時系列データベースのクエリ事前構文解析における効率化

西村　　卓†

† セコム株式会社 IS 研究所　〒 181–8528 東京都三鷹市下連雀 8–10–16 セコム SC センター
E-mail: †sugu-nishimura@secom.co.jp

あらまし　IoT が普及し、様々なデバイスが生成するデータを収集、分析することが当たり前になった。このようなデータの多くは時系列データであり、時系列データの蓄積・分析に特化した時系列データベース (TSDB) の需要が高まっている。時系列データは膨大なため、分析時のクエリに高速に反応するための近似クエリ処理 (AQP) やアクセス制御の処理などにおいて、事前にクエリを構文解析して変更処理を実施してから解釈する方法が研究され実践されてきている。本稿では、このような事前の構文解析と変更処理を、TSDB のクエリの特徴に適応させる手法を提案し、評価実験を行った。

キーワード　構文解析, 問合せ処理, 問合せ言語, 時系列データベース

1　はじめに

IoT [1] という言葉が登場してから既に 20 年以上が経過し、現在ありとあらゆるモノがインターネットで接続されていることが当たり前となった。この「モノ」がこの 20 年で急激に増加した結果、多種多様なデータを収集し活用できるようになった。収集されたデータの多くは時系列データであり、時系列データは「モノ」の数が多くなるほど増えるのは勿論、データを収集する頻度が高いと爆発的に増加する。このような時系列データを取り扱うために開発されたのが時系列データベース (Time-Series DataBase: TSDB) である。一般にデータベースと呼ばれるリレーショナルデータベースとは設計の根本から異なる、いわゆる NoSQL データベース [2] の 1 つであり、時系列データの特性に合わせた最適化が行われている。TSDB は IoT の普及と共に研究開発が進み、既に数十種類の TSDB がリリースされている。

TSDB の時系列データは非常に膨大であるため、分析時のクエリによって応答に非常に大きな時間がかかる。これらを高速化する手法の 1 つとして、近似クエリ処理 (Approximate Query Processing: AQP) [3] がある。これはクエリ処理に用いるデータポイントの数を減らすことで、近似値を計算する。例えば、1 ミリ秒の頻度で収集されたセンサデータの 1 年分のデータポイントの平均を計算するとき、そのデータポイントの数はおよそ 3.15×10^{10} 個にも上る。これを AQP として全てのデータポイントではなく 1 分毎のデータポイントだけを選択して平均を計算すれば、525,600 個のデータポイントで済む。

また、時系列データは様々なモノから収集した物であり、取り扱いの注意が必要なセンシティブな情報（例：個人の位置情報、行動履歴、生活パターンなど）も含む。このようなセンシティブなデータを必要とするサービス（例：高齢者の見守りサービスなど）があるが、ユーザはサービスを受けるのに必要最低限のデータだけをサービス提供者に提供できるのが望ましい。例えば室内熱中症のリスクを検知するようなサービスにおいては、全ての温度や湿度を詳細に提供する必要はなく、室温 30 度以上、湿度 70%以上になった時刻の情報だけを提供するだけで十分である。このようにサービス提供者はサービスに不要なデータにアクセスできないよう、TSDB のアクセス制御ができるとよい。

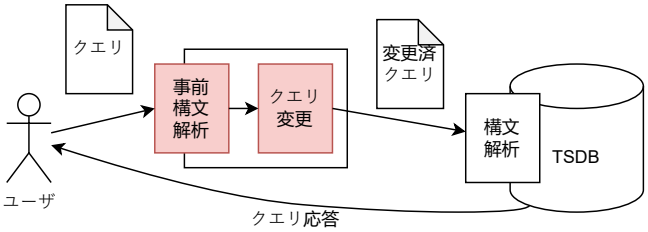

図 1　事前構文解析・クエリ変更のイメージ

これらの AQP やアクセス制御を実践する手法の 1 つとして、分析時のクエリを事前に解析し書き換え、その後 TSDB に変更済みのクエリで実行する方法が研究されている（図 1）[4] [5]。まず AQP について、先述の例のようにセンサデータの 1 年分のデータポイントの平均を計算するとき、全データポイントを選択する構文を一部のデータポイントを選択する構文に変更し、変更済みクエリを TSDB に渡す。次にアクセス制御について、サービスに不要なデータへアクセスするクエリ構文が発行されたとき、該当の構文を不要なデータにアクセスしないように変更し、変更済みクエリを TSDB に渡す。

本稿ではこのクエリの事前構文解析・変更処理において、TSDB のクエリの特徴を考慮することで、効率的に処理できる方法を 2 つ提案する。そしてそれらの提案手法の有効性を確かめるべく、評価実験を行ったため、これについても述べる。

以降、第 2 節では、時系列データ、TSDB についてより具体的に説明し、第 3 節では TSDB の一例である InfluxDB [6] について紹介する。第 4 節では、構文解析で用いられている形式文法、PEG(Parsing Expression Grammar) [7] について説明する。以上の第 2–4 節の説明を踏まえて、第 5 節で TSDB のクエリの事前構文解析・変更処理における効率化手法について 2 つ提案

する。次に第 6 節では提案した効率化手法の確認のため、評価実験を行う。最後に第 7 節で今後の課題を述べ、第 8 節でまとめる。

2 時系列データと時系列データベース

本節では、本稿の提案手法の前提知識として、まず時系列データと時系列データベース (TSDB) について、第 1 節よりも具体的に解説する。

2.1 時系列データ

時系列データとは、簡単に説明するとキーをタイムスタンプとする一連のデータ系列である。各タイムスタンプに対して、測定値及びタグ（測定場所や項目など）を持つ。時系列データは様々なモノから収集できる。最も有名な時系列データの 1 つとして、第 1 節でも紹介した IoT 環境でセンサやデバイス等から収集したデータが挙げられる。例えばスマートホームにおいて、各部屋における温度、湿度、電化製品の状態、電力使用量、カメラ映像、鍵の状態などのデータが収集できる。また、サーバやインフラの状況監視においても、多くの時系列データが収集される。他にも、株価の変動や銀行口座の履歴、天気予報、チャット、ログなども時系列データであるといえる。

次に時系列データの特徴について、書き込みと読み込みの視点で説明する。

2.1.1 書き込みの特徴

多くの時系列データ、特に IoT 環境で生成されるものは一定の間隔で連続的に生成される。一方でサーバやインフラのアラート、銀行口座の履歴、チャットと行った不定期に生成される時系列データもある。読み込みよりも遥かに書き込む頻度が高く、データの操作の 95%–99.9%が書き込みであり、生成間隔が短いほど書き込みの比重が高くなり、時系列データも非常に大きくなる。書き込みは最新のデータポイントを最後尾に追加するのみで、途中への挿入や更新、削除などは原則行わない。ただし手動でデータを修正したり、古くなったデータを削除したりする場合がある。

2.1.2 読み込みの特徴

タイムスタンプ及びタグを指定して時系列データを取得する一般的である。タイムスタンプの指定は大きく分けて 2 つあり、一方は最新タイムスタンプ（末尾のデータポイント）、もう一方は開始タイムスタンプと終了タイムスタンプを指定した時間範囲指定であり、過去 1 週間と言った指定もこれに含まれる。ある 1 つのタイムスタンプを直接指定して 1 つのデータポイントを取得することはまず行わない。以上で述べたように、最新のタイムスタンプの指定、最新からあるタイムスタンプへの時間範囲指定が多いため、新しいデータほど読み込む機会が多く、古いデータの重要性は低くなる。

2.2 時系列データベース

一般にデータベースと呼ばれるリレーショナルデータベース (RDB) [8] は表現能力が高く、あらゆるデータ形式を扱える万能なデータベースである。しかし RDB は第 2.1 節で述べたような特徴を持つ時系列データを取り扱うにあたっては様々な欠点がある。特に、時系列データの操作は書き込みに大きく偏っているが、RDB は書き込みより読み込みが多いデータの蓄積に向いており、また、時系列データの読み込むクエリで一般的な時間範囲指定を SQL で記述するのは直観的ではない。そのため時系列データの特徴に合わせた時系列データベース (TSDB) が開発された。

このように TSDB は RDB ではない、つまりクエリ言語として RDB で一般的な SQL を使わないデータベースであり、NoSQL データベースと呼ばれるデータベースの 1 種である。TSDB は Memcached [9] や redis [10] といったキーと値の組を蓄積するのに特化したキーバリュー型データベースや Cassandra [11] や BigTable [12] といった巨大なテーブルを扱うのに特化したカラム型データベースから派生して開発されたものが多い。これらのデータベースはキーに対して 1 つ（キーバリュー型）もしくは複数（カラム型）の値を保持するデータ形式のため、時系列データの格納に適しているためである。

ストレージのデータ構造には LSM ツリー (Log-Structured Merged-Tree) [13] を用いており、RDB で一般的な B ツリー [14] 系統とは大きく異なる。このデータ構造は、書き込みに定数時間、読み込みに線形時間を必要とする。書き込みも読み込みも対数時間である B ツリーと比べると書き込み速度の点で優れており、また、読み込み速度は劣るものの新しいデータほど早くアクセスできる特徴を持つため、書き込み重視で新しいデータほど読み込む時系列データの特徴と合致している。時系列データは際限なく収集するため、いずれストレージが足らなくなってしまう。そのため重要性の低い古いデータから自動的に削除できる機能をもつことが多い。さらに TSDB は時系列データの特徴に合わせた独自のクエリ言語が実装されており、特に時間範囲指定によるデータポイントの選択が容易に表現できるように設計されている。一方、書き込みは原則時系列データの収集でしか行わないため、SQL における INSERT、UPDATE、DELETE に相当するクエリは大きく機能が限定されているか、全く実装されていない。

IoT 普及と共に TSDB の需要が高まり、2010 年代から多くの TSDB が開発・リリースされた。これらの TSDB はそれぞれユースケースに応じてさらに特化されている。例えば巨大な行列演算を得意とし豊富な集計クエリ表現を持つことで、銀行や証券会社などで採用されている kdb+ [15]、値の型やサイズは固定、クエリの種類も制限し更新も削除も一切実装しないという大きな制約が課される一方で圧倒的なパフォーマンスを持つネットワーク監視システム専用の Confluo [16] などがある。各テック企業が自社サービスのために特化した TSDB を自ら開発しているケースも多い（図 1）。

3 InfluxDB

本節では TSDB の 1 つである InfluxDB [6] を紹介する。InfluxDB は本稿の評価実験でも用いられており、本稿で提案する効率化手法とも関係している。

表 1　自社サービスのために開発された TSDB

名称	デベロッパー	リリース年
LittleTable [17]	Cisco	2008
Prometheus [18]	SoundCloud	2012
GridDB [19]	東芝	2013
Atlas [20]	Netflix	2014
Gorilla [21]	Facebook	2015
Heroic [22]	Spotify	2015
M3 [23]	Uber	2018

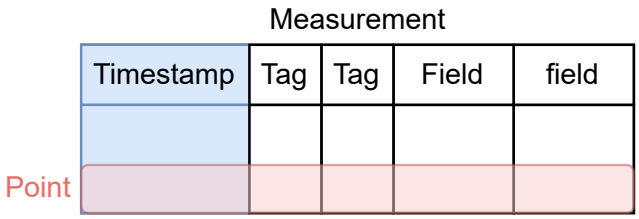

図 2　InfluxDB のデータ形式の概念図

表 2　実際のデータ形式の例

_time	_measurement	room	_field	_value
2021-11-20T00:00:00Z	sensors	kitchen	T	24.5
2021-11-20T00:00:00Z	sensors	kitchen	RH	67.8
2021-11-20T00:00:00Z	sensors	bedroom	T	21.6
2021-11-20T00:00:00Z	sensors	bedroom	RH	67.9
2021-11-20T00:10:00Z	sensors	kitchen	T	24.5
2021-11-20T00:10:00Z	sensors	kitchen	RH	67.7
2021-11-20T00:10:00Z	sensors	bedroom	T	21.7
2021-11-20T00:10:00Z	sensors	bedroom	RH	67.9
2021-11-20T00:20:00Z	sensors	kitchen	T	24.3
2021-11-20T00:20:00Z	sensors	kitchen	RH	67.7

InfluxDB は 2016 年に ver1.0 がリリースされた Influxdata 社が提供する最も人気の高い [24] オープンソース及び商用の TSDB である。第 2.2 節後半で説明したようなあるユースケースに特化した TSDB と異なり、様々なユースケースで利用できる万能型の TSDB である。商用版はオープンソース版と異なりクラスタ化によるパーティショニングやレプリケーション、部分バックアップができ、専門スタッフによるサポートを受けられるようになる。

データ形式について概念図を図 2 に示す。一連の時系列データをメジャメント (measurement) と呼び、タイムスタンプ (timestamp) は yyyy-MM-ddThh:mm:ss:nnnnnnZ の形式であり、ナノ秒まで取り扱うことができる。1 個以上のフィールド (field) は実際の測定値 (value) を保持し、0 個以上のタグ (tag) は測定場所や項目などを保持する。タグは省略可能だがこのタグを基準にインデックスを生成するため、クエリのパフォーマンスに影響する。RDB で例えるなら、メジャメントはテーブル、タイムスタンプ、タグ、フィールドはカラム、ポイントはレコードに相当する。実際のデータ形式は表 2 のようになる。アンダースコアから始まる _field などのカラムは、実際にクエリで指定できる。

InfluxDB はデータ構造に LSM ツリーを時系列データ用にさらに効率化させた独自の TSM ツリーを採用している [25]。第 2.2 節において TSDB の古いデータ自動削除する機能について述べたが、本来 LSM ツリーにおける大量の削除の効率はあまり良くない。TSM ツリーではその課題を解決し、タグを基準にしたインデックスを形成することで従来の LSM ツリー以上の効率化を達成している。

```
SELECT time, mean(T) AS temp
FROM "tsdb"."sensors"
WHERE (time > now() - 1d AND time < now() - 12h
       AND room = "bedroom")
GROUP BY time(1h)
FILL(0.0)
```

図 3　InfluxQL の例

3.1　クエリ言語

InfluxDB には 2 種類のクエリ言語が実装されており、初期リリースから利用可能な SQL ライクな InfluxQL と ver1.7 から利用可能になった関数型の Flux がある。ver2.0 からは Flux の利用が推奨されている。

図 3 に InfluxQL の例を、図 4 に Flux の例をそれぞれ示す。どちらも同じ内容のクエリであり、図 4 の Flux の例を 1 行ずつ解説すると以下のようになる：

- バケット（メジャメントのセット）は tsdb を指定
- 1 日から半日前の範囲
- メジャメントは sensors を指定
- タグキー room はタグ値 bedroom を指定
- フィールドキーは T を指定
- タイムスタンプとフィールド値以外の列を削除
- 1 時間毎に平均をとる
- 空のフィールド値を 0.0 で埋める
- フィールド値の列名を temp に変える
- クエリ結果を表示する

以上のように Flux は大きな時系列データの集合から、関数を繰り返し適用することで目的の系列データや値を取得していく様子が分かる。

4　構文解析と PEG

本節では PEG(Parsing Expression Grammar) [7] を用いた構文解析について説明する。PEG による構文解析は本稿の提案手法における事前構文解析に用いられている。

まず、構文解析とは、プログラミング言語などいった一定の規則（形式文法）によって生成される形式言語を解析することである。一般に構文解析においては、字句解析と呼ばれる事前処理が分離されている。字句解析は入力の形式言語を、トークンの定義を基にトークンに分割する処理である。例えば、トークンの定義（表 3）を基に”24 – 2 ∗ 7”という入力列に対して字句解析を実行すると、NUM”24”, SUB”–”, NUM”2”. MUL”∗”,

```
from(bucket: "tsdb")
 |> range(start: -1d, stop: -12h)
 |> filter(fn: r => r._measurement=="sensors")
 |> filter(fn: r => r.room == "bedroom")
 |> filter(fn: r => r._field == "T")
 |> keep(columns: ["_time", "_value"])
 |> aggregateWindow(every 1h, fn: mean)
 |> fill(column: "_value", value: 0.0)
 |> rename(colmuns: {_value: temp})
 |> yield()
```

図 4　Flux の 例

表 3　四則演算のトークンの定義

トークン	正規表現
NUM	[1-9]?[0-9]+
ADD	+
SUB	-
MUL	*
DIV	/
LP	(
RP	)

```
Expr  := Term ((ADD | SUB) Term)*
Term  := Value ((MUL | DIV) Value)*
Value := NUM | LP Expr RP
```

図 5　四則演算の構文解析規則（拡張 BNF 記法）

NUM”7”の 5 つのトークンに分割される。その後構文解析では、トークン列に対して規則（図 5）を繰り返し適用する。Expr, Term, Value は非終端記号と呼ばれ、規則の適用により 1 つ以上のトークンや非終端記号（規則の右辺）を 1 つの非終端記号（規則の左辺）へと置き換える。最終的に 1 つの非終端記号に到達したとき、解析は成功である。図 6 はトークン列から 1 つの非終端記号を導出する過程を表す。これを構文木と呼び、トークン列が葉ノード、最後の非終端記号が根である。この構文木によって計算結果 $24 - 2 * 7 = 10$ を得る。

字句解析器（レキサ）の実装は容易で、正規表現を用いて入力列の先頭から順にトークンを抽出すれば良い一方、構文解析器（パーサ）の複雑で実装は難しい。そのため BNF 記法などで記述された規則からパーサジェネレータを用いてパーサを自動生成するのが一般的である。パーサジェネレータの例として Yacc, Bison, JavaCC などが挙げられる。

字句解析を構文解析から分離する最大のメリットは、設計がよりシンプルになるため、処理が効率的である。しかし、字句解析により入力列をトークンに分割すると、本来持っていたトークンの意味を失ってしまうデメリットがある。近年、プログラミング言語は直観的に記述できるよう進化しており、特にこのデメリットが目立つようになり始めている。文字列補完は

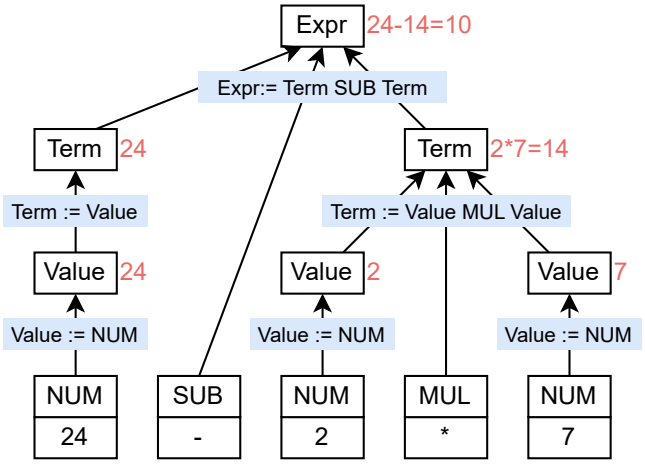

図 6　$24 - 2 * 7$ の構文解析

```
Expr  <- Term (("+" / "-") Term)*
Term  <- Value (("*" / "/") Value)*
Value <- [1-9]?[0-9]+ / "(" Expr ")"
```

図 7　四則演算の構文解析規則（PEG）

文字列の中に直接演算を埋め込むことができる処理であり、例えば Python において `f"1 + 2 = {1 + 2}"`のように記述する。この文字列を `print` などで出力すると `1 + 2 = 3` となる。2 つの `1 + 2` について、前者は文字列だが、後者は数式であり、字句解析で分割してしまうとどちらもただの数字”1”. 加算”+”, 数字”2”のトークン列となり区別ができなくなってしまう。

PEG による構文解析は、字句解析も一体化されており、先述のデメリットを解決できる。先述の四則演算の構文解析の例について、PEG による構文解析規則を図 7 に示す。表 3 のトークン規定が図 5 の構文解析に組み込まれたのが見て取れる。:=が<-に変わったのは単に拡張 BNF 記法と PEG の記法の違いで大きな差はないが、演算子 |（縦棒）と /（スラッシュ）は意味が異なる。$A|B$ は A か B のどちらかを選んで試行、失敗したらもう一方を試行するが、A/B は A から試行、失敗したら次の B を試行する。そのため PEG の記法における演算子 / は順番も重要で、曖昧性がない。この曖昧な表現がないのは PEG の特徴である。

5　事前構文解析における効率化

本節では、第 2–4 節で説明した内容を基に、事前構文解析の効率化手法を 2 つ提案する。この提案では、第 4 節で述べた PEG による構文解析を構文解析アルゴリズムとして採用する。

5.1　クエリの変更内容のメモ化

クエリ言語とプログラミング言語に対する構文解析を比較すると、その特徴は大きく異なる。まず、プログラムの長さに対してクエリは遥かに短く、特に TSDB におけるクエリは SQL クエリと比べて最新のデータの取得など、シンプルで非常に短

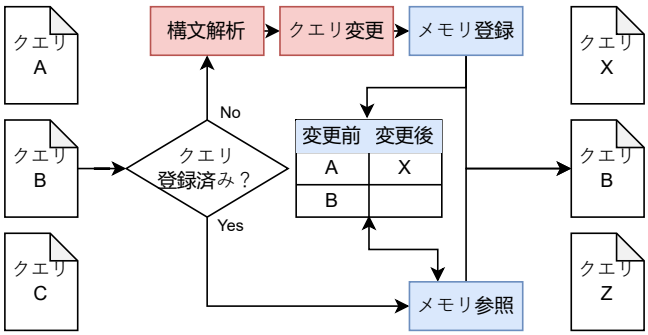

図 8　クエリの変更内容のメモ化

いクエリが多い。次に、プログラムに対する構文解析を実行する頻度は低く、プログラムのコンパイルまたは実行時だけであるが、クエリは発行される度に構文解析を行う。特に TSDB におけるクエリは頻度が高く、例えばダッシュボード上にグラフを描画するクエリは、最新のデータを取得するクエリを数秒単位で連続して発行している。最後に、TSDB におけるクエリは同じものが繰り返し利用される。先述のダッシュボード上にグラフ描画するクエリは同じクエリの繰り返しの発行であり、時系列データのサマリを毎日取得する場合も、毎日同じクエリを発行する。

事前構文解析後のクエリの変更処理において、あるクエリに対してクエリの変更結果は一意に決まる。つまり、変更前のクエリと変更後のクエリの組をメモリに保持することで、既に保持されたクエリが入力されたとき構文解析・クエリ変更処理を実行せず、そのまま対で保持している変更後のクエリを返すだけで良い。TSDB のクエリの長さは短いため、大きなメモリのコストにもなりにくく、頻繁に同じクエリが発行される特徴から、大きな効果があると予想される。このように処理結果を再利用するために保持し、再計算を防ぐことで高速化を実現する効率化手法をメモ化 (memoization) と呼ぶ。

この提案手法の概略図を図 8 に示す。クエリ A は既に登録されているため、計算は行わず、変更後のクエリ X を返す。クエリ C は登録されていないため、計算を行い、クエリの変更前、変更後の組 (C, Z) を登録し、クエリ Z を返す。クエリ B はクエリ変更処理によって変更されなかった例であり、このときメモリの節約のため変更後のクエリは登録しない。

5.2　構文解析規則の動的変更

第 4 節で述べたように、PEG の構文解析規則では演算子 / を用い、A/B のとき、A を先に試行し、失敗した場合は B を試行する。つまり $A/B/C/D/\ldots$ のとき、A, B, C, D, ... と順に試行することになる。先の出現ほど早く試行され、後ろの出現は後回しになるため、試行が成功しやすいもの（頻出しているもの）が先に出現していれば失敗率は低くなる。

第 3.1 節で述べた InfluxDB のクエリ言語 Flux は、最初に大きな時系列データの集合を取得してから、関数の繰り返し適用で目的の系列データや値を計算するような関数型クエリである。この Flux を認識する PEG の構文解析規則の一部を図 9 に示す。`TimeSeries` は Flux の一行目であり、`from` 関数など時

```
Flux <- TimeSeries ("|>" Func)*
Func <- Filter / Fill / Keep / Range / ...
```

図 9　Flux の構文解析規則（PEG）の一部

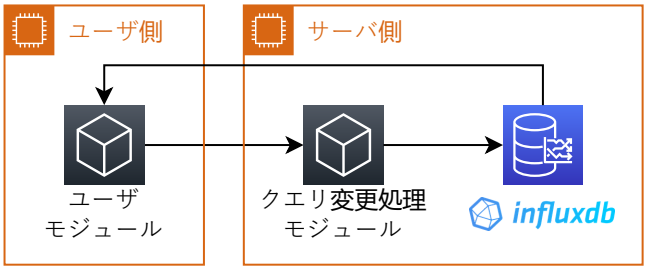

図 10　実 験 環 境

系列データの取得の処理を含む非終端記号である。その後 0 個以上の Func 非終端記号が連続するが、これは、Flux において関数を繰り返し適用する様子を表している。この Func は図 4 でも示されているようにそれぞれの関数に対応する約 50 種類の非終端記号が演算子 / によって連結されている。

以上より、Flux で頻出する関数に対応する構文解析規則の非終端記号ほど先頭にあるほど、この Func 非終端記号の右辺の規則は失敗するような試行が減ることになり、高速化が期待される。

本提案では、Func 非終端記号の規則において、右辺の非終端記号のそれぞれが試行に成功するたびに、左の非終端記号と位置を入れ替える。例えば、図 9 の状態から Keep 非終端記号の試行が 2 回成功したとき、2 回左の非終端記号と位置を入れ替える。結果として `Func <- Keep / Filter / Fill / Range / ...` となる。

6　評 価 実 験

本節では、第 5 節で提案した手法の有効性について評価実験を行った。実験環境の概略図を図 10 に示す。

ユーザモジュール、クエリ変更処理モジュール、TSDB(InfluxDB) の 3 つを用意した。ユーザモジュールはユーザに相当し、クエリ変更処理モジュールに対して自動的にクエリを発行する。クエリ変更処理モジュールはユーザモジュールからクエリを受け取り、PEG に基づいた構文解析を行い、クエリを変更し、変更済クエリを InfluxDB に送信する。どちらのモジュールも InfluxDB に合わせて Golang [26] で記述した。InfluxDB は ver2.5 を用い、クエリ変更処理モジュールからクエリを受け取り、ユーザモジュールに応答する。Amazon EC2 t3.large インスタンスを 2 つ用意し、一方をユーザ側としてユーザモジュールを動作させ、もう一方サーバ側としてクエリ変更処理モジュールと InfluxDB を動作させる。

次に、InfluxDB 上に完成済みのデータセットを用意した。サンプル数（タグとフィールドの組み合わせの数）は 100 で、全ての 0.1 秒間隔のデータ、合計で 24 時間分を用意する。合計で

```
from(bucket: "tsdb")
 |> range(start: X-1h, stop: Xh)
 |> aggregateWindow(every Ys, fn: mean)
```

図 11　ユーザモジュールが自動発行するクエリ

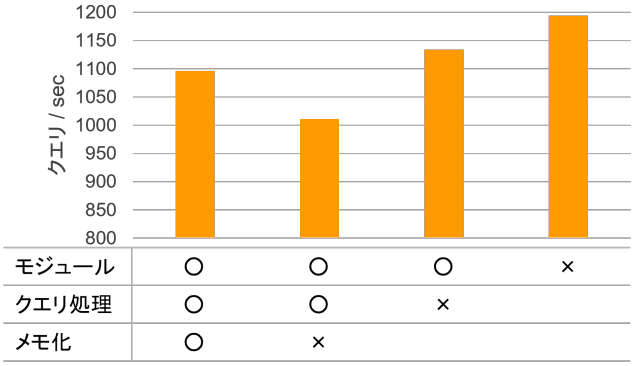

図 12　クエリの変更内容のメモ化：平均スループット

データポイント数は 86,400,000 となる。データセットは完成済みで、実験中に並行して書き込みを実行はしないものとする。

最後に、発行するクエリとクエリの変更方法について設定した。ユーザモジュールが自動的に発行するクエリを図 11 に示す。

このクエリは $X-1$ 時間から X 時間の 1 時間の全データポイントを取得し、そのデータポイントを Y 秒毎に平均を取ることを表している。X と Y はランダムに設定され、それぞれ $X = \{-23, -22, -21, \ldots, 0\}$, $Y = \{0.1, 1, 10\}$ とする。つまり、X は 24 通り、Y は 3 通りで計 72 通りのクエリがランダムで生成される。ユーザモジュールではこのランダムクエリを発行して応答があったら発行、を繰り返し一分間実行した。クエリ変更モジュールでは受け取ったクエリに対して、それぞれ Y を 10 に変更する。つまり元から $Y = 10$ であれば変更は発生しない。

6.1　クエリの変更内容のメモ化

本節では第 5.1 節で述べた 1 つ目の提案手法であるクエリの変更内容のメモ化について検証する。クエリ変更処理モジュールにメモ化機能を実装し、メモ化を有効化した場合と無効化した場合を比較した。参考に、クエリ変更モジュールを経由するが全くクエリ処理（構文解析及び変更）を実行しない場合と、クエリ変更処理モジュールを経由せず、ユーザモジュールから InfluxDB に直接クエリを発行する場合も比較した。

4 つの場合についての比較を、平均スループットについて図 12 に、クエリ変更処理モジュールのメモリ使用量について図 13 に示す。

メモ化機能の有効化と無効化の場合を比べて、メモ化機能を有効にした場合の方が 10%程度高速で処理できることが検証できた。一方でメモ化にメモリを使用した分、メモリ使用量は大きくなり、メモ化の高速化の代わりにメモリを使用するトレードオフがあることを確認できた。最初はメモ化されてないクエリが多いためメモ化無効の場合と比べて差が少ないが、多くのクエリが登録されてしまえば安定して高速にクエリ変更処理をすることができた。メモ化有効の場合とクエリの処理を行わない場合と比べると、当然クエリ処理を行わない場合の方が早いものの、大きく性能が低下することなくクエリ変更処理ができた。

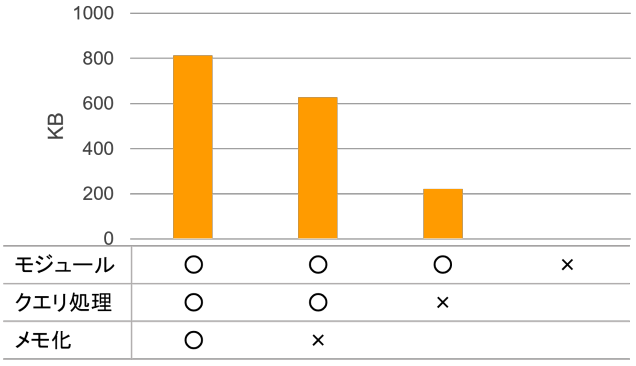

図 13　クエリの変更内容のメモ化：メモリ使用量

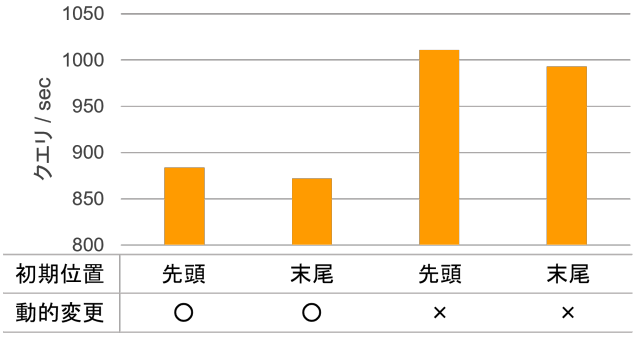

図 14　構文解析規則の動的変更：平均スループット

6.2　構文解析規則の動的変更

本節では第 5.2 節で述べた 2 つ目の提案手法である構文解析規則の動的変更について検証する。Flux クエリを認識する規則は図 9 のようになっているが、実験で用いるクエリは図 11 のようになっており、Func 非終端記号の規則の右辺のうち、選択される非終端記号は Range と AggregateWindow の 2 つだけである。まず、Func 非終端記号の規則の右辺において 2 つの非終端記号を初期位置を先頭、及び末尾にした 2 種類の規則を用意する。次に規則の動的変更を有効化した場合と無効化した場合を用意する。最後に以上の非終端記号の初期位置の前後、動的変更機能の有無の組み合わせ 4 種類をを比較する。初期位置が先頭で動的変更昨日が無効のときは、第 6.1 節の実験におけるメモ化が無効のときと同じである。

平均スループットについて 4 種の比較を図 14 に示す。

まず、非終端記号の初期位置の前後同士で比較すると、先頭にあるほど早くはなるものの、あまり大きな差は見られないことが分かった。PEG のパーサにはパックラット構文解析 [27] を利用している。このアルゴリズムは、構文解析の途中結果をメモ化することで、同じ非終端記号・入力列に対しては 1 度しか計算しない。これにより非終端記号の試行結果を早く応答でき

るため、非終端記号の位置の前後では大きな差が出ないと考えられる。

次に、動的変更を有効化した場合と無効化した場合を比較すると、有効化したときの方が約 20%低速になることが分かった。これは非終端記号を入れ替え、書き換えるオーバヘッドが大きいことを表している。特に本実験では `Range` と `AggregateWindow` が交互に出現するため、この 2 つの非終端記号を何度も入れ替えることになるのが原因と考えられる。

7 今後の課題

まず本稿の提案手法の 1 つ目、クエリの変更内容のメモ化について考える。この手法はメモ化により再計算を防ぐことで高速化を実現するものだが、トレードオフとしてメモリ使用量が増加する（図 13)。そのためメモリ使用量を節約が課題の 1 つである。本実験では受け取るクエリが決まっていたが、頻出するクエリがある一方、一度しか受け取らなかったクエリもあると考えられる。このようなクエリはメモ化の意味が少なく、メモリの無駄遣いになるだけの使われてないものは時間経過などで自動的に削除する機能が必要である。また、大部分は同じだが、細部が異なるクエリが多く発生することも考えられる。例えば、`filter` 関数で指定するタグ・フィールドや `range` 関数の数字だけが異なる場合である。このような複数のクエリでは異なる値のみを変数とし、変数部分を個別に具体的な値とその変更結果の組をメモ化することでメモリ使用量の削減に繋がると考えられる。

次に本稿の提案手法の 2 つ目、構文解析規則の動的変更について考える。本実験では規則の書き換えが頻繁に行われてことで多くのオーバヘッドが発生したが、これを軽減する方法が考えられる。例えば一定時間にそれぞれの非終端記号が呼び出された数をカウントし、頻出する順に規則の非終端記号を並べ替える。これにより書き換えの頻度は大きく減少させることができる。

最後に、この事前構文解析・クエリの変更処理そのものについて考える。本実装ではまだ単一のユーザについてしか考慮されておらず、実際には様々なユーザがクエリを発行する。ユーザによってはクエリの変更ルールが異なるため、それを区別できる機能が必要である。また、このクエリの変更ルールは、プログラムの実行後も変更できる機能も必要である。このとき、クエリの変更内容のメモ化においては一度メモをリセットする必要がある。

8 ま と め

本稿では、TSDB におけるクエリの事前構文解析・変更処理の効率化手法について提案した。

まず、前提として TSDB やそのクエリ言語の特徴、TSDB の実装の 1 つである InfluxDB、及び本稿の事前構文解析に用いられる形式文法 PEG にについて解説した。

これを踏まえて 2 つの手法を提案した。まず TSDB のクエリは短く、同じクエリが高頻度で発行される特徴がある。この特徴を活かし、クエリの変更内容をメモ化することで再計算をしないようにする手法を提案した。次に PEG に基づいた構文解析は、規則の実行に順序付けがされ、先頭ほど、つまり頻出するほど早く処理される。また、InfluxDB のクエリ言語 Flux は、時系列データに関数を繰り返し実行することで目的の系列データや値を計算するが、この関数は約 50 種類あり出現頻度が異なっていた。この特徴を活かし、規則の動的に変更することで頻出する関数に対応する構文解析処理を高速で行う手法を提案した。

この 2 つの提案手法について評価実験を行い、有効性について検証した結果、特にクエリの変更内容をメモ化する手法において高速化できることが分かった。

文　献

[1] Kevin Ashton, et al. That‘internet of things’ thing. *RFID journal*, Vol. 22, No. 7, pp. 97–114, 2009.

[2] NOSQL DEFINITION. https://hostingdata.co.uk/nosql-database/.

[3] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal*, Vol. 10, No. 2, pp. 199–223, 2001.

[4] Yong Yao, Johannes Gehrke, et al. Query processing in sensor networks. In *Cidr*, pp. 233–244, 2003.

[5] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 1461–1476, 2018.

[6] InfluxDB. https://www.influxdata.com/.

[7] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 111–122, 2004.

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

[9] Memcached. https://memcached.org/.

[10] Redis. https://redis.io/.

[11] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2, pp. 35–40, 2010.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, Vol. 26, No. 2, pp. 1–26, 2008.

[13] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, Vol. 33, No. 4, pp. 351–385, 1996.

[14] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pp. 107–141, 1970.

[15] kdb+. https://kx.com/.

[16] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp. 421–436, 2019.

[17] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. Littletable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 125–138, 2017.

[18] Prometheus. https://prometheus.io/.

[19] GridDB. https://griddb.net/.

[20] Atlas. https://github.com/Netflix/atlas/.

[21] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, Vol. 8, No. 12, pp. 1816–1827, 2015.

[22] Heroic. https://spotify.github.io/heroic/.

[23] M3. https://www.m3db.io/.

[24] Db-engines ranking of time series dbms. https://db-engines.com/en/ranking/time+series+dbms.

[25] In-memory indexing and the time-structured merge tree (tsm). https://docs.influxdata.com/influxdb/v1.8/concepts/storageengine/.

[26] The go programming language. https://go.dev/.

[27] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. *ACM SIGPLAN Notices*, Vol. 37, No. 9, pp. 36–47, 2002.