてジョブが実行されている。

表 4.4: ジョブ間比較の具体例

ジョブ	読取りファイル数	$\lvert F_j \rvert$	スキャン量 [MiB]	$\eta_j(C_j^*)$
trade-aggr	8	13	7.28	0.608
price-aggr	21	21	3522.42	0.564
market-features	–	0	–	–
enriched-features	3	3	529.76	0.579
sector-vwap	–	0	–	–
industry-spike	–	0	–	–

以上の通り、測定期間における、あるスケジューリング周期を例とし、提案手法

の各手順が想定通り動作していることを示した。

4.3.3 P 削減への寄与

本研究の性能指標 P は、各テーブル (raw テーブルを除く) の陳腐度 $S_i(\tau)$ の時間積分である。したがって、あるジョブ完了による陳腐度削減量が Δs であるなら、その完了が全量選択の場合より前倒しされることで、少なくともその期間においては更新先のテーブルにおいて陳腐度が低い状態が長く維持され，P が減少する。

具体例として，trade-aggr の代表的な 1 周期では、全量選択の推定実行時間 483.7 秒 に対し、部分選択の推定実行時間は 192.4 秒であり、完了時刻が約 291 秒前倒しされる。このとき陳腐度削減量は、全量選択が $G_{\text{full}} = 267$ 秒，部分集合が $G_{\text{sub}} = 117$ 秒 であった。

ここで、t_{sub} から t_{full} までの時間帯では，全量処理はまだ完了していないため、

部分集合選択は少なくとも G_{sub} だけ陳腐度を低下させた状態を Δt 秒だけ早く実現する。従ってこの区間における P の差分は

$$\Delta P \approx G_{\text{sub}} \cdot \Delta t \approx 117 \times 291 \simeq 3.4 \times 10^4$$

となり、部分集合選択が有利となる。

一方で，全量選択は完了時点で部分選択よりも陳腐度をさらに $(G_{\text{full}} - G_{\text{sub}}) = 150$ 秒だけ下げられる。したがって、全量処理の完了後にその差（150 秒）が維持される時間が十分に長い場合には、全量選択のほうが P を小さくし得る。この条件を表す閾値時間は、

$$\frac{G_{\text{sub}}\Delta t}{G_{\text{full}} - G_{\text{sub}}} \simeq \frac{3.4 \times 10^4}{150} \approx 2.3 \times 10^2 \text{ s}$$

である。すなわち、全量処理が完了する時刻から約 227 秒（約 3.8 分）以内に次のジョブが開始されるような状況では、部分集合選択の方が P を小さくすることができる。

また、実験ログ上、trade-aggr の部分選択が完了した直後に、別ジョブ（industry-spike）が開始されていることが確認できる。trade-aggr を全量処理した場では、部分処理と比較して industry-spike の開始は少なくとも約 291 秒遅延し得るため、当該 industry-spike 実行での陳腐度削減量は 1163 秒であったことから、この遅延だけでも全量選択では P に対しおおよそ $1163 \times 291 \approx 3.4 \times 10^5$ 相当の P の増加を発生させうる。

ただし、全量選択にすると他ジョブの投入順や並列実行の組合せも変わり得るため、この 291 秒がそのまま後続ジョブの開始遅延に対応するとまでは言い切れない。

4.3.4 部分選択の発生頻度と規模

前節までは、単一周期における部分集合選択が局所的に marginal-benefit を高めうることを例示した。本節では、そのような部分集合選択が評価期間全体でどの程度の頻度で生じ、どの程度の規模で入力削減を伴ったのかを整理する。

提案手法が増分データを含むファイルの一部のみが処理されたケース（以降、部分選択）は price-aggr と trade-aggr において観測され、全ジョブ実行回数 89 回のうち 6 回であった。表 4.5 は、ジョブごとの発生回数と、除外されたファイル数の規模、および選択時の marginal-benefit の代表値を示す。

表 4.5: 発生した部分集合選択の統計量

ジョブ	実行回数	部分集合回数	選択 MB (中央値)	全量 Scan 平均 [MiB]	選択 Scan 平均 [MiB]
price-aggr	24	3	1.095	1243.4	287.8
trade-aggr	23	3	1.091	301.1	15.2

部分集合選択が発生した各周期について、同一判断時点で全量選択をした場合と比較すると、全ての周期で、スキャン量を削減しつつ陳腐度削減量 G の低下を相対的に小さく抑えることで、marginal-benefit が全量選択より大きくなることを確認できた。例えば price-aggr の例では、増分データを含む 19 ファイルのうち 4 ファイルが選択され、スキャン量が 1497MiB から 13MiB へ大幅に縮小した一方で、marginal-benefit は 0.668 から 0.929 に増加した実行が観測された。

一方，残りの 4 ジョブでは部分集合選択が発生しなかった．これらのジョブでは，到着時刻の候補 u に対して定まる処理対象集合 $C_j(u)$ を小さくしても推定実行時間 E の低下が小さい一方で、陳腐度削減量 G の低下が相対的に大きく $\eta = G/E$ が全量選択で最大化されやすかった。したがって、本実験条件では，$C_j(u)$ の縮小による推定実行時間 E の削減量が小さい（または不利）一方で陳腐度削減量 G の損失が大きく，η は全量選択で最大化されやすかったと解釈できる．

4.4 既存手法との比較

本節では、既存手法である Max-Benefit、ランダムにジョブを選択する手法（Random）、および提案手法である Partitioned-Max-Benefit の 3 手法を比較し、(i) 各ジョブの実効 marginal-benefit の分布、(ii) データの陳腐度の 2 点から性能を評価する。

4.4.1 marginal-benefit の評価

図 4.2 は、各 ELT ジョブが実行された際の実効 marginal-benefit の分布を、スケジューリング手法ごとに箱ひげ図で比較したものである。実効 marginal-benefit は、各 ELT ジョブの各回の実行について、スケジューラーが算出した陳腐度改善量を当該実行の実測実行時間で除して算出した。陳腐度改善量と実測実行時間はいずれも実験ログから取得した。

箱ひげ図の箱の横線は実効 marginal-benefit の中央値を表している。また、箱の下端、上端は第一四分位点、第三四分位点を示している。また、縦軸は marginal-benefit、横軸は各テーブルを更新する ELT ジョブを表す。

図を見ると、trade-aggr および price-aggr では、3 手法の中央値、四分位点は概ね近く、提案手法が常に最大となるわけではない。market-features、enriched-features、sector-vwap では、Partitioned-Max-Benefit の中央値が Random および Max-Benefit より高く、箱の上下幅も狭いため、提案手法では実効 marginal-benefit が高い値で安定している傾向が確認できる。ただし、industry-spike では Max-Benefit の中央値が最も高く、提案手法が常に最大となるわけではない。

なお、前節の結果より、部分集合選択は price-aggr と trade-aggr でのみ発生しており（表 4.5）、それ以外のジョブでは部分集合選択は発生していない。したがって、これらのジョブにおける実効 marginal-benefit の分布改善は、部分集合選択が直接行われたことによって生じたものではない。

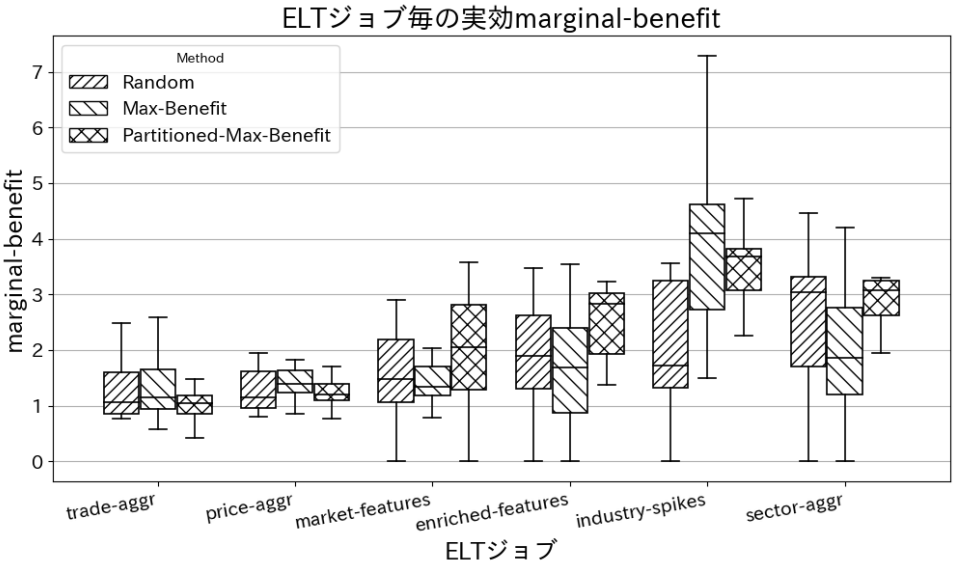

図 4.2: ELT ジョブごとの実効 marginal-benefit の分布

4.4.2 データの陳腐度の評価

図 4.3 は、各スケジューリング手法におけるレイクハウス全体のデータの陳腐度の推移を示す。縦軸は、時刻 τ における raw を除く更新先テーブル群の陳腐度の合計 $\sum_i S_i(\tau)$ を表す。曲線がより低い水準で推移する手法ほど、評価期間を通じて低い陳腐度を維持していることを示している。

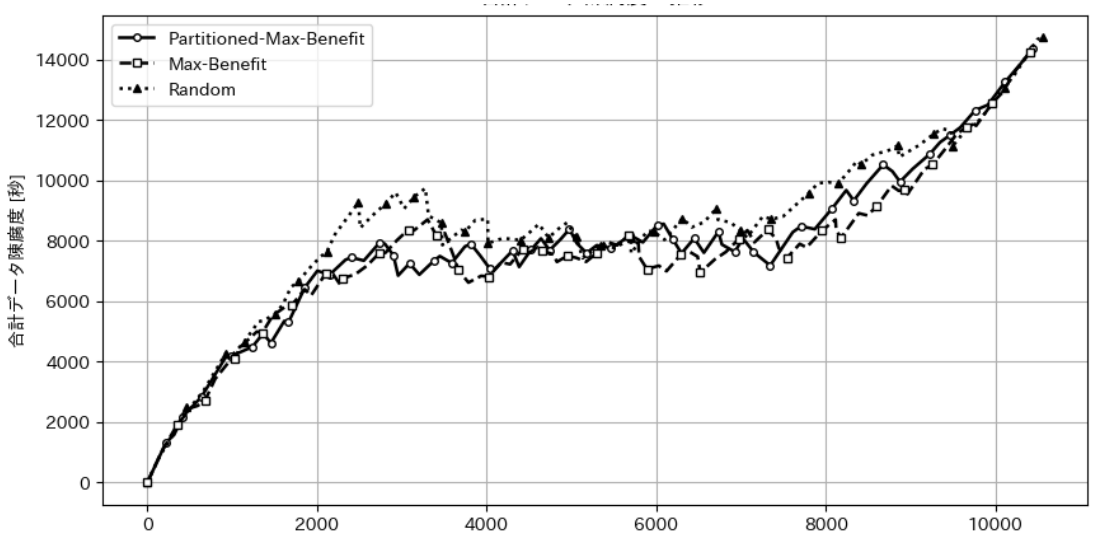

図 4.3: レイクハウス全体のデータの陳腐度の推移

Partitioned-Max-Benefit は全期間を通じて、Random より低い水準で推移しており、システム全体として陳腐度を抑えられていることが分かる。また Max-Benefit と比較しても、同程度かそれ以下の水準で推移する区間が多く、本実験設定では提案手法が全体の陳腐度を小さく保つ方向に働いたといえる。

次に、第 2 章で定義した性能指標 P について、計測開始から計測終了までの約 3 時間を評価期間として算出した結果を表 4.6 に示す。

表 4.6: レイクハウス全体の性能指標 P

手法	P の近似値
Partitioned-Max-Benefit	73,875,680
Max-Benefit	77,336,752
Random	85,474,258

性能指標 P について、評価では連続的な値をそのまま積分できないため、1 秒ごとに raw を除く各テーブルの t_i^{latest}（更新先テーブルに反映済みの最新到着時刻）をログから取得し、$S_i(\tau) = \tau - t_i^{latest}$ を計算して $\sum_i S_i(\tau)$ を得たうえで、その値を足し合わせることで P を算出した。Partitioned-Max-Benefit は Random に対して約 13.6%、Max-Benefit に対して約 4.5%、性能指標 P を削減した。

4.5 考察

図 4.2 で見られた提案手法における実効 marginal-benefit の相対的に高い分布は、各 ELT ジョブ実行において単位処理コストあたりの陳腐度削減効果が高い実行が多く観測されたことを示唆しており、表 4.6 において性能指標 P が既存手法と比較して削減できたことと整合している。

部分集合選択が発生した局面では、marginal-benefit を増加させつつ、陳腐度削減量の低下を小さく保てることが確認できた。したがって、このような局所的な効率改善が性能指標 P の低減に一定寄与していると言える。

一方で、P は評価期間全体にわたる陳腐度の累積量であり、部分集合選択の有無以外にも、各周期の評価値に基づく実行順序の変化など複数の要因が作用する。そのため、本実験の観測結果のみから、部分集合選択が P の低減に与えた寄与を定量的に示すことはできない。寄与の同定には、実行順序と同時実行数を固定した対照実験が必要である。

4.6 まとめ

本章では、Apache Hudi を用いたレイクハウス上に ELT パイプラインを構築し、約 3 時間のストリーム投入実験により提案手法 Partitioned-Max-Benefit を評価した。まず提案手法自身の評価として、増分データのファイル分散が生じる条件を確認した上で、Algorithm 1 の各手順が実験環境で追従して動作することを具体例により示した。そして、既存手法と比較では、実効 marginal-benefit の分布が、ジョブによっては提案手法が Max-Benefit や Random を上回る傾向が確認できた一方で、常に最大となるわけではないことも確認された（図 4.2）。また、レイクハウスの ELT ジョブが更新するテーブル群全体のデータの陳腐度は、提案手法が Random より低い水準で推移し、性能指標 P も 3 手法の中で最小となった（図 4.3, 表 4.6）。

第5章　おわりに

5.1　本研究の概要と成果

本研究では、Apache Hudi を用いたレイクハウス環境における ELT パイプラインを対象に、増分処理における読取り効率のばらつきがスケジューリング性能に与える影響に着目し、データの陳腐度の低減を目的としたスケジューリング手法の提案と評価を行った。特に、主キーの広範な分布により増分データが多数のファイルに分散し、増分クエリが多くのファイル読取りを要することで処理効率が低下しうる状況を想定した。

既存手法の Max-Benefit では、テーブル全体を不可分な処理単位として扱い、増分データを全て処理することを前提に marginal-benefit を評価する。しかしレイクハウスでは、ファイルごとに読取りに対して得られる増分データ量が異なり、この差が陳腐度の削減効率に影響する。そこで本研究では、テーブル単位ではなく増分データを含むファイル集合を処理単位として捉え、その部分集合に対する marginal-benefit を評価して処理対象を動的に選択するスケジューリング手法 Partitioned-Max-Benefit を提案した。提案手法は、増分データの取り込み範囲に応じた処理対象ファイル集合を複数候補として作り，marginal-benefit が最大となる処理範囲を選択した上で、ジョブ間ではその代表値に基づいて貪欲に実行優先度を決定する。これにより、増分が薄く読取り効率の低いファイルを一時的に処理対象から外し、限られた計算資源でも単位コスト当たりの陳腐度削減量が高まることを期待した。

実験では、Apache Hudi 上に構築した 6 ジョブからなる ELT パイプラインに対し、Max-Benefit、Random、Partitioned-Max-Benefit の 3 手法を適用して比較した。その結果、実効 marginal-benefit の分布は、market-features、enriched-features、sector-vwap では Partitioned-Max-Benefit の中央値が Random および Max-Benefit より高く、四分位範囲も狭い傾向が確認できた。一方で、trade-aggr と price-aggr では 3 手法の中央値・四分位範囲が概ね近く、また industry-spike では Max-Benefit の中央値が最も高いなど、提案手法が常に最大となるわけではない。また、性能指標 P（更新先テーブル群のデータの陳腐度の積分和）で評価すると、提案手法は Random および Max-Benefit と比較してレイクハウス全体の陳腐度を低減し、本設定において最も小さい P を達成した。さらに、部分集合選択が発生した局面では、全量処理と比較して入力スキャン量を大きく抑えつつ陳腐度の削減量の低下を小さく保ち、結果として marginal-benefit を高める挙動が観測された。

なお、部分集合選択の観測回数は限定的であるため、P の低減に対する寄与を

部分集合選択の直接効果として定量的に切り分けるには追加検証が必要である。また、部分集合選択が観測されたのは上流の price-aggr / trade-aggr に限られた一方で、下流ジョブでも実効 marginal-benefit の分布改善が観測されており、これは主に各周期の評価値に基づくジョブ投入順序の変化による影響を含むと考えられる。

以上より、本研究の成果は次の 3 点に整理できる。(1) レイクハウスにおける増分処理では、ファイル単位で読取り効率が異なるため、テーブル全量処理を前提とした marginal-benefit 評価には限界があることを整理した。(2) 増分データを含むファイル集合の部分集合に対して marginal-benefit を定義し、処理対象を動的に選択するスケジューリング手法 Partitioned-Max-Benefit を提案・実装した。(3) Apache Hudi を用いた実験により、提案手法が既存手法と比較して実効 marginal-benefit を高め、レイクハウスの ELT ジョブが更新するテーブル群におけるデータの陳腐度指標 P を低減しうることを本実験設定において確認した。

5.2 今後の課題

本実験のデータ規模、システムの規模は限定的であるが、実運用においても全てのワークロードが常に大規模とは限らず、更新頻度や対象範囲によっては相対的に小さい増分データを継続的に処理し続ける状況も存在する。したがって、本研究の設定は小規模な増分更新が継続する局面におけるデータ鮮度最適化という実運用上の一側面を捉えている点で一定の意義がある。一方で、データ量や同時実行ジョブ数が増加した場合には実行時間特性が変化し、スケジューリングの効果も異なる可能性がある。そのため、より大規模なデータ規模・高負荷条件においても、提案手法がデータの陳腐度を安定して低減できるかを追加で検証する必要がある。

また、提案手法は各周期での推定量に基づくヒューリスティックであり、局所的に高い marginal-benefit の選択が性能指標 $P = \sum_i \int S_i(\tau)\, d\tau$ の低減にどの程度一貫して結び付くかは自明ではない。本研究では実験により P の低減を確認したが、その効果が得られる条件（増分の分散度合い、ファイルサイズ分布、更新到着過程、依存関係の深さ等）を体系化できていない。今後は、部分集合選択の発生条件と P の改善量との対応を整理し、効果が得られやすい条件・得られにくい条件を明確化する必要がある。

さらに、本研究の実装は Apache Hudi を前提としており、増分クエリの入力となるファイル列挙やメタデータ取得方法は Hudi の管理方式に依存している。しかし、MVCC に基づくファイル管理と増分読取りを提供する点は他のレイクハウス実装である Delta Lake や Apache Iceberg とも共通している。今後は、それぞれの実装が提供するメタデータを用いて、増分データを含むファイル集合の構成や部分集合選択を同様に定義できるかを整理し、提案手法の適用範囲と移植に必要な前提条件の明確化が課題である。

参考文献

[1] Databricks, “Databricks launches Lakehouse for financial services to accelerate data-driven innovation across the industry,” Databricks Newsroom. (参照日: 2025-01-01). https://www.databricks.com/company/newsroom/press-releases/databricks-launches-lakehouse-for-financial-services-to-accelerate-data-driven-innovation-across-the-industry.

[2] Oracle Corp., “OCI finance lakehouse solution,” https://docs.oracle.com/en/solutions/oci-finance-lakehouse/plan-data-lakehouse1.html, (参照日: 2025-01-01).

[3] Amazon Web Services, “Amazon Simple Storage Service (S3),” https://aws.amazon.com/s3/, (参照日: 2025-01-01).

[4] Apache Software Foundation, “HDFS: Hadoop Distributed File System design,” https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, (参照日: 2025-01-01).

[5] Armbrust, Michael, et al., “Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics.” Proceedings of CIDR, Vol. 8, 2021.

[6] Codd, Edgar F., “A relational model of data for large shared data banks.” Communications of the ACM, 13(6), 377–387, 1970.

[7] Rucco, F., et al., “Formalizing ELTT and ELTL Design Patterns and Proposing Enhanced Variants: A Systematic Framework for Modern Data Engineering.” arXiv:2511.03393, 2025. [Online]. Available: https://arxiv.org/abs/2511.03393.

[8] Schneider, Jan, et al., “The Lakehouse: State of the Art on Concepts and Technologies.” SN Computer Science, 5(5), 1–39, 2024.

[9] Apache Software Foundation, “Apache Hudi File Sizing,” https://hudi.apache.org/docs/file_sizing/, (参照日: 2025-01-01).

[10] Golab, L., Johnson, T., and Shkapenyuk, V., “Scalable Scheduling of Updates in Streaming Data Warehouses,” IEEE Transactions on Knowledge and Data Engineering, Vol. 24, No. 6, pp. 1092–1105, June 2012. doi: 10.1109/TKDE.2011.45.

[11] Jain, Paras, et al., “Analyzing and Comparing Lakehouse Storage Systems.” CIDR, 2023.

[12] Apache Software Foundation, “Apache Spark,” `https://spark.apache.org/`, (参照日: 2025-01-01).

[13] Apache Software Foundation, “Apache Kafka,” `https://kafka.apache.org/`, (参照日: 2025-01-01).

[14] IEX Group Inc., “IEX Market Data,” `https://iextrading.com/trading/market-data/`, (参照日: 2025-01-01).

[15] Aroussi, Ran, “yfinance: Yahoo! Finance market data downloader,” `https://pypi.org/project/yfinance/`, (参照日: 2025-01-01).

研究業績

本研究に関連する研究業績はない

謝辞

本研究および本論文の執筆にあたり、ご指導・ご助言を賜りました主指導教員の井口先生に深く感謝申し上げます。副指導教員の田中先生、並びに審査員の方々にも、貴重なご指摘をいただきましたことを御礼申し上げます。また、超並列処理システム研究室の皆様には、発表や議論の機会を通じて有益な示唆をいただきました。ここに感謝いたします。最後に、研究活動を支えてくれた家族に感謝します。