修士論文

shellstreaming: シェルコマンド群による分散処理のためのストリーム処理系

shellstreaming: A Stream Processor for Distributed Computing by Shell Commands

平成 26 年 2 月 6 日提出

指導教員　　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-126430 中谷 翔

概 要

Web の発達した近年，大規模ストリームデータを扱う処理系の需要は大きい．複雑なストリーム処理を記述する場合には，処理に用いるジョブをカスタマイズする機能は必須であり，多くのストリーム処理系では UDF (User-Defined Function) やアダプタといった形式でジョブの拡張をサポートしている．しかし，多くの処理系がサポートする拡張方法では，処理に組み込みたいプログラムを処理系に合わせて変更・記述する労力をストリーム処理アプリケーションの開発者が負うこととなり，生産性が低下する．プログラミング言語や開発者を問わず，様々なシェルコマンドをストリーム処理系に組み込むことが可能ならば，高生産なストリーム処理アプリケーションの開発の大きな助けとなる．

本研究では，シェルコマンドをジョブとして組み込むことのできるストリーム処理系，shellstreaming を提案する．本稿では，shellstreaming の基本的な利用例，アーキテクチャ，構成要素について説明し，特にシェルコマンドの入出力と shellstreaming のデータ構造の変換方法について詳説する．また，shellstreaming が行なっている種々の高速化についても説明し，特にシェルコマンドをデーモン化する手法について詳しく述べる．

評価では，複数のワーカ間で通信が発生する場合の性能への影響，デーモン化手法を含むシェルコマンドオペレータの性能評価，Storm[21] 処理系との性能比較を示し，考察した．高速化手法の中心であるシェルコマンドのデーモン化の効果により，ストリーム処理のスループットが顕著に向上することを示し，また Storm との比較でも，起動時間の長いシェルコマンドを含むストリーム処理では 15 倍高速であることを示した．

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

1.1.1　大規模データ，ストリームデータ

コンピュータネットワークの発展に伴い，人々が扱う電子データの量は増加の一途をたどっている．IT 専門調査会社大手 IDC の 2012 年末の報告 [1] は，2005 年の時点の総量が 130 エクサバイトだった電子データは，2020 年に 40,000 エクサバイトに至ると指摘する．また，大手 SNS の Twitter は，2010 年の時点において一日当たり 12TB ものデータを扱っている [43] と報告した．近年になり，このような潮流はビッグデータという言葉に象徴されるようになった．

巨大なデータから人々にとって有益な知を抽出するためには，データは解析される必要があり，先に述べたようなデータ量の増加の速度を顧みると，高速なデータ解析が多くの分野において不可欠である．大量のデータを高速に扱うためのプラットフォームとして，複数のコンピュータノードをネットワークでつないだクラスタ環境は非常に一般的になりつつある．1 つの CPU コアについて見たとき，その動作周波数を上昇させて性能向上を実現するアプローチは，発熱などの物理的な制限により限界を迎えている．従って，近年の CPU の性能向上は主に CPU ソケット内に複数のコアを搭載することによって実現されている．これと同様に，1 台のノードの性能向上のためにはノード内に複数の CPU ソケットを搭載し，ハードウェア環境全体の性能向上のために複数ノードをコンピュータネットワークで接続しクラスタ環境を構築するのである．ノード同士を接続してクラスタ環境を構成するアプローチは，近年のスーパコンピュータやデータセンタの常識となっている．そして，大規模なデータを扱う企業や研究所，或いは個人に至っても，自前のクラスタ環境や AWS (Amazon Web Service) に代表されるサービスとしてのクラスタ環境の利用が広がっている．

データを処理する各々の主体が，データ処理の際にとるアプローチは，データを蓄えて処理する方法とデータを蓄えずにリアルタイムに処理する方法に大別できる．

前者の方法はバッチ処理と呼ばれる．過去の情報まで遡った解析が可能なメリットがある一方で，二次記憶装置を大量に使用するのと，二次記憶装置の使用による解析速度の低下が問題となるケースが多い．後者のアプローチはストリーム処理と呼ばれる．データをリアルタイムに解析したい要求がある場合に，ディスクを消費せず高速なオンメモリ処理で実現できるメリットがある．

近年の Web やセンサネットワークの発展を受け，あらゆるところでストリームデータを目にすることになった．大手 SNS の Twitter は，Twitter Streaming API[24] を通じて開発者にツイートデータをストリームデータとして提供する．また，Web サイトへのアクセスをストリームデータとして処理する研究・事例もある [48, 25]．更に，スマートフォンの普及に伴い，スマートフォ

[1] *The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East.* http://www.emc.com/leadership/digital-universe/iview/executive-summary-a-universe-of.htm

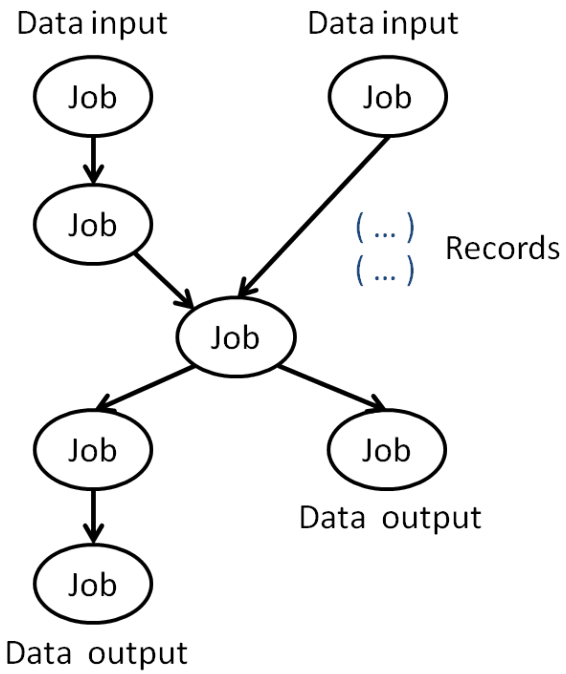

図 1.1　ストリーム処理の基本的な概念図

ンをセンサとして利用することで人々のライフログを収集し，それをサーバに送って逐一解析するようなアプリケーションもその数を増やしている．

ストリームデータの処理の重要性，感心の高さは，トップカンファレンスと評される SIGMOD (Special Interest Group on Management of Data) において，ストリームデータの処理が 2009 年に端を発し 2011 年から 2013 年まで続けて “Topic of Interest” と扱われていることからも窺える．

1.1.2　ストリーム処理

ストリーム処理の性能上の目的は，より大量のデータをリアルタイムに処理することである．ストリーム処理の扱えるデータ量の指標には，データサイズと処理の開始点から終了点までに要した時間の比であるスループットを用いる．これはバッチ処理でも用いられる指標である．ストリーム処理とバッチ処理に求められる性能面での最も大きな違いは，ストリーム処理では個々のデータ処理に要する時間が重要になることである．リアルタイムな処理であるためには，1 つのデータが処理の開始点から処理の終了点までに要するためのレイテンシが小さいことが重要である．ストリーム処理は，スループットを可能な限り大きく，レイテンシを可能な限り小さくする性能上の目的を持つ．

先に「個々のデータ」と述べたものはストリーム処理においてレコード (或いはタプル，イベント) と呼ばれる．レコードは意味上の最小のまとまりを単位とすることが基本であり，これは RDBMS (Relational Database Management Systems) が基礎とするリレーショナルモデル [36] におけるタプルと同様の概念である．

ストリーム処理の基本的なモデルを図 1.1 に示す．ストリーム処理の内容は，個々のジョブ (或

いはオペレータ，クエリ) とそれらの間のデータフローによって記述される．図 1.1 はジョブグラフ (或いはワークフロー) と呼ばれる DAG (Directed Acyclic Graph) であり，グラフのノードがジョブを，エッジがジョブ間のデータフローを示す．ストリーム処理のモデルにおいては，データを蓄積せずにリアルタイムに処理するため，データ入力は無限列であり得る．

レコード毎の処理遅延 (レイテンシ) を小さくする目的を達成するため，各ジョブはイベントドリブンで駆動する．即ち，レコードが上流のエッジからジョブに到達すると，ジョブはそのレコードに対する処理を開始し，処理結果を下流のエッジに出力するというのがジョブの基本動作である．この動作から，ストリーム処理は CEP (Complex Event Processing) [39] と呼ばれることもある．

ストリーム処理の枠組みで記述されるアプリケーションは様々ある．サーバへのアクセスが発生するたびに，追記されたアクセスログから必要な情報を抽出し，直近数日間の集計結果にマージするような比較的単純な例 (3.1 節で具体例を紹介) や，Web を定期的にスクレイピングし，得られたテキストデータを様々なテキスト処理モジュール (構文解析やインデキシング) によって解析し，より構造化された情報を得るような複雑な例も挙げられる．

1.1.3　ストリーム処理アプリケーション開発

ストリーム処理で記述するアプリケーションが複雑なものであれば，ジョブグラフの各ジョブに複雑な処理が要求される場合が多い．もちろん，基本的なデータ操作[2]の組み合わせである程度複雑な処理は実現することができるが，例えば日本語テキストが入力レコードとして与えられ，その中に含まれる形態素群を出力するような処理は，基本的なデータ操作では最早対応できない．

ストリーム処理系よりも歴史の長いデータベース処理系を使用したアプリケーション開発では，この問題に対して大きく分けて 2 つのアプローチをとることができる．

1. 基本的なデータ操作によって必要となるデータをデータベースから抽出し，プログラミング言語上で複雑な処理を適用する．プログラミング言語のプラグインとして提供される，データベースエンジン操作のインターフェイス (JDBC など) を用いた処理がこれに当たる．
2. 複雑な処理をデータベースエンジンプラグインとして開発し，データベースへのクエリの際にその処理を内部的に適用させる．UDF (User-Defined Function) と呼ばれる機能がこれに当たる．

ストリーム処理においては，扱うデータが無限列であることもあるため，後者のアプローチが取られる．即ち，ストリーム処理に複雑なジョブを組み込む場合には，UDF と呼ばれるような拡張ジョブを予め定義することとなる．

拡張ジョブを定義しジョブグラフに組み込むための方法は，ストリーム処理系によって異なるものである．しかし，多くの処理系において採られるアプローチは次のようなものである．拡張ジョブは，定められたプログラミング言語の関数として定義する．この関数定義があるライブラリやスクリプトファイルを，ストリーム処理系が動作時に組込み，拡張ジョブ関数をイベントドリブンで駆動する．このアプローチのメリットは，拡張ジョブの入出力が，関数の引数やパラメタとして制御しやすいものになるところである．一方で，ストリーム処理のアプリケーション開

[2] RDBMS (Relational Database Management Systems における選択，射影，整列，結合，のような操作を意図している．)

発者からすると，拡張ジョブ記述のためにプログラミング言語や関数の規約を強制されることによる生産性の低下がデメリットとして挙げられる．特に，拡張ジョブとして組み込みたい動作を実現するプログラムを既に所持している場合には，それをストリーム処理系に適合させるために記述し直す必要がある．

我々は，ストリーム処理系がシェルコマンドを拡張ジョブとして組み込む機能を持つことが，アプリケーション開発の生産性において極めて重要と考える．ここでシェルコマンドとは，シェル上で実行可能なあらゆるプログラムを指す．例えば，シェルスクリプトや Ruby, Python, Perl のような各種軽量言語のスクリプト，C や C++に代表されるコンパイル型言語から生成される実行ファイル，あるいは R や MATLAB といったより特定領域に特化したスクリプトの全てが該当する．もちろんそれらは，ストリーム処理アプリケーション開発者が自ら記述したプログラムであっても，標準的なコマンドや各種パッケージシステム経由で入手したプログラムであってもよい．

我々は，複雑なストリーム処理を高生産に実現するために，シェルコマンドを拡張ジョブとして組み込むこと，更に，シェルコマンドによるジョブが持つ性能上の特性にも考慮し，高効率な実行を可能にすることが重要と考える．

1.2　研究の動機

本研究では，シェルコマンドをオペレータとし分散実行するストリーム処理系，shellstreaming の設計・実装を行った．

単一ノード上での実行を念頭に置いて作成されたシェルコマンドのうち一定の制約を満たすものを，ユーザが僅かな労力で大規模分散ストリーム処理に組み込めるシステムを目指したものである．

本システムの応用例の一つとして，次のような災害対策アプリケーションを念頭に置いている．2011 年 3 月の東日本大震災は記憶に新しいが，このような大規模な災害発生時に，Twitter などの SNS に流れる情報を元にし，より価値の高い情報に随時まとめあげる．より具体的に検討しているのは，災害発生時にコンピュータクラスタが Twitter からのストリームデータを受信し，それに対し自然言語処理を施し,「食料の配給場所」といった被災者に有益な情報を抽出してすぐさま配信するというものである．ここで求められるのは，ツイートデータを秒単位で反映するリアルタイム性である．新しい情報をできる限り高速に配信するためには，情報を蓄えてから加工するバッチ処理ではなく，ストリーム処理の枠組みが適している．この処理を実現するためのプログラム要素は別途開発されているため，それらのシェルコマンドをストリーム処理に組み込むコストを抑えることが重要となる．

更に，シェルコマンドをストリーム処理に実用的に組み込むためには，シェルコマンド起動時のオーバヘッドをストリーム処理系が管理する必要がある．具体的には，プロセスとして起動するシェルコマンドの起動に要する時間，起動しているだけで消費するメモリフットプリントを考慮しなければ，継続的な低遅延処理は実現できない．本研究では，シェルコマンドのデーモン化 (4.1 節) を中心に，ストリーム処理の拡張ジョブとしてのシェルコマンドの高効率実行を目指す．

1.3　本研究の貢献

本研究の貢献は以下の通りである．

3.6.2 節において，シェルコマンドによる拡張ジョブをストリーム処理の枠組みに組み込むためにシェルコマンドに課せられるべき制約と，その上で実際にストリーム処理系とシェルコマンドが行うべきデータのやりとりについて考察する．更に，4.1 節でシェルコマンド拡張ジョブの実行を高速化するためのデーモン化手法を提案する．

その他，4.2, 4.3, 4.4 節でストリーム処理系としての基本的な高速化手法を述べ，5 章でそれらの評価を行った．

評価では，高速化手法の中心であるシェルコマンドのデーモン化の効果により，ストリーム処理のスループットが顕著に向上することを示し，また Storm[21] との比較でも，起動時間の長いシェルコマンドを含むストリーム処理では 15 倍高速であることを示した．

1.4　本稿の構成

本稿の以降の構成は次のとおりである．まず 2 章で本研究の関連研究について述べる．次に 3 章において，シェルコマンドを用いた分散ストリーム処理を実現するための shellstreaming を提案する．ここでは，システムの応用例，アーキテクチャ，動作，アプリケーションの記述方法など，shellstreaming の基本的な部分について詳説する．4 章では，shellstreaming のアプリケーション実行を高速化するための手法を提案する．そして 5 章で shellstreaming の性能評価を行う．評価には，4 章で述べた高速化手法の効果測定と，現在最も広く利用されているストリーム処理系であると考えられる Storm[21] との性能比較を含む．最後に 6 章で本研究の総括を示し，今後の展望について述べる．また，shellstreaming の入手方法，アプリケーション例を付録に記した．

第2章　関連研究

2.1　ストリーム処理系

大規模データを分散・並列処理するためのシステムとしては，ストリーム処理系よりもバッチ処理系の方が普及が速く，現在までに多数のバッチ処理系が提案・利用されている．特に，Google の MapReduce[37] から着想を得た OSS (Open Source Software) の Hadoop[3] や，Hadoop を基盤とした研究やシステム [41, 5, 2, 4, 1] を取り巻く動きが大きい．

一方で，扱うデータのサイズが増加することで，リアルタイム性やディスク消費量の小ささが求められるアプリケーション開発の需要も伸びてきていて，ストリーム処理系に関する研究や実際に多数のユーザを抱えるシステムの数も近年増加している．著名なものだけでも，商用システムでは InfoSphere Streams (IBM)[11], Oracle/CEP (Oracle)[15], StreamInsight (Microsoft)[14] が，OSS(Open Source Software) システムでは Esper[7], Storm [21], Spark Streaming [20, 47], S4[18, 32], Fluentd[9] などが挙げられる．この他にも，オンライン機械学習用の Jubatus[13] のように，特定の処理領域に特化した処理系も存在するが，本章では汎用的なストリーム処理系に焦点を当てる．

これらのストリーム処理系に対し，拡張ジョブの記述方法という観点で分類を行ったのが表 2.1 である．1.1.3 節で示したように，複雑なストリーム処理を実現するためには拡張ジョブの作成が不可欠であり，拡張ジョブの記述方法はアプリケーション開発の生産性に直結する．そのため，Storm や Fluentd のような処理系では拡張ジョブを多数のプログラミング言語で作成するためのアダプタを設けている．

一方で，シェルコマンドを拡張ジョブとして組み込める処理系は，shellstreaming を除くと Storm のみである．複数のプログラミング言語に対応させるために，ストリーム処理系がアダプタを提供する方法には限界がある上，アプリケーション開発者は既に拡張ジョブに相当する機能を持った実行可能なプログラムを有していたとしても，アダプタに適合させるためにそのプログラムを記述し直す必要がある．我々は，シェルコマンドをそのまま拡張ジョブとして利用することの重要性は大きいと考える．

2.2　大規模分散データ処理におけるシェルコマンドの利用

表 2.1 に示したように，我々の調査の限り，シェルコマンドを拡張ジョブとして利用できるストリーム処理系は Storm のみである．Storm は現在 Twitter 社が保有している OSS であり，もちろん Twitter サービスのバックグラウンドでも稼働している．Twitter 社以外での採用実績も数多い[1]．

[1] https://github.com/nathanmarz/storm/wiki/Powered-By 参照.

Storm はシェルコマンドを拡張ジョブとして利用するためのインターフェイスとして，ShellBolt[19] を持つ．ここで Bolt とは，Storm におけるジョブの総称と考えれば良い[2]．

たとえ ShellBolt を使っても，シェルコマンドを直接的に利用することはできない．Python または Ruby を用いて，Storm のタプルを標準入出力からやりとりするためのアダプタを記述し，アダプタの中で所望のシェルコマンドを実行する必要がある．

ShellBolt を使った実際の例として，cat コマンドを拡張ジョブ (Bolt) として利用する方法を示す．コード 2.1 が Python によるアダプタの記述で，コード 2.2 でそのアダプタを使用している．

コード 2.1　cat コマンドを用いた ShellBolt 作成 (Python サイド, cat.py)

```
1  import storm
2  import shlex
3  from subprocess import Popen, PIPE
4
5  class CatBolt(storm.BasicBolt):
6      def process(self, tup):  # called per-tuple manner
7          # invoke `cat` command
8          p = Popen(shlex.split('cat'), stdin=PIPE, stdout=PIPE)
9
10         # input tuple's contents into `cat`
11         p.stdin.write(tup.values[0])
12         p.stdin.flush()
13         p.stdin.close()
14
15         # get output from `cat` and emit it as tuple
16         col0 = p.stdout.read()
17         storm.emit([col0])
18
19 CatBolt().run()
```

コード 2.2　cat コマンドを用いた ShellBolt 作成 (Java サイド)

```
1  public static class CatBolt extends ShellBolt implements IRichBolt {
2
3    public CatBolt() {
4      super("python", "cat.py");
5    }
6
7    @Override
8    public void declareOutputFields(OutputFieldsDeclarer declarer) {
9      declarer.declare(new Fields("col0"));
10   }
11 }
```

特に，コード 2.1 の 6 行目を見ると，ShellBolt のアダプタはタプルを 1 つずつ取ってそれを実際のシェルコマンドに入力するインターフェイスになっていることが確認できる．これは，特にシェルコマンドプロセスの起動が低速である場合には大きなオーバヘッドとなり得る．

このように，Storm の ShellBolt は，シェルコマンドを拡張ジョブとして利用することは可能であるが，そのための方法は直接的でなく，また 1 タプルずつシェルコマンドを起動するためのオーバヘッドも掛かってしまう．

範囲をストリーム処理系から広げると，シェルコマンドを活用する大規模分散データ処理システムは様々見受けられる．特に，ワークフロー処理における使用に特化した並列 RDBMS の ParaLite[35, 34] は，UDF として扱うシェルコマンドの持つ制約に関して shellstreaming のシェ

[2] Storm ではタプルを処理する Bolt の他に，外部データソースからタプルを生成する Spout が処理の構成要素としてある．

ルコマンドオペレータと近しい．shellstreaming は ParaLite と比べ，シェルコマンドの出力を処理系で使用するレコードに変換する機能が柔軟であり (3.9.2 節)，更にシェルコマンドの起動コストを削減するためのデーモン化手法を提案している (4.1 節)．また，ParaLite はバッチ処理のワークフローを対象としたシステムであり，この点において shellstreaming とは用途を異にしている．

他のバッチ処理のワークフロー基盤システムで，シェルコマンドをジョブとして利用するものに，GXP Make[45] や Makeflow[26] が挙げられる．これらのシステムはシェルコマンドの入出力文字列に対し特別な扱いをしないが，shellstreaming や ParaLite は，システムの扱うカラム分割されたレコードと，シェルコマンドの入出力テキストの変換をする仕組みを持つ．

また，MapReduce フレームワークの Hadoop[3] の Hadoop Streaming[10] という API を利用することで，シェルコマンドを map ジョブや reduce ジョブとして利用することができる[3]．

このように，様々な処理系がジョブの拡張手段としてシェルコマンドをサポートしていることから，アプリケーション開発におけるシェルコマンド活用の需要の大きさが伺える．しかし，ストリーム処理の中で実用的なレベルにシェルコマンドをサポートしているものは見受けられない．我々はこのようなストリーム処理系を開発すると共に，4.1 節においてシェルコマンドのデーモン化手法を提案する．

[3] Hadoop *Streaming* という名称は，シェルコマンドジョブの入出力を標準入出力ストリーム (stdin/stdout) で扱う点から来ていると考えられ，ストリーム処理とは無関係である．

[1] http：//www.ibm.com/developerworks/library/bd-streamsrtoolkit/ より．

[2] http：//docs.oracle.com/cd/E21764_01/apirefs.1111/e12048/funcusr.htm より．

[3] http：//technet.microsoft.com/en-us/library/ee842720.aspx より．

[4] 開発レポジトリ https：//github.com/s4/s4/tree/master/s4-driver でプログラミング言語とのアダプタを確認．

表 2.1　各ストリーム処理系のジョブ拡張方法．脚注がない限り，拡張ジョブ記述の言語は公式サイトのトップページを参照にした．

ストリーム処理系	拡張ジョブ記述	シェルコマンドジョブとのデータ受け渡し
InfoSphere Streams (IBM)	C++, Java [1]	
Oracle/CEP (Oracle)	Java [2]	
StreamInsight (Microsoft)	C# [3]	
Esper	Java	
Storm	Closure, Java, Scala, JRuby Perl, PHP, シェルコマンド (ShellBolt)	△
Spark Streaming	Java, Scala, Python	
S4	Java, Perl, Python[4]	
Fluentd	Ruby, Java, Python, PHP, Perl, Node.js, Scala	
shellstreaming	シェルコマンド (シェルコマンドオペレータ), Python	○

第3章　提案システム: shellstreaming

本研究では，シェルコマンドを用いた分散ストリーム処理を実現するための shellstreaming を提案する．本章においては，shellstreaming の応用例，アーキテクチャ，ストリーム処理系としての基本機能を説明し，シェルコマンドをオペレータとして扱うための手法とデーモン化する手法について述べる．更に，shellstreaming を用いたアプリケーション開発を支援する機能についても紹介する．

3.1　システム概要

shellstreaming は，シェルコマンドを用いて様々な分散ストリーム処理を実現するためのストリーム処理系である．通常のストリーム処理の枠組みに収まる限り汎用的に利用できるものであるが，特に次のような場合の利用を推奨する．

- ストリーム処理に組み込みたいプログラムを既に所持している場合．
- ストリーム処理に組み込みたいプログラムを，習熟度の高いプログラミング言語を用いて開発したい場合．特に，表 2.1 の中に使用したいプログラミング言語がない場合には，最もアプリケーション開発コストの小さい選択となり得る．

想定されるユースケースの具体例として，Apache HTTP Server[22] (Apache ウェブサーバ, apache2) のアクセスログを解析するストリーム処理アプリケーションを挙げる．アプリケーションの構成は図 3.1 のジョブグラフで表される．大半のジョブが，grep, sed, awk, sort, uniq といった馴染み深い POSIX[23] コマンドで構成されている点に注目いただきたい．shellstreaming はこのような既存のシェルコマンドを組み合わせたアプリケーションの生産性が非常に高い[1]．

このアプリケーションは，Apache HTTP Server のアクセスログの追記を監視し，2014 年 1 月 1 日から 2014 年 1 月 4 日までの日毎のアクセス数とレスポンスのステータスコードを出力するものである．出力のイメージは図 3.2 を参照されたい．

ここでは，アプリケーションの動作環境を次のようなものと想定し，動作説明を行う．

ウェブサーバノード (アクセスログ出力先) WebServer0, WebServer1, WebServer2

ワーカノード (アクセスログ解析ノード) WebServer0, WebServer1, WebServer2, Worker0, Worker1

即ち，ウェブサーバは負荷分散のために 3 台に分散されており，アクセス解析のストリーム処理も 5 台で分散されて行われる．

1. WebServer0-2 が，各々のローカルディスクに存在するアクセスログファイルを監視し，追記があるたびにその行をレコードとして生成する (0:TextFileTail).

[1] Apache HTTP Server のアクセスログ解析アプリケーションの全コードは付録 B.1 に掲載した．

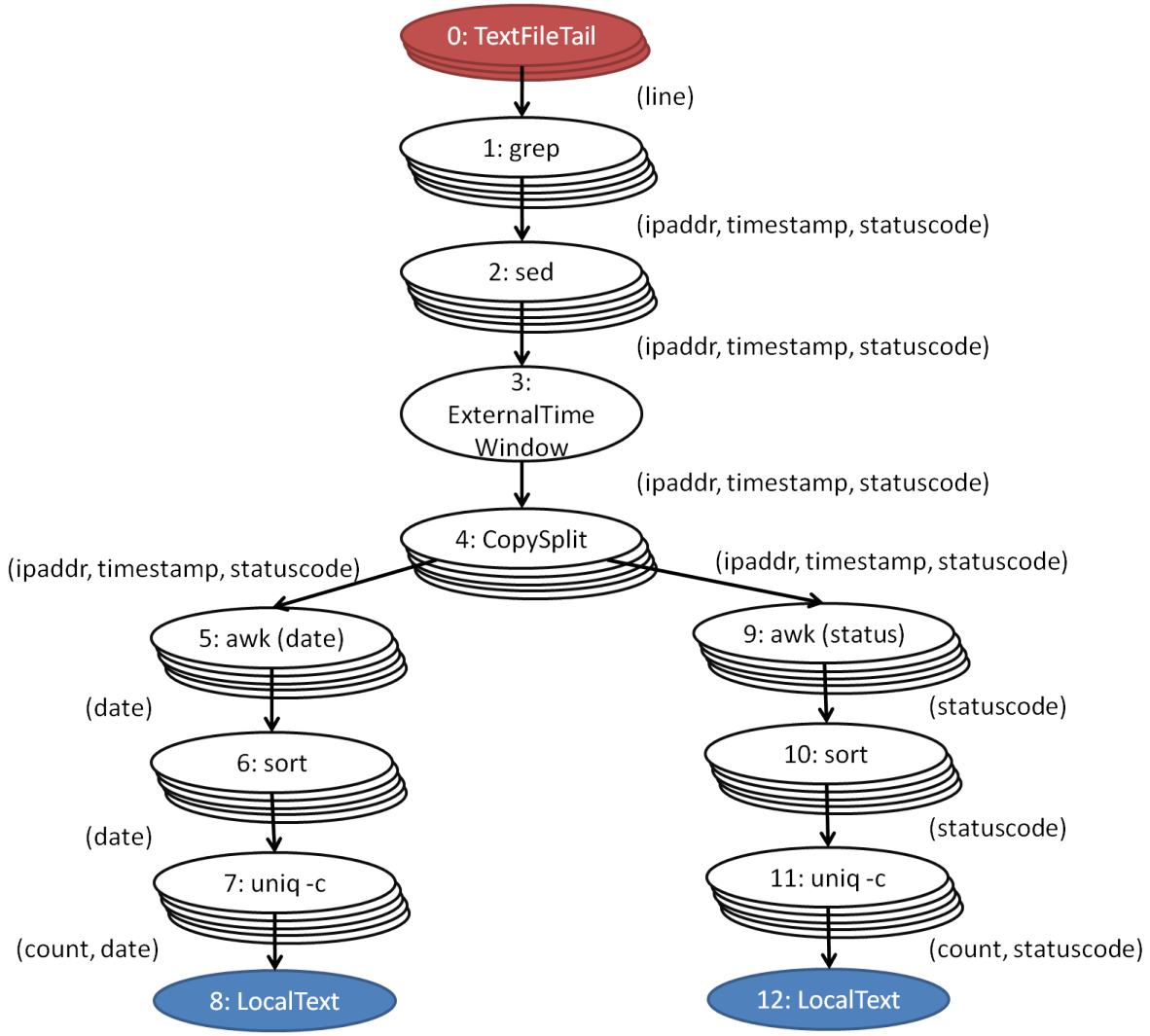

図 3.1　Apache HTTP Server アクセスログ解析アプリケーションのジョブグラフ

2. WebServer0-2, Worker0-1 が，上流のジョブから出力されたアクセスログのレコードを入力とし，GET メソッドでトップページ “/” にアクセスしているもののみを選択する．更に，選択したログテキストから IP アドレス, アクセス時刻, ステータスコードの情報を分け，それぞれをカラムとして持つレコードを出力する (1:grep).
3. 現在時刻を shellstreaming の時刻型に適合するように正規化する (2:sed).
4. アクセスログのうち，アクセス時刻が 2014 年 1 月 1 日から 2014 年 1 月 4 日のものを保持する[2]．ただし，該当の日付以内のレコードを全て一度に下流のジョブに渡すために，ここでは Worker0 のみが上流のジョブからレコードを入力し，ジョブ実行を行う (3:ExternalTimeWindow) ．

[2] ストリーム処理は無限の入力データを扱う可能性があるが，アクセス時刻属性は時間と共に単調増加すると仮定し，ある時刻からある時刻までのデータはメモリ上に保持できるサイズのものであることを前提としたアプリケーションである．このようにレコードを有限個保持するためのウィンドウオペレータ (3.6.1 節で詳説) は他のストリーム処理系にも一般的に見られる．

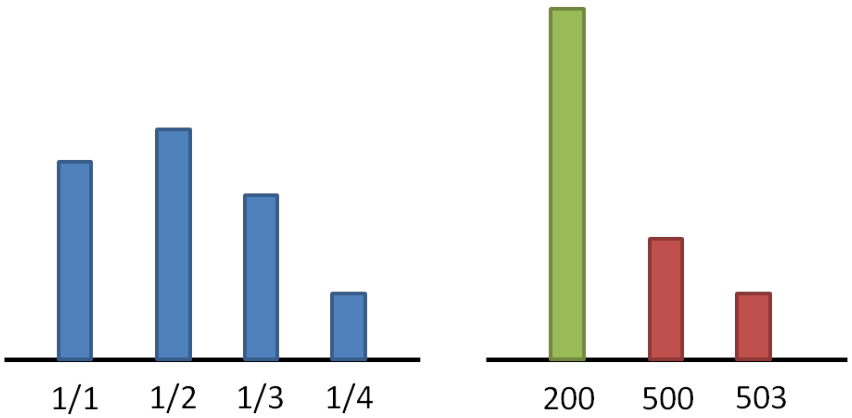

図 3.2　Apache HTTP Server アクセスログ解析アプリケーションの期待される出力 (左:日毎のアクセス数, 右:レスポンスのステータスコード種別回数)

5. WebServer0-2, Worker0-1 が，上流ジョブからのレコードをコピーし，2 つのパスに流す (4:CopySplit).
6. 左側のパスでは，アクセス時刻から日付情報だけを絞り込む (5:awk).
7. アクセス時刻順にソートする．これは，通常のシェル操作の “sort | uniq -c” イディオムと同様に，下流の *uniq -c* ジョブでそれぞれの日付の出現回数を正しく出力するための操作である (6:sort).
8. それぞれの日付の出現回数を出力する (7:uniq -c)
9. 各日付の出現回数をテキストファイルに保存する．ただし，1 つのテキストファイルに全ての出力レコードを集約するため，このジョブは Worker0 のみが行う (8.LocalText).
10. 右側のパスでも 6-9 と同様に，各ステータスコードごとの出現回数を Worker1 のテキストファイルに出力する．

shellstreaming を使わず，シェルコマンドのみを用いてこの処理を実現するのは困難を伴う．0 番のジョブを WebServer0-2 で，1,2 番のジョブを WebServer0-2 と Worker0-1 で，3 番のジョブを Worker0 で，… というように，各ジョブごとに並列度を調整することは，shellstreaming では fixed_to 指定によって容易にできる (3.8 節) 一方で，シェルスクリプトなどだけで実現するのは難しい．更に，上流ジョブの出力結果を，ジョブを並列に動作させるノード間で排他的に取り合う実装も容易ではない．そして，3 番のジョブのウィンドウのような操作は，ストリーム処理系は標準的にサポートするものだが，自らの手で実現するのは手間である．

また，他のストリーム処理系の使用を考えた場合にも，2.1, 2.2 節で示したように Storm しかシェルコマンドを明確にサポートしているものはない上，Storm のシェルコマンド対応は shellstreaming に比べると不完全である．shellstreaming には，各シェルコマンドの出力するテキストを正規表現を用いてレコード化する機能 (3.9.2 節) があるため，各シェルコマンド間の接続も容易である．

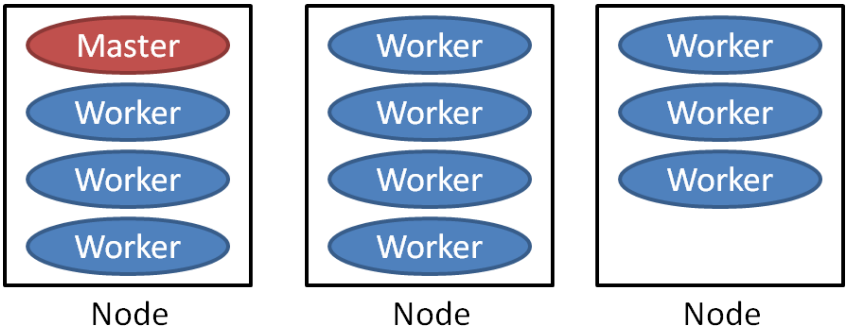

図 3.3　マスタとワーカの物理的配置例．1 つのノード辺り 4 つの CPU コアがあると仮定している．

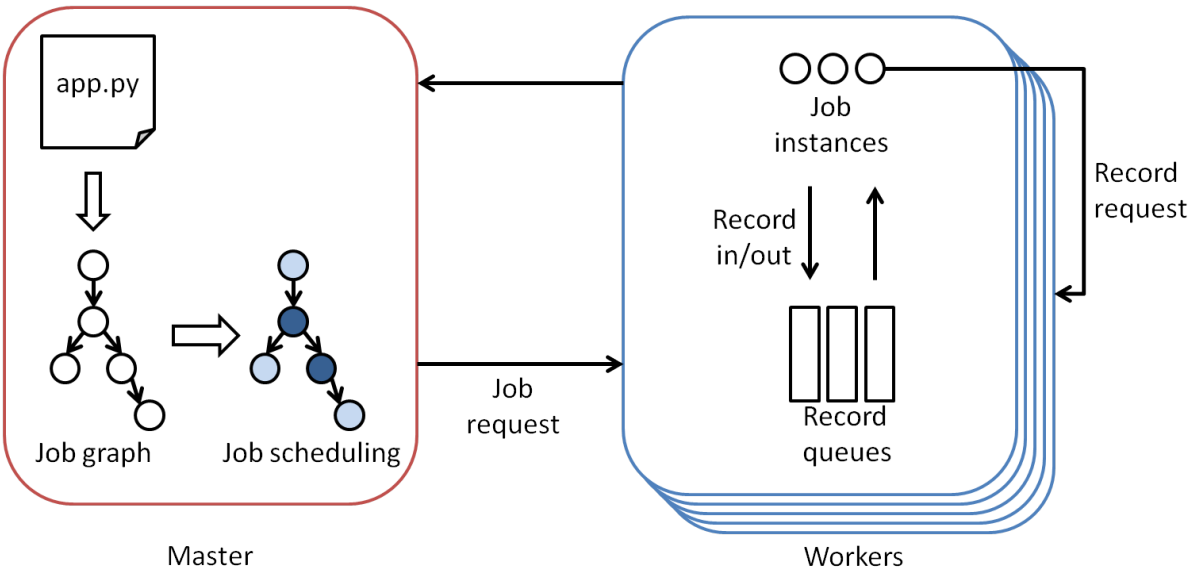

図 3.4　マスタとワーカの役割

3.2　システムアーキテクチャ

shellstreaming はマスタ・ワーカ構成をとる．マスタは実行中のシステム全体で 1 台，ワーカは 1 台以上稼働する．マスタ，ワーカの物理的な配置例を図 3.3 に示す．ただし，図では 1 つのノード辺り 4 つの CPU コアが存在するものとした．

- マスタとワーカは 1 つのノードに共存できる
- 1 つのノードは複数のワーカを持てる
- 1 つのノードにおけるワーカ数は CPU コア数を上限とするのが性能上望ましい

という点に注意されたい．

マスタとワーカの主要な役割を図 3.4 に示す．まず，マスタがアプリケーション記述からジョブグラフを生成する．マスタはジョブグラフからジョブスケジューリングを決定する．ワーカの主要な役割は，マスタからスケジュールされたジョブを実行することである．スケジュールされたジョブを実際に実行することを，ジョブをインスタンス化するという．ジョブのインスタンスは，

レコードをキューを介して入出力する．即ち，上流のジョブがキューに出力したレコードを，下流のジョブはキューから入力する．キューからレコードを入力する際，キューはワーカをまたぐものを選択することもある．マスタは各ワーカのジョブインスタンスの進行状態を監視し，全てのジョブが完了した場合にはアプリケーションを終了する．

shellstreaming のコードは全て Python で実装されている．動作時インタプリタは CPython を使用することを想定しており，この場合，各ワーカのジョブインスタンス (Python のスレッドとして実装) は GIL (Global Interpreter Lock)[3] の影響を受け，逐次動作をする．従って，各ワーカは基本的に CPU コアを 1 つ消費することを考え，ワーカの物理配置を決定するのが望ましい．ただし，シェルコマンドオペレータ (3.6.2 節) を使用する場合にはこの限りではない．シェルコマンドはワーカプロセスの子プロセスとして実行され，GIL の影響を受けない．各ワーカの動作を確実に 1 つのコアに限定するために，CPU アフィニティを設定するオプションも設けている (worker_set_cpu_affinity)．

3.3　実行系の基本構成要素: バッチ

shellstreaming は，レコードを意味上の最小単位としているが，レコードは数バイトから数百バイトとサイズの小さいものである．従って，レコードを単位に他のワーカからの入力を得たり，ジョブインスタンスを起動したりすると，その際のオーバヘッドが顕著になり性能が大きく劣化する．従って，shellstreaming ではレコードの集合であるバッチを処理の基本単位としている．

バッチのサイズはストリーム処理のアプリケーションにおいて極めて重要である．バッチのサイズが小さ過ぎれば，上述の理由によりアプリケーション全体のスループットが低下する一方で，バッチのサイズが大き過ぎれば，各レコードがジョブグラフの入力から出力に至るまでのレイテンシが増加する．バッチのサイズは入力ストリーム (3.4 節) でアプリケーション開発者による調整が可能であり，バッチサイズに応じた性能特性の変化は 5.1, 5.2 節で議論する．

アプリケーション開発者は，基本的にバッチという単位を気にすることなく，ジョブグラフのエッジにはレコード列が流れると考えることができる．ただし，ウィンドウオペレータ (3.6.1 節) を使用した場合には，ウィンドウオペレータの出力の最小単位はレコードの集合であるウィンドウになる．shellstreaming ではウィンドウはバッチとして実装されているため，ウィンドウオペレータを使用する場合にはアプリケーション開発者はバッチを意識する必要があるとも言える．

3.4　実行系の基本構成要素: 入力ストリーム

本節から 3.6 節にかけて，入力ストリーム，出力ストリーム，オペレータの構成要素を説明するが，これらを総称したものがジョブであることをここに指摘しておく．つまり，ジョブグラフのノードは，入力ストリーム，出力ストリーム，オペレータのいずれかであると言える．

入力ストリームは，外部のデータソースを shellstreaming が扱うレコードに変換するためのジョブである．入力は外部のデータソースであるため，ジョブグラフ上での入力エッジは 0 本である．また，出力エッジは 1 本に制限されている．

[3] https://wiki.python.org/moin/GlobalInterpreterLock

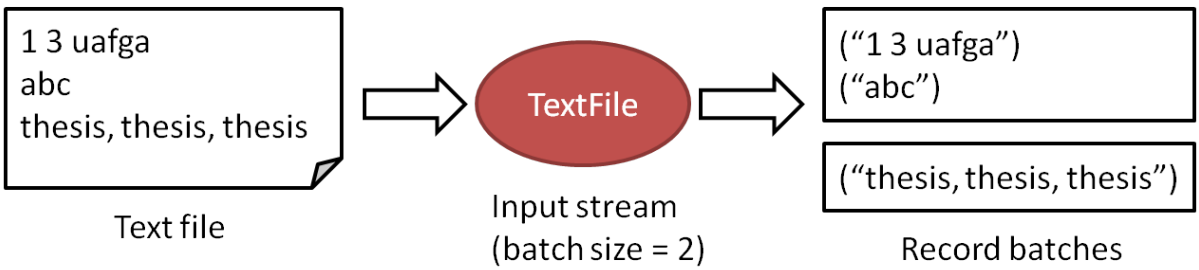

図 3.5　入力ストリームの動作例 (TextFile)

現在サポートするデータソースは，Twitter Streaming API[24] を利用したツイートデータ，テキストファイル，ランダムなテキスト列，乱数列などがある．

例として，TextFile 入力ストリーム (付録??) の動作を図 3.5 で示した．TextFile 入力ストリームが外部データソースであるテキストファイルを shellstreaming のレコードに変換している．

入力ストリームはクラス継承によって拡張することができ，HDFS[44] からの入力や JDBC[12] を経由した RDBMS からの入力も実装可能である．

3.5　実行系の基本構成要素: 出力ストリーム

出力ストリームは， shellstreaming のレコードを外部に出力するためのジョブである．入力エッジは 1 本，出力エッジは 0 本に制限されている．

現在サポートする出力先は，テキストファイルへの追記と標準出力である．これも入力ストリームと同様にクラス継承によって拡張することが可能である．

3.6　実行系の基本構成要素: オペレータ

入力ストリーム，出力ストリームのいずれでもないジョブはオペレータに分類される．オペレータは，バッチを入力とし入力バッチに応じたバッチを出力するものである．入力エッジ，出力エッジともに 1 本以上をとる．

オペレータは更に次の 4 つに分類できる．

1. 基本的なデータ操作をサポートするプリセット
2. ウィンドウ操作を行うウィンドウオペレータ
3. 複雑な処理を実現するシェルコマンドオペレータ
4. 拡張オペレータ

ストリーム処理の特徴的な操作であるウィンドウオペレータは 3.6.1 節で詳説し，shellstreaming の特徴であるシェルコマンドオペレータは 3.6.2 節で詳説する．拡張オペレータは，入力ストリームと出力ストリームと同様のクラス継承による拡張である．

ここでは，プリセットのオペレータから FilterSplit を例にあげてオペレータの動作を説明する．FilterSplit オペレータを含んだジョブグラフを図 3.6 に示す．赤色のジョブは乱数列を生成する入力ストリーム，青色のジョブはレコードをテキストファイルに出力する出力ストリームであり，

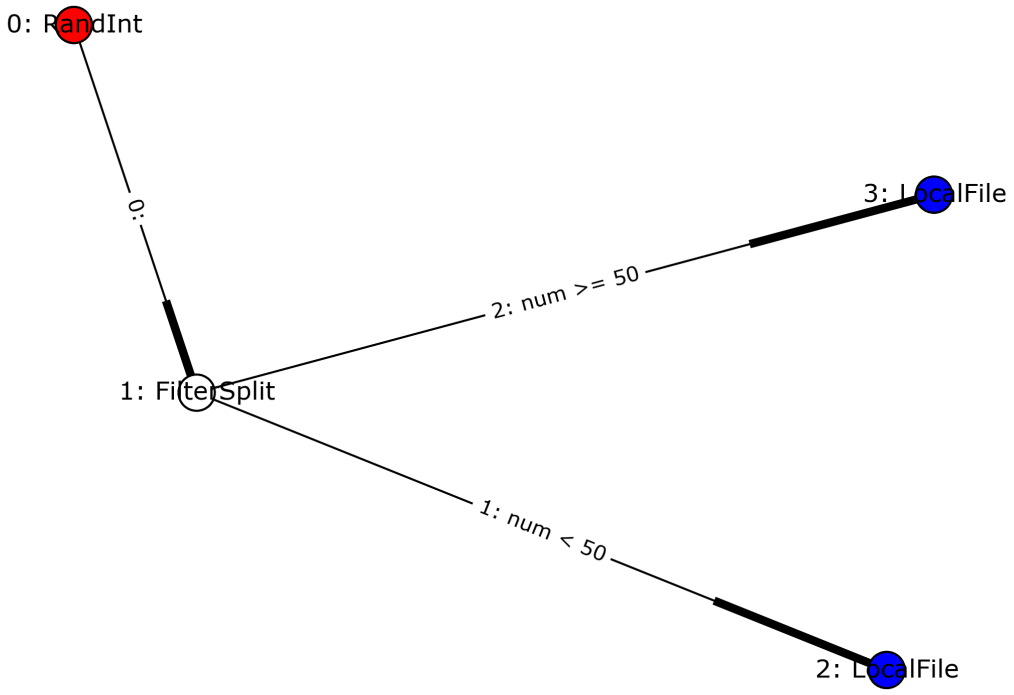

図 3.6　FilterSplit オペレータを用いた shellstreaming アプリケーション例

白色のジョブが FilterSplit オペレータである．FilterSplit オペレータは，レコード中の “num” カラムの値が 50 より小さいか否かによって，レコードを 2 つのパスに分けて出力する．図 3.6 の上側の出力オペレータに対応するテキストファイルには 50 以上の数値が，下側の出力オペレータに対応するテキストファイルには 50 より小さいの数値が書き込まれる．

3.6.1　ウィンドウオペレータ

ウィンドウは，ストリーム処理の特徴的な操作である．ストリーム処理のモデルは，基本的に各ジョブに無限のレコード列が入力され，レコードごとにジョブが駆動し出力を出すというものである．しかし，ジョブによっては無限のレコード列をとることができないものもある．ソートをするジョブがそのであって，一度ソート対象となる有限個のレコードを蓄積しなければソートの仕事を完結することはできない．

このような場合に使用するのがウィンドウオペレータである．ウィンドウオペレータは，無限のレコード列であるストリームを，有限のレコード列の連続に区切る動作をする．ウィンドウオペレータの出力である有限のレコード列をウィンドウと呼ぶ．

ウィンドウオペレータは，ウィンドウのサイズの決定法と，ウィンドウからレコードを排除するポリシーにより分類することができる．

ウィンドウサイズの決定法には，代表的な 2 つの方法がある．

1. ウィンドウ内のレコード数 (Data-based window)
2. レコードに含まれるタイムスタンプの幅 (Time-based window)

前者の Data-based window では，例えばウィンドウをレコード 10 個の幅に指定し，後者の Time-based window ではウィンドウを直近 10 分間の幅に設定する．

ウィンドウ中のレコードの排除ポリシーも，レコード数によるものと時間によるものがある．排除ポリシーとは，そのポリシーを満足した時に，ウィンドウオペレータに蓄積されたレコードのうち古いものから削除するためにある．例えば，サイズ 3 の Data-based window オペレータが排除ポリシーをレコード 2 つとしていた場合を考える．今，オペレータが古い順に (1, 2, 3) という 3 つのレコードを持っていた場合に，(4, 5) という 2 つのレコードを入力することを考える．排除ポリシーがレコード 2 つなので，4 が入力された時点でオペレータは何も出力しないが，5 が入力されるとオペレータは古い 2 つのレコード 1 と 2 を削除し，その後に (3, 4, 5) というレコードを蓄積する．この排除のタイミングでは以前の蓄積レコードを出力するので，この場合 5 が入力された時点でウィンドウオペレータは (1, 2, 3) という出力を出す．

ウィンドウサイズの決定法と排除ポリシーで使う軸は一致する必要はなく，例えばウィンドウサイズはレコード数により決定し，排除ポリシーは時間で指定することもウィンドウオペレータの枠組みの中で可能である．

特に，ウィンドウサイズと排除ポリシーが完全に一致しているものを Tumbling window と呼び，ウィンドウサイズと排除ポリシーの軸は一致していて，排除ポリシーのほうが小さいものを Sliding window と呼ぶ場合もある．

shellstreaming ではウィンドウオペレータとして CountWindow オペレータと ExternalTimeWindow オペレータを持つ．前者は Data-based window オペレータであり，排除ポリシーとしてレコード数を指定できる．後者は Time-based window オペレータであり，排除ポリシーとして時間幅を指定できる．共に 1 本の入力エッジと 1 本の出力エッジをとる．

3.6.2　シェルコマンドオペレータ

シェルコマンドオペレータは shellstreaming を特徴付けるものである．シェルコマンドを拡張オペレータとして使用するためのものであり，以下に記載する制約を満たすシェルコマンドであればあらゆるものをオペレータとして使用できる．

- (制約 1) shellstreaming のレコードを変換したテキストデータを 1 つのファイルまたは標準入力から受け取る
- (制約 2) shellstreaming のレコードへ変換されるべきテキスト出力を 1 つのファイルまたは標準出力へ出力する
- (制約 3) 入力テキストデータから EOF (End Of File) を受け取った時点でシェルコマンドが終了する

これらの制約は，ストリーム処理に組み込むことを検討するシェルコマンドであれば多くのものが自然に満たすと考える．

シェルオペレータは入力エッジと出力エッジを 1 本ずつ持ち，コード 3.1 に示すような記述法により shellstreaming のレコード (正確にはバッチ) をシェルコマンドとやりとりする．shellstreaming が特別に解釈するキーワードは “IN_STREAM”, “OUT_STREAM”, “<”, “>” の 4 つである．

IN_STREAM api.Operator 関数の第 1 引数に与えられるストリーム中の入力バッチを，シェルコマンドに渡す方法を指示する．“<” の直後に置かれた場合，入力バッチは標準入力から与えられる．そうでない場合，IN_STREAM は一時ファイルのパスへと書き換えられ，その一時ファイルの中に入力バッチの内容が書き込まれる．

OUT_STREAM api.Operator 関数の返り値に与えられるストリーム中の出力バッチを，シェルコマンドから受け取る方法を指示する．“>” の直後に置かれた場合，出力バッチは標準出力から出たコマンドの出力から生成される．そうでない場合，OUT_STREAM は一時ファイルのパスへと書き換えられ，コマンドは通常の動作通りにそのファイルにテキスト出力を書き込み，それを shellstreaming がバッチに変換する．

コード 3.1　シェルコマンドオペレータのシェルコマンド記法．ただし，api.Operator 関数の引数は一部省略．

```
# the `command` has `-i` option to specify input file path, `-o` option output file path
stream1 = api.Operator(
    [stream0], ShellCmd,
    'command  < IN_STREAM  > OUT_STREAM', ...)  # batch from stdin, to stdout
stream1 = api.Operator(
    [stream0], ShellCmd,
    'command -i IN_STREAM  > OUT_STREAM', ...)  # batch from file,  to stdout
stream1 = api.Operator(
    [stream0], ShellCmd,
    'command  < IN_STREAM -o OUT_STREAM', ...)  # batch from stdin, to file
stream1 = api.Operator(
    [stream0], ShellCmd,
    'command -i IN_STREAM -o OUT_STREAM', ...)  # batch from file,  to file
```

コマンドの扱うテキスト入出力と，shellstreaming の扱うバッチの相互変換については 3.9.2 節に記載している．

シェルコマンドは，1 つの入力バッチにつき 1 度起動され，1 つの出力バッチを出力し終了することを基本とする (図 3.7)．シェルコマンドオペレータは入力エッジからバッチの列を受け取るため，シェルコマンドを繰り返し起動することで，シェルコマンドオペレータはストリーム処理のオペレータとして動作する．レコードがシェルコマンド起動の単位であると，プロセスの起動によるオーバヘッドがストリーム処理全体の性能低下を招く可能性が高いため，バッチをシェルコマンド起動の単位としている．一部のシェルコマンドはバッチ単位でも起動時のオーバヘッドが大きいため，そのようなシェルコマンドのデーモン化の手法も考案し，4.1 節で説明している．

3.7　実行系の基本構成要素: キュー

キューはジョブグラフのエッジで結ばれたジョブ同士がバッチをやりとりするための機構である．概念的には，上流のジョブが出力するバッチをキューに入れ，下流のジョブが入力するバッチをそのキューから取り出すもので，ジョブグラフ上のエッジがキューに対応すると言える．

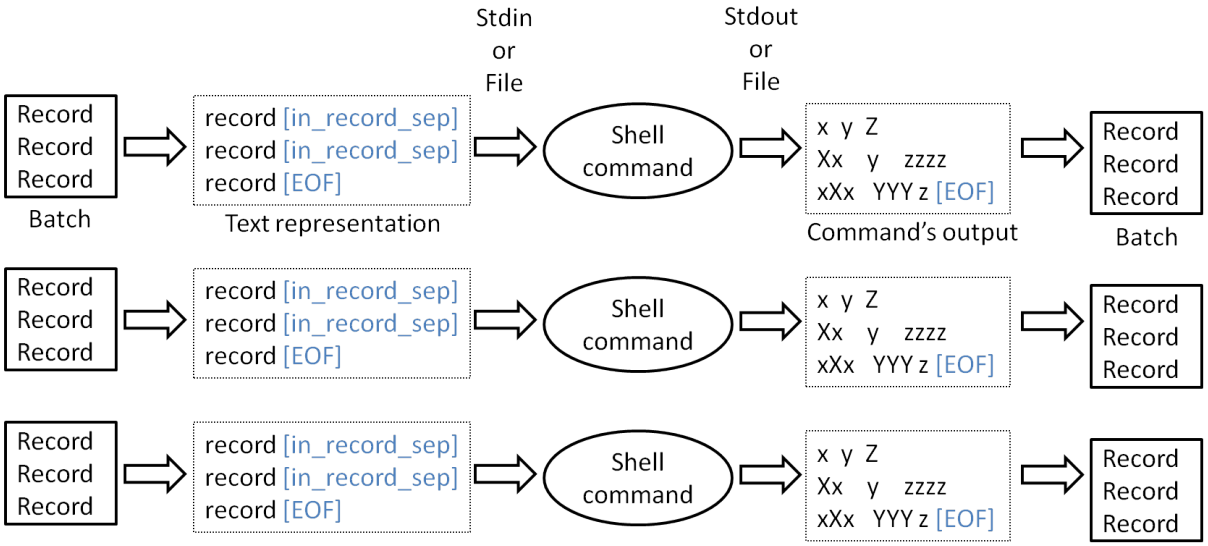

図 3.7　shellstreaming プロセスとシェルコマンドプロセスのデータの受け渡し

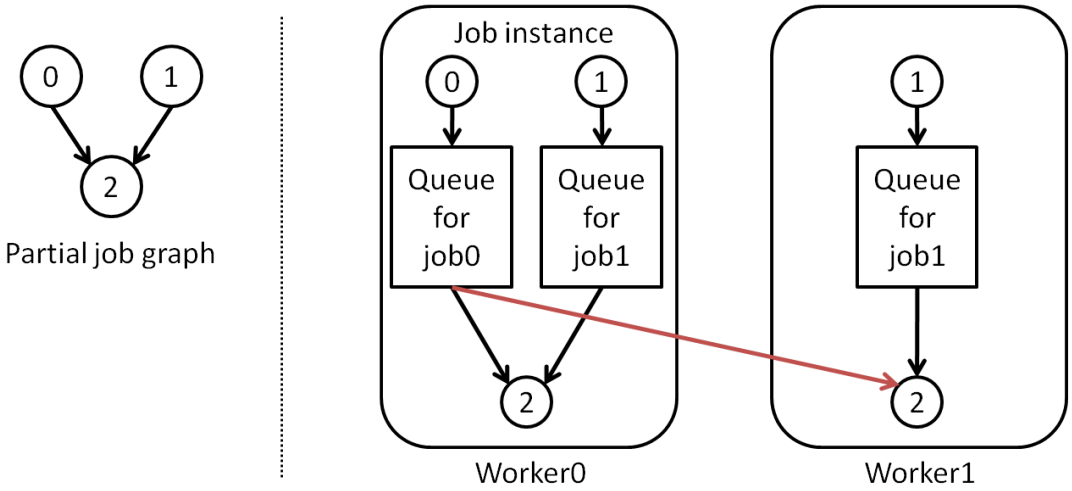

図 3.8　キューの説明図．Worker1 のジョブインスタンス 2 は Worker0 のジョブインスタンス 0 のキューから入力バッチを得ている．

キューは実際には，ワーカ内のジョブインスタンスにつき 1 つ作成される．図 3.8 にその様子を示す．Worker0 はジョブインスタンス 0, 1, 2 を実行しているため，それぞれに対応したキューを合計 3 つ持っている (ただし，ジョブインスタンス 2 のキューは図では省略されている)．Worker1 はジョブ 0 のインスタンスを持たないため，それに対応したキューも持たない．ジョブインスタンスがキューからバッチを入力する際には，入力エッジに対応するキューのいずれかからバッチを 1 つ取り出す．ワーカをまたいでキューを選択すると通信が発生する[4]．

各キューは FIFO (First-In, First-Out) である．

キューのエントリは基本的にはバッチであるが，上流ジョブインスタンスの終了通知もキュー

[4]現在の実装では，同一ノード内に存在するワーカ同士であっても TCP プロトコルによる通信が発生する．

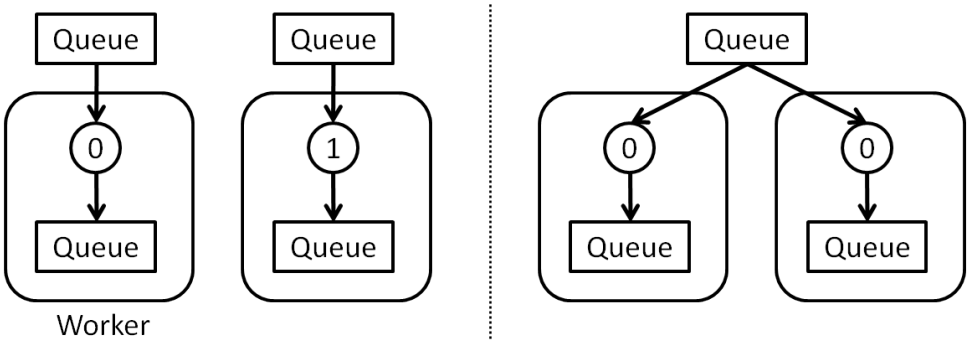

図 3.9　ジョブの分散実行方式．左は異なるジョブのインスタンスを異なるワーカに配置する方式，右は同一のジョブのインスタンスを異なるワーカに配置する方式．

に入る．例えば，入力ストリームのインスタンスがデータソースから入力を終えた場合には，そのインスタンスに対応したキューに終了通知を push する．下流ジョブのインスタンスは終了通知を pop した場合に上流ジョブの終了を認識し，インスタンスを終了することができる．細かくは，上流ジョブの全てのジョブインスタンスが終了するまで下流のジョブはインスタンスを立て続ける必要があるのだが，やや説明が煩雑になるため本稿ではその詳細には触れない．

3.8　ジョブ実行モデル

ジョブグラフ上に存在する各ジョブは，ワーカにおいてインスタンス化されて実行される．

本節では，ジョブインスタンスの配置によるジョブの分散実行について述べる．ジョブの分散実行には，図 3.9 に表す 2 つの方式がある．即ち，異なるジョブのインスタンスを異なるワーカに配置する方式と，同一のジョブのインスタンスを異なるワーカに配置する方式である．図ではいずれの入力キューも各ワーカの外側に配置しているが，実際にはワーカ内に存在するキューであることも別のワーカに存在するキューであることもあり得る．また，図の左側の方式では，例えばジョブ 0 のインスタンスの出力がジョブ 1 のインスタンスの入力であることもあり得る．

どのワーカでどのジョブをインスタンス化するかを決定するのはマスタのジョブスケジューラである．アプリケーション開発者が何も指定しない限り，ジョブはどのワーカでもインスタンス化され得る．一部のジョブでは，インスタンスを起動するワーカを固定したい要望がある．具体的には次のような場合が考えられる．

- あるワーカが存在するノードに，外部データソースとして使用したいファイルがある場合
- 使用するワーカ数全てで外部データソースにアクセスすることができない場合 (例: Twitter Streaming の API 制限)
- 内部状態を持つジョブを 1 つのワーカのみで起動したい場合 (例: ウィンドウオペレータは現在のウィンドウという状態を持つ)

この場合には，ジョブを記述する関数 (“api.IStream”, “api.OStream”, “api.Operator”) の引数に “fixed_to=[’ ワーカ名’, . . .]” という引数を渡すことでジョブを起動するワーカを固定することができる．

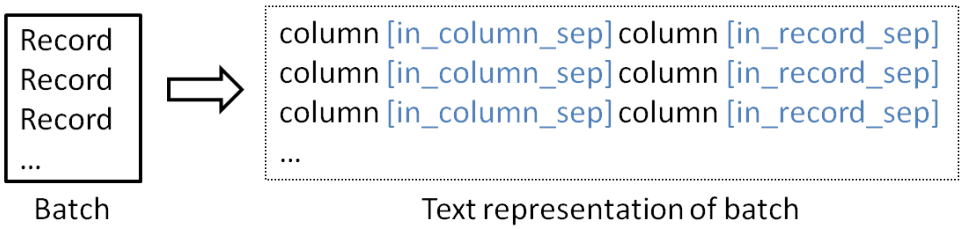

図 3.10　バッチからシェルコマンド入力文字列への変換．説明の都合上，レコードのセパレータの後に改行を入れている．

3.9　データモデル

shellstreaming のレコードは，型付きのカラムデータの集合になっていて，RDBMS で使用されるレコードと使用上変わらない概念となっている．一部のオペレータは，レコード全体でなく特定のカラムの値のみを参照して処理を行う．3.6 節の説明に使用した FilterSplit オペレータもその一つで，条件分岐には 1 つのカラムの値を使用する．

3.9.1　データ型

shellstreaming が使用できるデータ型について説明する．今後，更に多くのデータ型をサポートする予定である．

INT　符号付き整数値型．

TIMESTAMP　時刻型．”%Y-%m-%d %H:%M:%S” というフォーマットの文字列 (例: “2014-02-06 15:00:00”) を時刻型として使用できる．

STRING　文字列型．実際には任意のバイト列を文字列型として使用できる．

3.9.2　バッチとシェルコマンド入出力文字列の相互変換

shellstreaming のバッチをシェルコマンドの入力とする際，バッチをプレーンテキストに変換し，シェルコマンドの標準入力やファイルを介して入力する．また同様に，シェルコマンドの出力を shellstreaming のバッチにする場合，シェルコマンドの出力であるプレーンテキストをシェルコマンドの標準出力やファイルを介して取得し，それをバッチに変換する．本節では，この相互変換の方法について詳説する．

バッチからシェルコマンド入力文字列への変換

バッチからシェルコマンド入力文字列への変換は図 3.10 で表される．まず，各カラムの文字列への変換は，各カラムの型の文字列変換方式に従う．そして，各カラムの間に “in_column_sep” というユーザが指定可能な文字列が挿入され，レコードの文字列表現となる．レコード間にも “in_record_sep” という区切り文字が挿入される．4.1 節で説明するシェルコマンドのデーモン化を行わない限り，バッチの最後のレコードの文字列には EOF が付加される．

シェルコマンド出力文字列からバッチへの変換

シェルコマンド出力文字列からバッチへの変換は，その逆よりも複雑である．シェルコマンドの出力文字列は定型的な構造をとっているとは限らないため，定形データであるレコードへの変換は単純な問題ではない．

shellstreaming では，正規表現マッチング [46] を用いて非定形の文字列を定形のレコードに変換することをサポートする．

3.1 節で登場した Apache HTTP Server のアクセスログ解析アプリケーションの 1 つ目のシェルオペレータを例に，変換方法を説明する (コード 3.2)．

コード 3.2　Apache HTTP Server アクセスログ解析アプリケーションの 1 つ目のオペレータ (説明に不要な部分を一部省略)

```
1  # filter lines in which '/' is 'GET' accessed
2  access_stream = api.Operator(
3      [log_stream], ShellCmd,
4      r'''grep -E '"GET / HTTP/[.0-9]+"' < IN_STREAM > OUT_STREAM''',
5      out_record_def=api.RecordDef([
6          {'name': 'ipaddr'     , 'type': 'STRING'},
7          {'name': 'timestamp'  , 'type': 'STRING'},
8          {'name': 'statuscode' , 'type': 'INT'},
9      ]),
10     # output line example:
11     # 151.217.31.218 - - [30/12/2013:10:29:50 +0900] "GET / HTTP/1.1" 200 265 "-" "-"
12     out_col_patterns={
13         'ipaddr'     : re.compile(r'^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+'),
14         'timestamp'  : re.compile(r'(?<= - - \[).+(?=\] )'),
15         'statuscode' : re.compile(r'[0-9]{3}'),
16     },
17 )
```

このシェルオペレータは grep コマンドを起動し，grep コマンドは出力として ‘151.217.31.218 - - [30/12/2013:10:29:50 +0900] “GET / HTTP/1.1” 200 265 “-” “-”’ のようなフォーマットの行を複数出す．この出力から，(ipaddr, timestamp, statuscode) という形式のレコード群を作成することを，“out_record_def” 引数によって指定している (7-10 行目)．各 3 つのカラムが grep の出力のどの部分に対応するかは，”out_col_patterns” の 3 つの正規表現によって示している．“out_col_patterns” で指定した全てのカラムを取り終わった時点で出力レコードが生成され，その時点で指しているシェルコマンドの出力テキストのカーソルから再び次のレコードの生成が始まる．

正規表現の表現力には限界があり，任意のシェルコマンドの出力を定型的なカラムで構成されるレコード群に変換することはできない．しかしそのような場合においても，あるバッチ入力に対する全ての出力文字列を STRING 型のカラムとして持つ 1 つのレコードを出力バッチとすることは可能である．

3.10　ストリーム処理の記述方法

本節では，shellstreaming のアプリケーション開発におけるストリーム処理の記述法について述べる．ストリーム処理の記述言語は，ストリーム処理言語, ルール指向言語, エージェント指向言語の 3 種に分類される [38, 49]．shellstreaming の記述言語は，このうちエージェント指向言語

に分類できる．シェルコマンドを用いたストリーム処理を記述する開発者が生産性を最大限発揮できるための言語として，実際の処理系への採用事例のあまり多くないルール指向言語は考慮から外し，ストリーム処理言語とエージェント指向言語から後者を選んだ．ここではまず，ストリーム処理言語とエージェント指向言語のそれぞれの特徴について説明する．

ストリーム処理言語は，RDBMS の多くが採用しているデータ処理言語である SQL に類似した宣言的な言語である．SQL と同様に，ストリーム処理言語によるデータへの問い合わせ処理のことをクエリと呼ぶ．クエリはストリームデータの到着に合わせて継続して実行されるため，一般的に Continuous Query [29] と呼ばれる．Continuous Query を記述する具体的な言語，または言語を持つシステムとしては，CQL[28, 29], TelegraphCQ[31], NiagaraCQ[33], Esper[7] の EPL, Oracle/CEP[15], StreamInsight[14] などが挙げられる．ストリーム処理言語の持つ宣言的な性質は，ストリーム処理を高生産に定義できるものだと言われる．特に，RDBMS を基盤とし，SQL で大規模なデータを処理するアプリケーションを記述する経験を詰んだ開発者にとっては習熟しやすいものと考えられる．一方で，shellstreaming がバッチデータをシェルコマンドの入出力に相互変換する際には，アプリケーション開発者が変換の方法を一部指示する必要があり (3.9.2 節)，これは SQL のような宣言的な言語とは親和性が低いと考える．

エージェント指向言語は，手続き的にジョブ間のデータフローを記述し，ジョブグラフを構成するものである．ストリーム処理言語が SQL に似通ったものなのに対し，エージェント指向言語は処理系によって様々な形式がある．InfoSphere[11] の SPL[16] は InfoSphereDSL でデータフローを記述し，Storm では Java や Scala , Fluentd では XML というように，各々の処理系に特色がある．shellstreaming では Python でジョブグラフを記述する．エージェント指向言語は，宣言的な言語とと比べて，同じ処理を定義する際の記述量は増える傾向にあると言われる．一方で，宣言的な言語と比較した際に，手続き的な言語は習熟度が低くても記述・解釈しやすい．SQL と手続き的なスクリプト言語を比べれば，手続き的な言語の上から読み下してデータフローを追うことのできる性質の利点が理解できるだろう[5]．更に，関数呼び出しによってオペレータを定義する方法は，バッチデータをシェルコマンドの入出力と相互変換する際の変換方法の指示も自然に記述できる．shellstreaming のアプリケーションコードの断片であるコード 3.3 では，“in_column_sep”, “out_col_patterns” によって相互変換の方法を自然に指示している．

コード 3.3　オペレータ定義関数の引数にバッチデータとシェルコマンド入出力データの変換を指示する例．カラム間区切り (in_column_sep) などが，使用するシェルコマンドの付加情報として自然に指定できている．

```
output_edge = api.Operator(
    [input_edge], ShellCmd,
    'awk -F "|" "{print \$3}" < IN_STREAM > OUT_STREAM',
    in_column_sep='|',
    out_record_def=api.RecordDef([{'name': '3rd_col', 'type': 'INT'}]),
    out_col_patterns={'3rd_col': re.compile(r'^.+$')})
```

ここで，比較的単純な shellstreaming のアプリケーションコードをコード 3.4 に掲載する．これは実際に動作するコードである．

コード 3.4　FilterSplit オペレータを用いた shellstreaming アプリケーション

```
# -*- coding: utf-8 -*-
from shellstreaming import api
```

[5] Hadoop を基盤とした処理系の Pig は，まさにこの点を利点として持つ Pig Latin[42] を記述言語としている．

```
3  from shellstreaming.istream import RandInt
4  from shellstreaming.operator import FilterSplit
5  from shellstreaming.ostream import LocalFile
6
7
8  LOW_OUTPUT_FILE  = '/tmp/FilterSplit_lo.txt'
9  HIGH_OUTPUT_FILE = '/tmp/FilterSplit_hi.txt'
10
11
12 def main():
13     randint_stream       = api.IStream(RandInt, 0, 100)
14     lo_stream, hi_stream = api.Operator(randint_stream, FilterSplit,
15                                         'num < 50', 'num >= 50')
16     api.OStream(lo_stream, LocalFile, LOW_OUTPUT_FILE,
17                 output_format='json', fixed_to=['localhost'])
18     api.OStream(hi_stream, LocalFile, HIGH_OUTPUT_FILE,
19                 output_format='json', fixed_to=['localhost'])
```

shellstreaming は，main 関数を実行することでジョブグラフを生成する．api.IStream, api.OStream, api.Operator は，それぞれ入力ストリーム，出力ストリーム，オペレータを宣言するための関数で，その引数に具体的なジョブの種類や実行条件を記述する．この例では，

1. 入力ストリームとして RandInt で乱数列を生成
2. FilterSplit オペレータでレコード中の乱数の値に応じてレコードを 2 つのパスに分配
3. LocalText で 2 つのテキストファイルに分けてレコードを保存

というジョブグラフを記述している．api.OStream と api.Operator が第一引数に入力エッジに相当するストリームを取っていること，api.IStream と api.Operator が返り値に出力エッジに相当するストリームを返していることに注意されたい．shellstreaming のアプリケーション開発はジョブグラフを記述することとも言える．

shellstreaming は無限の入力列も有限の入力列も扱える．この例では，RandInt 入力ストリームが無限の乱数列をシステムに入力する．もし api.Istream のキーワード引数に “max_records” を整数値で設定すれば，入力ストリームが生成するレコード数を制限することができる．

3.11　アプリケーション開発支援機能

ここでは，shellstreaming がストリーム処理アプリケーションの開発や実行サイクルを支援するために備えている機能をいくつか紹介する．

3.11.1　自動デプロイ機能

ストリーム処理系に限らず，多くのマスタ・ワーカ構成を取る処理系では，(1) ワーカプロセスを起動し，(2) その後にマスタプロセスを起動して，(3) ワーカプロセスに接続する という工程を手動で行う必要がある．特に，ワーカプロセスを物理的に離れたノードで実行する場合，ワーカプロセスの起動の手間は大きい．

shellstreaming は，ワーカを起動するノードのホスト名とワーカサーバが使用する TCP ポート番号を設定ファイルに記述しておくだけで，マスタ プロセス起動時に自動的にワーカプロセスが起動される．この機能を自動デプロイ機能と呼んでいる．

自動デプロイ機能の利点は，単にワーカを起動するノードにログインし起動手順を踏む手間を省くだけではない．各ワーカへのデプロイは並列で行われるため，特にワーカの数が多い時にはストリーム処理を開始するまでのオペレーション時間が大きく削減できる．また，マスタを起動するノードで shellstreaming 自体のソースコードを編集した場合には，最新のコードがワーカノードに転送されて，そのコードでワーカが起動されるので，shellstreaming 自体のデバッグや機能拡張に大いに役立つ．

自動デプロイ機能は次のような具体的な手順の処理を踏むことで実現されている．

1. マスタを起動し，マスタが自動デプロイ機能を開始する
2. マスタは shellstreaming 自らのソースコード群を圧縮する
3. マスタが設定ファイルを読み，ワーカプロセスを起動すべきノード (以下，ワーカノード) のホスト名，ポート番号を認識する
4. ワーカノードに shellstreaming のソースコード群を転送する．この際 ssh プロトコルが使用される．
5. マスタが，ワーカノードに，ワーカプロセスを起動するためのコマンドを転送し，ワーカはそれを実行する．ここでも ssh プロトコルが使用される．
6. ワーカノードはワーカプロセスを起動し，ワーカプロセスは設定ファイルに記述されたポート番号を Listen する．このポート番号は，マスタ-ワーカ間の通信にもワーカ-ワーカ間の通信にも用いられる．
7. マスタが全てのワーカとの接続を確認し，ストリーム処理を開始する．

以上から分かるように，マスタノードとワーカノード間は ssh による接続ができることを要求する．ただし，ワーカノード同士の接続は設定ファイルに記述された TCP ポートを使用するので，ssh 接続は必要ない．

この機能は，Python の fabric [8] ライブラリを用いて実現した．

3.11.2　テストケース実行機能

プログラムを記述する際，その動作の正しさを確認するためにテストケースが有用であることは言うまでもない．テストケースの利点はそれだけでない．プログラムがどのように動作する想定で記述されたかを，開発者以外に伝える点も見逃せない．

ストリーム処理をアプリケーションにとってもテストケースは有用であると考え，shellstreaming にはテストケースを実行する機能を設けた．コード 3.5 はストリーム処理記述である main 関数はコード 3.4 のものと同様であるが，test 関数にテストケースが実装されている．

コード 3.5　テストケースが付随した shellstreaming アプリケーション

```
1 # -*- coding: utf-8 -*-
2 from shellstreaming import api
3 from shellstreaming.istream import RandInt
4 from shellstreaming.operator import FilterSplit
5 from shellstreaming.ostream import LocalFile
6
7
8 LOW_OUTPUT_FILE  = '/tmp/FilterSplit_lo.txt'
9 HIGH_OUTPUT_FILE = '/tmp/FilterSplit_hi.txt'
```

```
10
11
12  def main():
13      randint_stream        = api.IStream(RandInt, 0, 100)
14      lo_stream, hi_stream = api.Operator(randint_stream, FilterSplit,
15                                          'num < 50', 'num >= 50')
16      api.OStream(lo_stream, LocalFile, LOW_OUTPUT_FILE,
17                  output_format='json', fixed_to=['localhost'])
18      api.OStream(hi_stream, LocalFile, HIGH_OUTPUT_FILE,
19                  output_format='json', fixed_to=['localhost'])
20
21
22  def test():
23      import json
24      # low records
25      with open(LOW_OUTPUT_FILE) as f:
26          for lo, line in enumerate(f):
27              record = json.loads(line)
28              # testing if numbers are between 0 and 49
29              assert(0 <= int(record['num']) < 50)
30      # high records
31      with open(HIGH_OUTPUT_FILE) as f:
32          for hi, line in enumerate(f):
33              record = json.loads(line)
34              # testing if numbers are between 50 and 100
35              assert(50 <= int(record['num']) <= 100)
36      # testing total number of records is same that of input random integers
37      assert((lo + 1) + (hi + 1) == NUM_RECORDS)
```

テストケースだけを見ても，このストリーム処理アプリケーションは 2 つの出力ファイルを持ち，片方のファイルには 0 から 49 の整数値が，もう片方のファイルには 50 から 100 の整数値が書き込まれていて，合わせて 10000 行の出力を持つということが確認できる．

test() 関数に記述されたコードは，ストリーム処理終了時に通常の Python インタプリタによって実行され，テストケースが異常終了した場合には shellstreaming がその旨を通知する．テストケースが実行されるのはマスタノードであることに注意されたい．従って，処理結果の出力先がマスタノード内に存在しない場合は，テストケース記述が複雑になってしまう．

3.11.3　ジョブグラフの可視化

3.10 節において，shellstreaming のアプリケーション記述はジョブグラフを記述することであると述べた．shellstreaming には，アプリケーションコードから生成されるジョブグラフを可視化し画像ファイルとして出力する機能がある．例えば，図 3.6 はこの機能によって出力されたジョブグラフである．この機能は，アプリケーション開発者が自らの記述した Python コードが意図するジョブグラフを形成しているかを確認したり，既成の shellstreaming アプリケーションを入手した者がアプリケーションの構造を視覚的に把握するのに有用である．

設定ファイルに “job_graph_path = /path/to/jobgraph.png” という記述をすることで，アプリケーションコードを解釈した時点で指定したパスにジョブグラフの画像が出力される．

3.11.4　マスタ・ワーカの単一プロセス実行によるデバッグモード

shellstreaming のソースコード自体を改変するとき，たとえ自動デプロイ機能を使用しても，改変の効果を確認するのに手間がかかる．また，自動デプロイは ssh を用いてコードを転送する性

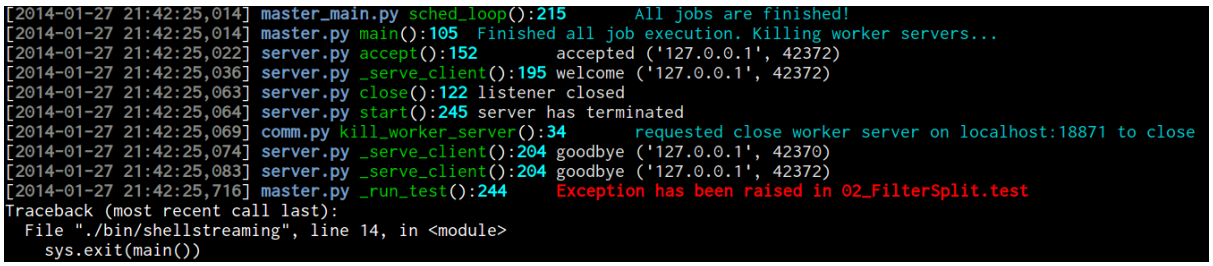

図 3.11　shellstreaming の色付きログ出力

質上，多少の時間 (30 秒程度) を要する．また，マスタプロセスは実行時のログを標準エラー出力に出すのに対し，ワーカプロセスはログを指定されたファイルに出力するが，shellstreaming のログは標準ストリームへの出力の時のみ色付けされる[6]．色のついたログは図 3.11 をご確認いただきたい．

そこで，shellstreaming には，マスタとワーカを単一プロセスで実行するデバッグモードを実装した．これを使うと，マスタプロセスを立てる際，1 つのワーカをマスタプロセスのスレッドとして立ち上げることができる．マスタとワーカの通信は通常通りのプロトコルで行われるため，デバッグモードの動作は通常時の動作と大きく変わることはない．しかし，マスタとワーカが同一プロセスで起動することで，実行ログの確認やプロファイリングが容易になる．

デバッグモードを有効にするには，設定ファイルに “localhost_debug = yes” の一行を記述する．

[6]拙作の Python ライブラリである RainbowLoggingHandler [17] が使用されている．

第4章　高速化手法

本章では，shellstreaming のアプリケーション実行を高速化するための手法を紹介する．特に，4.1 節で詳説するシェルコマンドのデーモン化は，我々の知る限り他の処理系に見られない特徴である．

4.1　シェルコマンドのデーモン化

4.1.1　デーモン化の定義・目的

shellstreaming は，ストリーム処理アプリケーションが組み込むオペレータにシェルコマンドを使用することを全面的に支援する．しかし，シェルコマンドをストリーム処理に組み込む際の性能上の問題として，一部のシェルコマンドは起動時間が長くなることが挙げられる．これは，シェルコマンドを fork(2) すること自体がある程度時間の掛かる処理であることの他，シェルコマンドはコマンド自身で処理を完結させる必要があることが原因となる．ストリーム処理システムと同一の言語処理基盤を用いて実行されるジョブであれば，ジョブの実行以前のどこかのコードパスにおいてジョブが必要とする初期化処理などを済ませておけばよいが，プロセスをまたいで起動する必要があるシェルコマンドのようなジョブでは，初期化処理はジョブ自身が行うことが現実的な選択となる．

この初期化処理に，ストリーム処理全体の遅延に影響するほど時間の掛かるシェルコマンドが存在していて，実際に英文を解析するストリーム処理において有用な Enju[40, 6] というシェルコマンドでは，2.40GHz の動作周波数の CPU を用いて約 8 秒の起動時間を要する．また，1.2 節に挙げた shellstreaming の応用先として検討しているアプリケーションにおいても，起動時間が無視できないシェルコマンドが含まれる．

このようなシェルコマンドを，3.6.2 節で説明したようにバッチ入力毎に繰り返し起動していては，ストリーム処理の性能が極めて劣化する．

そこで我々は，シェルコマンドをデーモン化する機能をシェルコマンドオペレータに実装した．ここでデーモン化とは，シェルコマンド入出力可能な状態でを shellstreaming が指定するタイミングまで起動し続けることを指す．

4.1.2　デーモン化可能なシェルコマンドが満たすべき制約

シェルコマンドをデーモン化するためには，3.6.2 節で示した 3 つの制約の他に，シェルコマンドは次の制約を満たす必要がある．

- (制約 A) 標準入力からテキスト入力を受け付ける．

- (制約 B) 入力テキストに特定の識別文字列が含まれている場合，その識別文字列以前の入力テキストに対応した出力テキストを，標準出力から出す．
- (制約 C) 特定の入力テキストに対しては，出力テキストが予測可能である．

POSIX コマンドの cat を例にして，この制約を具体的に説明する．まず cat は制約 A を満している．また，cat は入力テキストに改行文字 (\n) が含まれる場合には，改行文字以前の入力テキストと改行文字を標準出力から出力する (制約 B)．更に，cat に例えば “Hello, world!!\n” と入力した場合には，出力にも “Hello, world!!\n” というテキストデータが返答される (制約 C)．

以上の 3 つの制約，並びに 3.6.2 節で示した 3 つの制約は，文字列処理に利用される POSIX コマンドの多くが満たすものである．また，通常の作成されるシェルコマンドであっても，ストリーム処理に組み込むことを検討するようなものであれば，これらの制約は多くの場合自然に満たされていると考えられる．制約 A,B に関しては，入力レコードを低遅延で出力レコードにするシェルコマンドを記述する際に最も自然な実装方法であると言える．制約 C に関しては，シェルコマンドが入力テキストのみに依存して出力テキストを返すようなもの (ステートレス，副作用のないプログラム) であれば，予測可能である．しかし，ストリーム処理に組み込むことを検討するようなシェルコマンドであっても，この制約を満たさない場合はある．例えば，5.3.1 節でも使用しているワードカウントのシェルコマンドもステートフルなものの代表である．これは，標準入力から単語を受け取り,「その単語が今までに何回入力されたか」というステートを元にした出力を標準出力から返すコマンドである．このようなシェルコマンドをデーモン化するためには，特定の入力テキストに対して特定の出力テキストを出力する機能を加えるのが最も簡単な対処である．例えば，5.3.1 節で使用しているワードカウントのコマンドは，標準入力からスペースを含むテキストを入力すると，”single word is expected” というテキストが標準出力から出されるようにできている．詳しくは付録 B.2 を参照されたい．

4.1.3　デーモン化手法

3.6.2 節では，シェルコマンドをデーモン化しない場合に，いかにしてシェルコマンドと shell-streaming でデータ入出力をやりとりするかを説明した．本節では，デーモン化した場合のデータ入出力の扱いについて説明する．

3.6.2 節と比べると，デーモン化をする場合には，シェルコマンドの入力元が標準入力に，出力先が標準出力に限定されるため，シェルコマンドが入出力にファイルを利用するケースは考慮から外れる．ここでは，デーモン化したシェルコマンドの標準入出力を介したバッチの入出力の方法を述べる．

シェルコマンドに標準入力から入力テキストを与える際，EOF の扱いに注意が必要である．3.6.2 節に挙げた制約 2 により，対象とするシェルコマンドは EOF を受け取るとその処理を終了してしまう．従って，シェルコマンドをデーモン化するためには，入力するバッチの間に EOF を入れないようにする工夫が必要である

ここで浮上してくる単純ではない問題がある．EOF を使わずに，いかにしてシェルコマンドに入力バッチの間を認識させればよいだろうか．全てのオペレータは，1 つのバッチ入力に対し 1 つのバッチを出力するようにできているため，各入力バッチの終了を認識させることは欠かすことができない．特に，3.3 節で述べたようにウィンドウはバッチと完全に対応しているため，入力

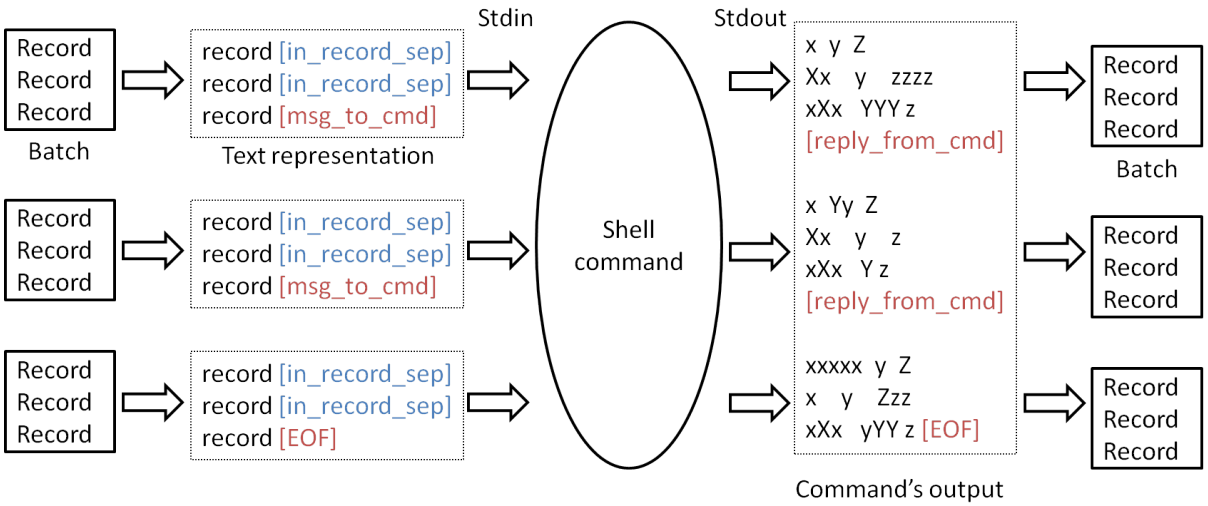

図 4.1　shellstreaming プロセスとデーモン化したシェルコマンドプロセスのデータの受け渡し

バッチと出力バッチの対応は shellstreaming アプリケーション開発者の意図する動作を実現するために重要な性質である．

入力バッチの終了をシェルコマンドに認識させる方法を実現させるために重要となるのが，4.1.2 節の制約 C である．この方法を図 4.1 を用いて図説する．デーモン化しない場合のバッチの入出力方法を示した図 3.7 と比較していただきたい．デーモン化しない場合には，シェルコマンドの出力テキストを最後まで (EOF) まで読んで，それをバッチに変換すれば，シェルコマンドへの入力バッチに対応した出力バッチを得られていた．入力バッチの終わりに EOF を付けずして，シェルコマンドからの出力テキストからバッチの区切りを認識するためには，シェルコマンドへの入力文字列にインジケータ (msg_to_cmd) を付加する．シェルコマンドがこのインジケータに対して出力するテキスト (reply_from_cmd) を予測できていれば，それをバッチの区切りとして使用できる．msg_to_cmd, reply_from_cmd の組は，shellstreaming アプリケーション開発者が，デーモン化シェルコマンドを起動する際に指定する．

コード 4.1 に，デーモン化するシェルコマンドオペレータの例を示す．7,8 行目で，”msg_to_cmd” と “reply_from_cmd” をそれぞれ指定している．この例では使用コマンドが cat であるため，両者は同一のテキストになっている．cat にはバッチの終了時に “msg_to_cmd” が付与されたテキストが与えられ，デーモンを終了するタイミングでは EOF が付与されたテキストが与えられる．

コード 4.1　デーモン化シェルコマンドオペレータの例

```
1 cat_stream = api.Operator(
2     [num_stream], ShellCmd,
3     'cat < IN_STREAM > OUT_STREAM',
4     daemon=True,
5     out_record_def=api.RecordDef([{'name': 'num', 'type': 'INT'}]),
6     out_col_patterns={'num': re.compile(r'^.+$', re.MULTILINE)},
7     msg_to_cmd='THIS IS INDICATOR\n',
8     reply_from_cmd='THIS IS INDICATOR\n')
```

4.2　ローカルワーカを優先したバッチ入力

3.7 節で説明したように，ジョブインスタンスが異なるワーカからバッチを入力する場合には通信を伴う．通信のコストを抑えるために，ローカルワーカに入力すべきバッチを持つキューが存在した場合には，必ずそこから優先してバッチを取る手法を実装した．この手法の効果は 5.1 節で測定した．

4.3　リモートワーカからバッチを取得する際のバッチ集約

ローカルワーカに入力すべきバッチを持つキューが存在しない場合でも，リモートワーカから一回に複数のバッチを受け取ることにより，通信の回数を減らすことができる．通信の回数を減らすことにより，通信の遅延を被る回数を削減でき，性能の向上を見込める．shellstreaming にはこのバッチ集約手法を実装し，5.1 節でその効果を評価した．

4.4　カラム値に応じたデータ分散

shellstreaming で複数のジョブインスタンスが同一のキューからレコードを取る際，基本的には「早いもの勝ち」の方式でバッチ毎に取得される．しかし，一部のアプリケーションによってはレコード中のカラムの値に応じたデータ分散が有効であることがある．

その典型であるワードカウントアプリケーションを例にとって説明する (図 4.2).「早いもの勝ち」方式でレコードを取得した結果，左図の 2 段目のように，各ワーカに “apple” というワードを持つレコードも “orange” というワードを持つレコードも入力されたとすると，最終的に正しく単語の出現回数の総数を求めるためには，3 段目のような集約処理が必要となる．一方で，右図のように入力キューがワードに応じて予め分かれていれば，2 段目の処理をする際に，各ワードは特定のワーカにのみ処理されるようにできる．これにより，集約処理を省略することができ，性能向上に繋がる．

カラム値に応じてキューを分ける際には，キューを全ワーカの数だけ用意し，カラム値のハッシュ値に従ってレコードを push するキューを選択する．それぞれのキューは 1 つずつの実行ワーカに結びついているので，特定のカラム値は特定のワーカにしか処理されないようになる．

カラム値に応じたデータ分散は，各種ジョブ関数の返り値であるストリームオブジェクトに “partition_by_key” 関数を適用することで行える．詳しくは付録 B.2 を参照のこと．この指定をされたストリームオブジェクトを入力に取るジョブは，必ず全てのワーカによってインスタンス化される．これは，上流の分割された全てのキューからレコードを取るための実装上の制約である．

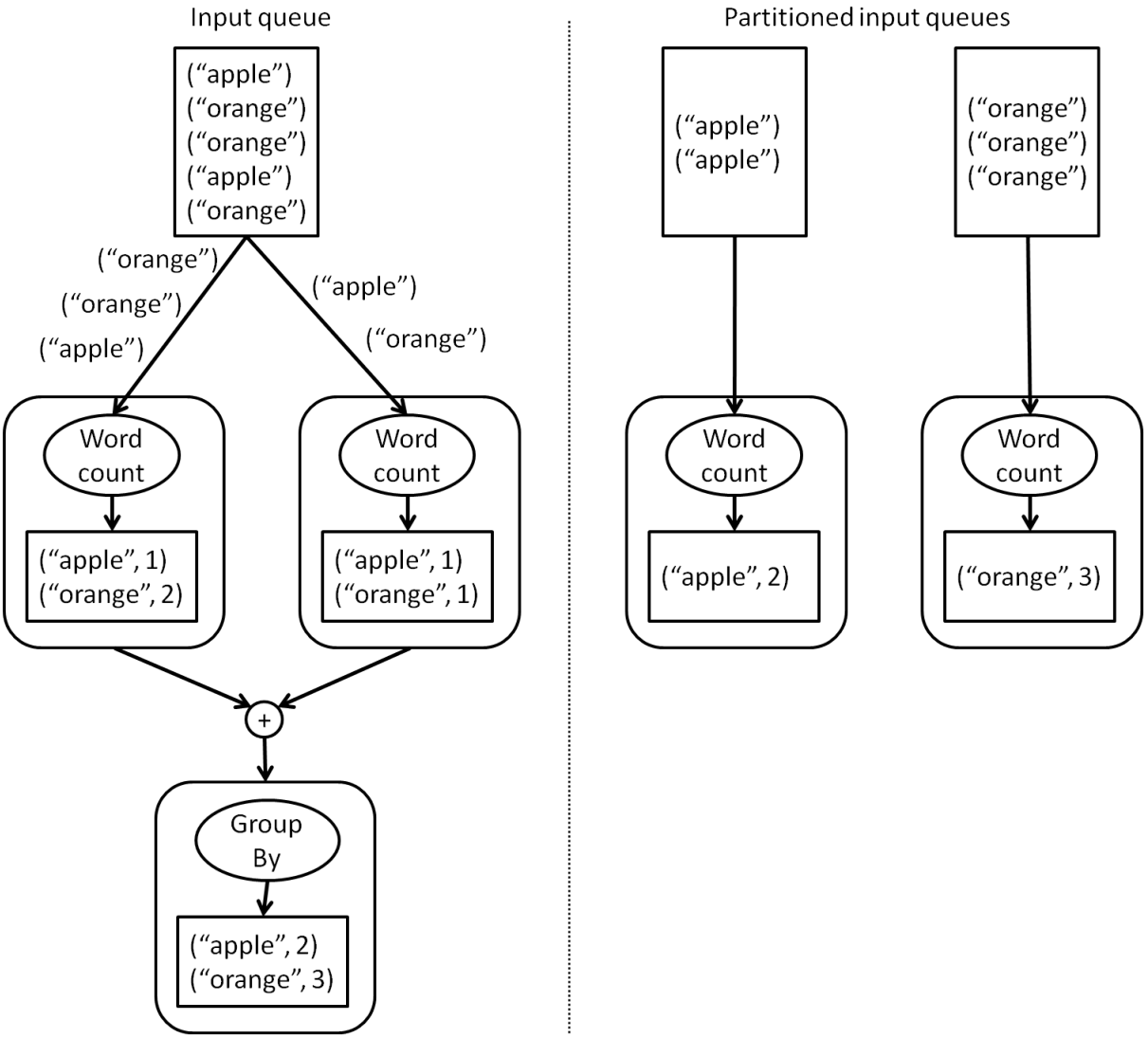

図 4.2　ワードカウントアプリケーションを例とした，カラム値に応じたデータ分散の図説．右ではカラム値に応じたデータ分散を行なっているため，最後の集約処理が不要になっている．

第5章　評価

本章では，shellstreaming の性能評価を行う．評価には，4 章で述べた高速化手法の効果測定と，幅広く利用されているストリーム処理系である Storm[21] との性能比較を含む．

実験環境として，表 5.1 に示されたノードを 10 Gbps のネットワークで接続したクラスタ環境を使用した．

5.1　バッチ取得時の高速化手法の評価

本節では，4.2, 4.3 節に示した，ローカルワーカを優先したバッチ入力手法とリモートワーカからバッチを取得する際のバッチ集約手法を評価する．

この評価には，図 5.1 で示されるアプリケーションを用いた．このアプリケーションは pushpop アプリケーションと呼称する．pushpop アプリケーションの動作は単純である．まず IncInt 入力ストリームが 1, 2, ···, 1,000,000 という 100 万個のレコードを生成し，キューに push する．Null 出力ストリームはキューからレコードを pop し，あとは何もしない．IncInt 入力ストリームも Null 出力ストリームも全てのワーカでインスタンス化され，即ちワーカ数 × 1,000,000 個のレコードが push/pop される．IncInt 入力ストリームの出力するバッチのサイズはレコード 100 個とした．

このアプリケーションを，ワーカの数と各種パラメータを変更して性能測定した結果を図 5.2 に示す．ワーカは各ノードに 1 台のみを立てている．

まず，図における青色とオレンジ色の線に着目する．これらは共に，Null 出力ストリームが入力バッチを取るキューを選択する際に，全てのワーカからランダムに選択する戦略を取っている (Random)．従って，ワーカ数が増加するに従いリモートノードからバッチを取る回数が増加し，通信がボトルネックになる．実際，ワーカ数が増加するのに伴いアプリケーションの処理時間が長くなっていくことが確認できる．実行時間の伸びが段々と緩やかになるのは，ローカルワーカから入力キューを選択する可能性はワーカ数と反比例するからである．青色とオレンジ色の線の違いは，バッチ集約手法の有無である．青色の線ではバッチ集約を行っていないため，リモートワーカからバッチを取る度に 100 個のレコードが入った 1 つのバッチを取っている．一方でオレ

表 5.1　実験環境

CPU	Intel Xeon CPU E5530 @ 2.40GHz, 物理 4 コア × 2 ソケット，Hyper-Threading.
メモリ	24GB
OS	2.6.32-5-amd64

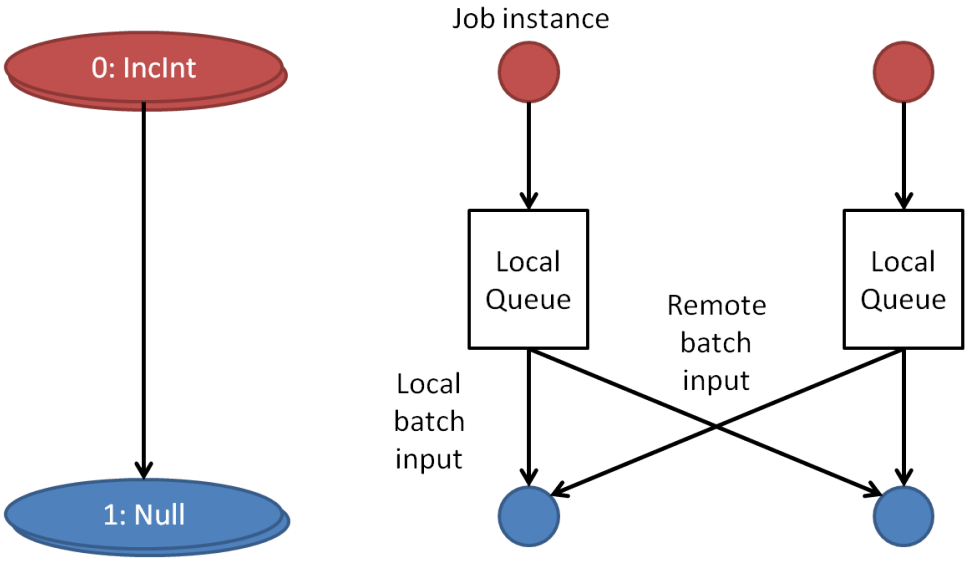

図 5.1　pushpop アプリケーションの動作 (ワーカが 2 台の場合)

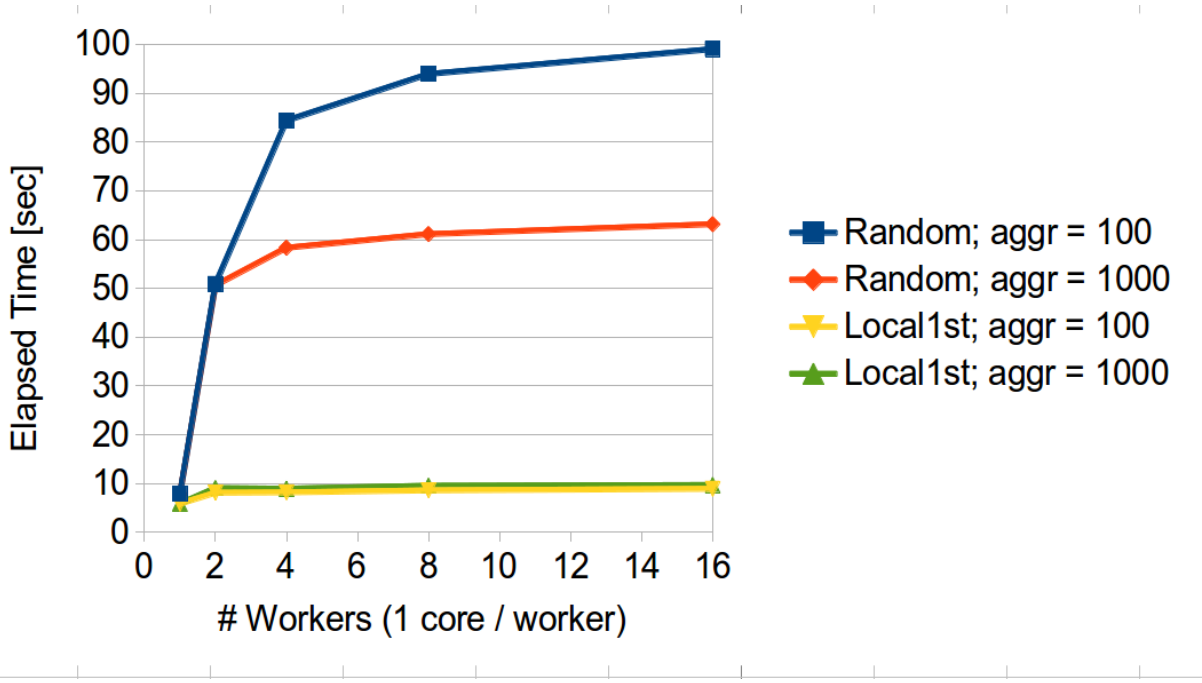

図 5.2　バッチ取得時の高速化手法の評価

ンジ色の線では，リモートワーカからバッチを取る際には 10 個まとめてとる戦略を取っている．この結果，特にリモートワーカとの通信が多発する場合に，処理時間が大幅に減少した．

次に黄色と緑色の線に着目する．これらは，Null 出力ストリームが入力バッチを取るキューを選択する際に，ローカルワーカがバッチを持っている場合にはそれを pop し，そうでない場合に限りリモートワーカからランダムに選択する戦略を取っている (Local1st)．この結果，アプリケー

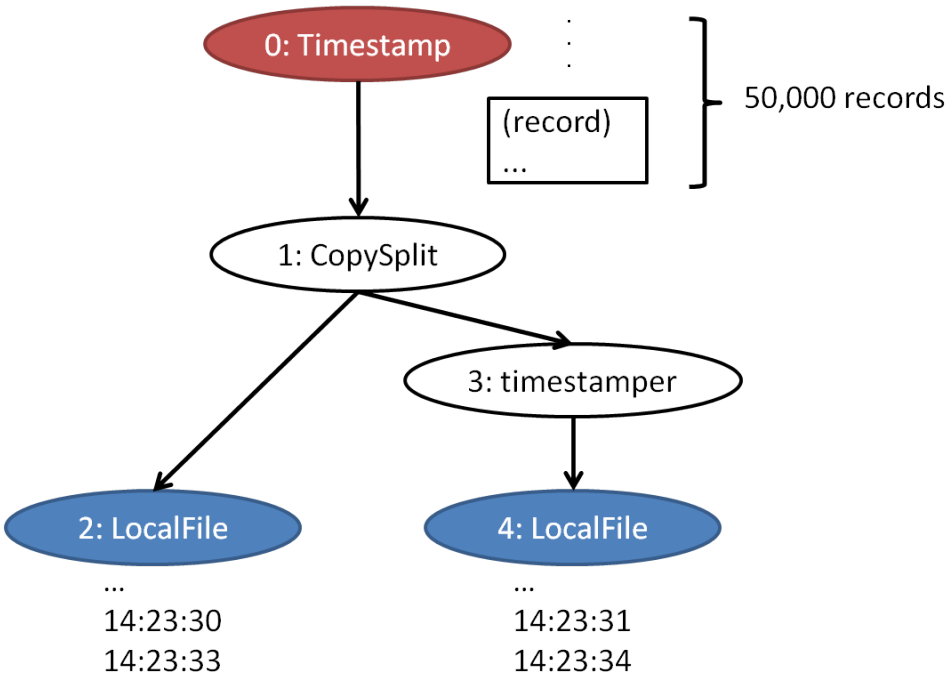

図 5.3　timestamper アプリケーションのジョブグラフ

ション実行時の通信は激減し，CPU ボトルネックのアプリケーションとなっている．ノードを 16 まで増やしても，pushpop のような均一なアプリケーションであればほぼ完全にスケールすることができている．黄色と緑色の違いはバッチの集約数だが，そもそもリモートワーカからバッチを取る機会が少ないため，性能差は見られなかった．しかし，当然ながらより不均一なアプリケーションにおいてはバッチ集約も効果的な戦略となりうることが，青色とオレンジ色の線の比較から分かる．

5.2　シェルコマンドオペレータの性能評価

本節では，シェルコマンドオペレータの基礎的な性能評価を行う．シェルコマンドオペレータのスループット，レイテンシのバッチの大きさによる変化と，4.1 節に示したシェルコマンドのデーモン化手法の評価を含む．

評価には，図 5.3 のアプリケーションを用いた．このアプリケーションを timestamper アプリケーションと呼称する．このアプリケーションは，主に各レコードの処理レイテンシを計測するためのものである．まず，図の “0:Timestamp” 入力ストリームで，現在時刻を中身としたレコードを生成する．そのレコードは “1:CopySplit” によって 2 つのパスにコピーされ，まず “2:LocalFile” でファイルに書き込まれる．もう一方のパスでは，レコードはまず “3:timestamper” に処理される．このオペレータは timestamper というシェルコマンドを立ち上げるシェルコマンドオペレータで，timestamper コマンドは，1 つのレコードを受け取ると，(その中身に関係なく) 現在時刻を出力するものである．このジョブグラフの中では timestamper への入力は “0:Timestamp” 時点での時刻なので，timestamper の入出力の時刻の差は，概ね timestamper シェルコマンドオペレー

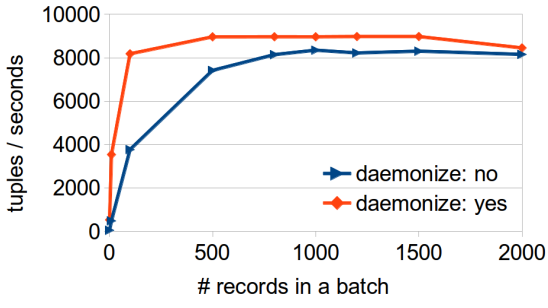

図 5.4　timestamper アプリケーションのスループット

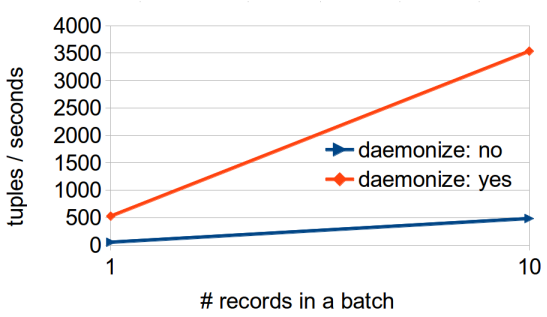

図 5.5　timestamper アプリケーションのスループット (詳細)

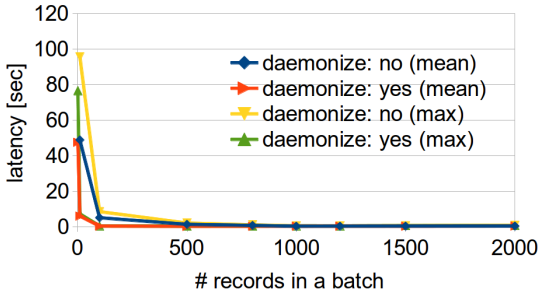

図 5.6　timestamper アプリケーションのレイテンシ

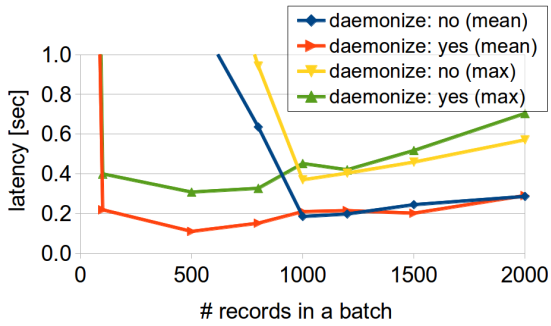

図 5.7　timestamper アプリケーションのレイテンシ (詳細)

タをインスタンス化するのに要した時間であると言える．“3:timestamper” の出力も “4:LocalFile” によってファイルに書き込まれる．

評価では，50,000 個のレコードを処理する際のスループットとレイテンシを計測した．ここでスループットは，50,000 個のレコードを timestamper アプリケーションが処理し終わるための時間から算出し，レイテンシは “2:LocalFile” と “4:LocalFile” が出力する時刻の差とした．通信の影響を排除するために，評価は 1 つのワーカのみを用いて行った．また，シェルコマンドオペレータ性能計測が目的であるため，ファイル書き込みの影響をできる限り小さくするべく，出力ストリームの使用するファイルは tmpfs [1] 上に作成した．

スループットの測定結果を図 5.4, 5.5 に，レイテンシの測定結果を図 5.6, 5.7 に示す．

まず，スループットについて考察する．デーモン化をしない場合には，バッチのサイズが大きくなるに連れてスループットも大きくなる．これは，バッチのサイズが小さいほどシェルコマンドを起動する回数が増加するため，シェルコマンドの起動オーバヘッドによりスループットを発揮できないためである．特に，バッチのサイズを 1 レコード分にし，レコード毎に毎回シェルコマンドを起動する場合には，毎秒 52.7 個のレコードしか処理できていない．バッチのサイズを 1,000 レコード分程度にすると，スループットはそれ以上向上しなくなる．この時点で，ボトルネックはシェルコマンドの起動オーバヘッドでなく，shellstreaming の別の実装部分になったものと考えられる．デーモン化をした場合には，バッチサイズが比較的小さい段階から良好なスループッ

[1] 物理的な記憶媒体としてメインメモリを使用するファイルシステム．

トを発揮している．デーモン化をするとシェルコマンドを起動する回数が 1 度で済むため，シェルコマンドの起動オーバヘッドが無視できるためである．

次に，レイテンシについて考察する．レイテンシは 50,000 レコードのそれぞれについて測定しているが，特にその平均時間と最大時間に着目した．平均，最大レイテンシはストリーム処理アプリケーションの QoS (Quality of Service)[30] 指標として用いられる場合が多く，これを小さくすることはシェルコマンドオペレータにとって重要である．まず，デーモン化の有無にかかわらず，バッチのサイズが小さい場合にはレイテンシが非常に大きくなることが分かる．これは，バッチが小さい場合には “3:timestamper” オペレータがボトルネックとなり，ここでレコードが詰まりを起こすためである．図 5.7 からは，バッチサイズが 800 程度になると，最もレイテンシの大きなレコードでも 1 秒以内に収まっていることが分かる．一方で，バッチサイズが 1000 を超えると，レイテンシが大きくなる傾向が見て取れる．これは，バッチが大きければバッチの末尾にあるレコードが処理されるまでの遅延が大きくなるためである．

以上のことから，デーモン化手法でシェルコマンド起動のオーバヘッドを抑えることでシェルコマンドオペレータの性能向上が達成されることと，スループットとレイテンシを適切に保つための適切なバッチサイズが存在することが確認できる．

5.3　ストリーム処理系との性能比較

本節では，現在最も広く利用されているストリーム処理系であると考えられる Storm[21] との性能比較を行う．データ処理系の一般的なベンチマークであるワードカウント処理による比較を 5.3.1 節に，shellstreaming が得意とする，起動の遅いシェルコマンドを含んだストリーム処理による比較を 5.3.2 節に示す．

Storm は，複数のミドルウェアを基盤として動作する処理系である．評価に使用したソフトウェアのバージョンを表 5.2 に示す．

5.3.1　ワードカウントアプリケーション

本節では，一般的なベンチマークであるワードカウントアプリケーションにより，shellstreaming と Storm の性能比較を行う．

図 5.8 にアプリケーションのジョブグラフを示す．大まかな処理の流れは，

1. ランダムな英文センテンスを生成
2. 英文センテンスを単語に分割

表 5.2　Storm で使用するソフトウェアのバージョン

Storm	0.8.1
ZeroMQ	2.1.7
ZooKeeper	3.3.6

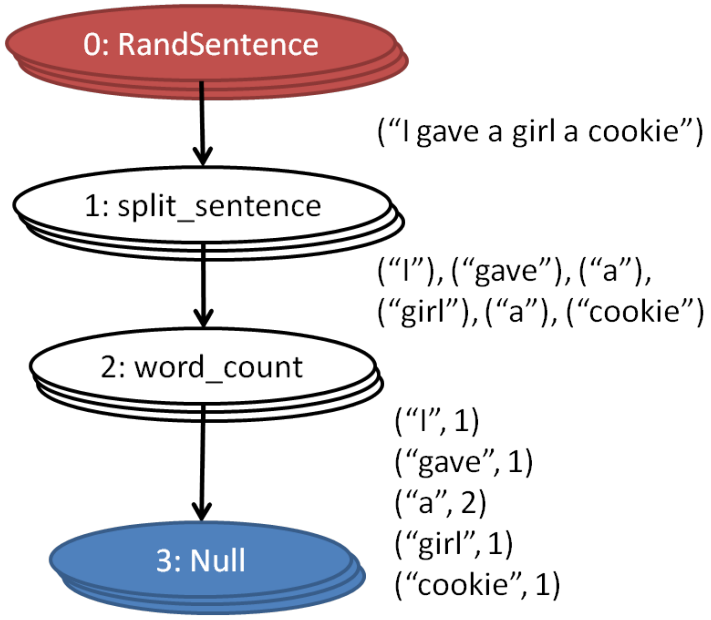

図 5.8　ワードカウントアプリケーションのジョブグラフ

3. 各単語の今までの総出現回数を出力

というものになる．

shellstreaming においては "1:split_sentence" と "2:word_count" に当たる処理をシェルコマンドオペレータで実装した．詳しくは B.2 を参照されたい．各ジョブは，1 つのノードに 1 つずつインスタンス化されるようにした．

Storm では，基本的に storm-starter [2] の WordCountTopology をそのまま用いているが，以下の点に変更を加えた．

- ランダムに生成する英文センテンスを変更[3]
- 各 Bolt に十分な速度で入力されるように，英文センテンスを生成する Spout のスリープ時間を減少
- 各 Bolt, Spout が全てのノードに 1 つずつ配置されるように変更．

実験結果を図 5.9 に示す．青色，オレンジ色の実線がそれぞれ Storm, shellstreaming での性能であり，オレンジ色の破線が shellstreaming で理想的なスケールアウトをした場合の性能である．

Storm は良好なスケールアウトを示している一方で，shellstreaming は理想的な性能には届いていない．16 ノードを使用した場合は Storm の方が 3.7 倍高速である．

これは，shellstreaming ではワーカを跨いだレコードのやり取り，即ち通信部分がボトルネックとなっているためと考えられる．4.4 節の説明と同様に，評価で使用したワードカウントアプリケーションではカラム値に応じたデータ分散を行なっている．即ち，ある単語は必ず同じノードに転送されるようになっているため，"1:split_sentence" と "2:word_count" の間で多くの通信が

[2] Storm の入門的なアプリケーションのセット．`https://github.com/nathanmarz/storm-starter` にて配布されている．

[3] コーパスとして `http://en.wikipedia.org/wiki/Beer` の記事を使用した．

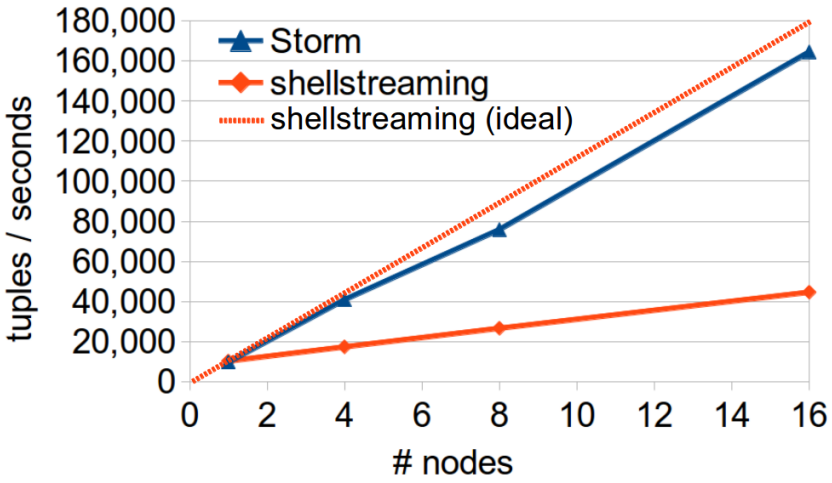

図 5.9　ワードカウントアプリケーションの性能評価

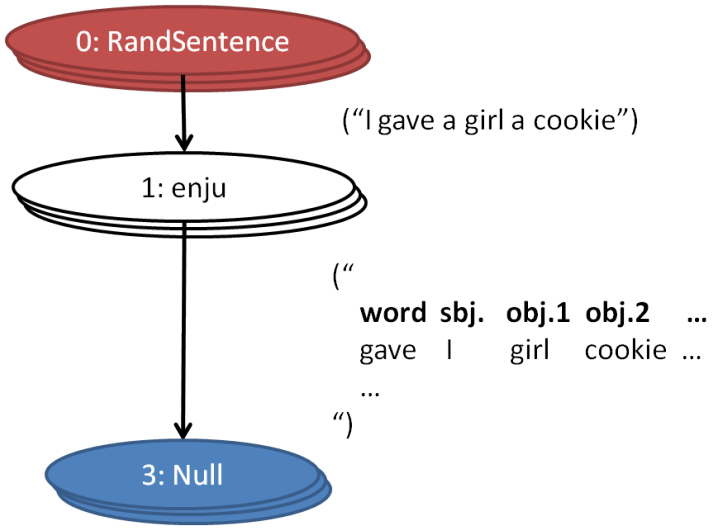

図 5.10　Enju アプリケーションのジョブグラフ

発生している．これがボトルネックとなり，Storm と比べてスケールアウト性が悪くなっていると考えられる．

5.3.2　英文構文解析アプリケーション

本節では，ストリーム処理アプリケーションに起動時間の大きいシェルコマンドを組み込む場合に，shellstreaming でデーモン化手法が効果的であることを示す．

評価には図 5.10 のアプリケーションを用い，shellstreaming と Storm で実行性能を比較した．このアプリケーションでは，まず英文センテンスをランダムに生成し，それを Enju[40, 6] というシェルコマンドによって英文構文解析する．Enju は起動時に辞書定義ファイルを読み込むために起動時間が大きく，表 5.1 のノードを用いて約 8 秒の起動時間を要する．英文構文解析は自然言

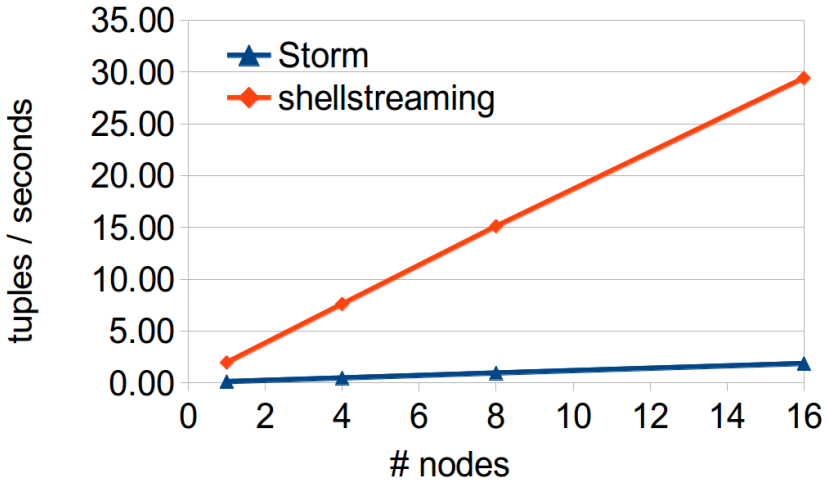

図 5.11　英文構文解析アプリケーションの性能評価

語処理アプリケーションでの応用が多く，shellstreaming がターゲットとするアプリケーションでも使用されることが考えられる．

Enju シェルコマンドを Storm の Bolt として使用するために，2.2 節に示した方法で ShellBolt を使用した．

評価の際，Enju アプリケーションの各ジョブ (または Spout, Bolt) を各ノードにつき 1 つずつ配置した．shellstreaming では，4.2 節で説明した，ローカルワーカを優先したバッチ入力を行うために，ワーカ間でレコードをやり取りするための通信はほとんど発生しない．これは Storm でも同様と考えられる．また，shellstreaming では Enju シェルコマンドをデーモン化している．

実験結果を図 5.11 に示す．各ワーカは均一にジョブを割当てられているため，使用する台数を増やすとスループットは線形に上昇している．shellstreaming ではデーモン化により Enju シェルコマンドを立ち上げるオーバヘッドが 1 度しか掛からないため，15.1 倍高速に動作する．2.2 節で述べたように，Storm の ShellBolt では 1 レコード毎にシェルコマンドを立ち上げている．このため，起動時間が大きい Enju コマンドを使用する Enju アプリケーションでは，デーモン化をする shellstreaming が Storm よりも十分高速に動作する．

第6章　結論

6.1　まとめ

本研究では，シェルコマンドをジョブとして組み込むことのできるストリーム処理系，shellstreaming を提案した．複雑なストリーム処理アプリケーションを開発する際，プログラミング言語を問わず，また自分が開発したかを問わずに，様々なシェルコマンドを拡張ジョブとして拡張することを shellstreaming はサポートする．

本稿では，shellstreaming の基本的な利用例，アーキテクチャ，構成要素について説明し，特にシェルコマンドの入出力と shellstreaming のデータ構造の変換方法について詳説した．また，shellstreaming が行なっている種々の高速化についても説明し，特にシェルコマンドをデーモン化する手法について詳しく述べた．

評価では，複数のワーカ間で通信が発生する場合の性能への影響，デーモン化手法を含むシェルコマンドオペレータの性能評価，Storm[21] 処理系との性能比較を示し，考察した．高速化手法の中心であるシェルコマンドのデーモン化の効果により，ストリーム処理のスループットが顕著に向上することを示し，また Storm との比較でも，起動時間の長いシェルコマンドを含むストリーム処理では 15 倍高速であることを示した．

6.2　今後の課題

最後に，本研究の今後の課題を述べる．

第一に，shellstreaming のフォールトトレラント性の実装が重要である．ストリーム処理系は基本的に処理時間の長いアプリケーションを動作させる基盤であり，動作中にノードの物理故障や処理系の不具合に見舞われる確率が高くなる．そのような場合でも，アプリケーション処理を停止せずに復旧するための手段を提供するのは極めて重要と言える．

性能面においては，多数の改善の余地が残されている．現在の shellstreaming の大きなボトルネックはネットワーク通信であることが見受けられるため，ネットワークを介したレコードのやり取りの高速化は単純ながら重要であると考える．また，高効率なスケジューラの開発も重要である．ストリーム処理系のスケジューラの研究は広く行われていて，その中でも特に適応的スケジューリング [27] は，刻一刻と変化する入力に応じて最適なジョブの配置を行うために有効であると考えられる．更に，ジョブグラフが生成された段階で，RDBMS の理論におけるクエリ最適化のようにジョブグラフを変形することも検討の余地がある．

謝辞

修士生活で，或いはそれ以前からお世話になった皆様に感謝申し上げます．私がこのような形で修士論文をまとめあげることができたのも，皆様が心の支えになった部分が少なからずあるでしょう．ここからは，直接的にまたは特別にお世話になった方々への感謝を述べますが，そうでない方々へのお礼は又の機会にできればと思います．

まず誰よりも指導教員の田浦先生，コンパイラ実験や OS の授業を含めると 3 年半でしょうか．長きに渡る丁寧なご指導をありがとうございました．研究テーマを大なり小なりころころ変えたり，SQLite の高速化にのめり込んだりと，ともすれば飽き性とも言える私でしたが，幅広い知識をお持ちの先生の元で過ごせたからこそ全てのことに満足感を持って取り組めたと思っております．もっと色々な面で報いたかった部分はございますが，至らぬ私をお許しください．

また，博士課程の井上 拓さんと秋山 茂樹さんは，自身の研究について色々とアドバイスをくださったり気にかけてくださっていました．ご指導誠にありがとうございました．

OB・OG の中では，特に Chen Ting さん，加辺 友也さん, 堀内 美希さん, 河野 瑛さんにお世話になりました．Ting-san, thank you for all the kindness to me. I enjoyed discussing on research and programming, and just chatting with you sometimes. Take me to Chinese restaurants and billiards again! 加辺さんは口うるさいこともありましたが，それだけ気にかけてくれてるのが嬉しかったです．サーバ管理術をはじめ，色々な知識を吸収させてもらいました．口うるさいこともありましたが．また会社でも先輩としてお世話になります．お手柔らかにお願いします．ミキティさんは色々な面でとても良い先輩でした．修士 2 年になられた辺りから研究の成果もメキメキ出し始め，研究面でも尊敬しておりました．私生活面でも，私の (時に) 下らない相談に乗ってくださることが多く，大変助かりました．プログラミングコンテストや学会前の追い込みもとても楽しかったです．今後も (1 ホップ介して) お会いできる機会が多くあることを願います．河野さんはとても楽しい先輩でした．以上．

同期の Ganbat Amgalan, 林 伸也, 中澤 隆久，これを書いてる時点ではとりあえず無事に皆揃って修了できそうでよかったです．いつか昔話に花を咲かせましょう．

後輩の皆さんは当然全員かわいいのですが，印象深いのは M1 の早水 光と B4 の島津 真人です．ISUCON 予選も出ましたしね．卒業しても遊んでください．特に島津は実装面で困った時の素晴らしい相談相手でした．2 人とも実力も人格も申し分ないと思いますが，更に成長して先輩を驚かせてください．早水は体は成長しちゃだめだぞ．

家族にも大変世話になりました．研究が切羽詰まった時には実家に帰ってタダ飯タダ酒タダ風呂を頂戴しました．もう少しで 26 歳，まさかこんなに遅くなるとは思っていませんでしたが，これから沢山恩返しさせてください．

最初に述べたように，この他にも名前を挙げ切れないほど沢山の方にお世話になりました．皆様への心からの感謝をもって，謝辞を締めさせていただきます．

参考文献

[1] Apache HBase. http://hbase.apache.org/.

[2] Apache Hive. http://hive.apache.org/.

[3] Apache Software Foundation. Hadoop. http://hadoop.apache.org.

[4] Apache Software Foundation. Pig. http://pig.apache.org/.

[5] Cloudera Impala Community. http://impala.io/.

[6] Enju - A fast, accurate, and deep parser for English. http://www.nactem.ac.uk/enju/.

[7] EsperTech: Event Stream Intelligence. http://www.espertech.com/.

[8] Fabric. http://fabfile.org.

[9] Fluentd: Open Source Log Management. http://fluentd.org/.

[10] Hadoop Streaming. http://hadoop.apache.org/docs/stable1/streaming.html.

[11] IBM - InfoSphere Platform. http://www-01.ibm.com/software/data/infosphere/downloads/.

[12] JDBC Overview. http://www.oracle.com/technetwork/java/overview-141217.html.

[13] Jubatus : Distributed Online Machine Learning Framework. http://jubat.us/en/.

[14] Microsoft StreamInsight. http://technet.microsoft.com/library/ee362541.aspx.

[15] Oracle Complex Event Processing. http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html.

[16] Overview of SPL and related documentation. http://pic.dhe.ibm.com/infocenter/streams/v2r0/topic/com.ibm.swg.im.infosphere.streams.doc/doc/ibminfospherestreams-spl-introduction.html.

[17] rainbow_logging_handler : Python Package Index. https://pypi.python.org/pypi/rainbow_logging_handler.

[18] S4: Distributed Stream Computing Platform. http://incubator.apache.org/s4/.

[19] ShellBolt. http://nathanmarz.github.io/storm/doc/backtype/storm/task/ShellBolt.html.

[20] Spark Streaming. http://spark.incubator.apache.org/streaming/.

[21] Storm, distributed and fault-tolerant realtime computation. http://storm-project.net/.

[22] The Apache HTTP Server Project. http://httpd.apache.org/.

[23] The Open Group Base Specifications Issue 7, 2013 Edition. http://pubs.opengroup.org/onlinepubs/9699919799/.

[24] The Streaming APIs | Twitter Developers. https://dev.twitter.com/docs/streaming-apis.

[25] Web Click Stream Analysis using Linux Clusters. http://www.ibm.com/developerworks/data/library/techa

[26] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 1:1–1:13, New York, NY, USA, 2012. ACM.

[27] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[28] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[29] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.

[30] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective (Advances in Database Systems)*. Springer, 2009 edition, 5 2009.

[31] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[32] J. Chauhan, S.A. Chowdhury, and D. Makaroff. Performance evaluation of yahoo! s4: A first look. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on*, pages 58–65, Nov 2012.

[33] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. *SIGMOD Rec.*, 29(2):379–390, May 2000.

[34] Ting Chen and K. Taura. A selective checkpointing mechanism for query plans in a parallel database system. In *Big Data, 2013 IEEE International Conference on*, pages 237–245, Oct 2013.

[35] Ting Chen and Kenjiro Taura. ParaLite: Supporting Collective Queries in Database System to Parallelize User-Defined Executable. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 474–481, Washington, DC, USA, 2012. IEEE Computer Society.

[36] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[37] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[38] Bry F, Eckert M, Etzion O, Paschke A, and Riecke J. Event Processing Language Tutorial. *DEBS, 3rd ACM International Conference on Distributed Event-Based Systems*, July 2009.

[39] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[40] Yusuke Miyao, Takashi Ninomiya, and Jun 'ichi Tsujii. Corpus-oriented grammar development for acquiring a head-driven phrase structure grammar from the penn treebank. In Keh-Yih Su, Jun 'ichi Tsujii, Jong-Hyeok Lee, and OiYee Kwong, editors, *Natural Language Processing — IJCNLP 2004*, volume 3248 of *Lecture Notes in Computer Science*, pages 684–693. Springer Berlin Heidelberg, 2005.

[41] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: Continuous Pig/Hadoop Workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1081–1090, New York, NY, USA, 2011. ACM.

[42] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[43] Mike Schroepfer. Inside Large-Scale Analytics at Facebook. Hadoop Summit, June 2010.

[44] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[45] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun, and Jun ichi Tsujii. Design and Implementation of GXP Make - A Workflow System Based on Make. In *eScience*, pages 214–221. IEEE Computer Society, 2010.

[46] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[48] 照之 今井, 知生 海老山, 弘司 喜田, 健一郎 藤山, and 暢達 中村. D-034 データストリーム処理手法を用いた web アクセス解析システム (データベース, 一般論文). **情報科学技術フォーラム講演論文集**, 8(2):207–208, aug 2009.

[49] 富士通株式会社. 平成 22 年度産業技術研究開発委託費（次世代高信頼・省エネ型 IT 基盤技術開発事業（大規模データストリーム処理基盤の研究開発)） 事業報告書. Technical report, 2001.

発表文献

国際発表（査読あり）

- Jun Nakashima, Sho Nakatani, and Kenjiro Taura. Design and Implementation of a Customizable Work Stealing Scheduler. *3rd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS2013)*, Eugene, Oregon, USA. Jun 2013.

国内発表（査読なし）

- 中谷翔, Ting Chen, 田浦健次朗. ワークフローアプリケーション基盤としての並列 DB の性能評価. **並列／分散／協調処理に関するサマーワークショップ（*SWOPP2012*）**, 鳥取, 2012/8.
- 中谷翔, Ting Chen, 田浦健次朗. ストリーミングデータを扱うワークフローの外部モジュールの扱いに着目した低遅延実行. **並列／分散／協調処理に関するサマーワークショップ（*SWOPP2013*）**, 宮城, 2013/8.
- 中島潤, 中谷翔, 田浦健次朗. スケジューリング方針をカスタマイズ可能な軽量スレッド処理系の提案. **並列／分散／協調処理に関するサマーワークショップ（*SWOPP2013*）**, 宮城, 2013/8.

受賞

- 第 1 回 ICT プログラミングコンテスト 受賞. 2012/5.
- 2012 年度 未踏 IT 人材発掘・育成事業 採択. 2012/8.

付 録A　shellstreaming の配布先

shellstreaming は，`https://github.com/laysakura/shellstreaming` に配布されている．使用方法は配布ページのドキュメントを参照．

付 録B　shellstreaming アプリケーション例

B.1　Apache HTTP Server アクセスログ解析

3.1 節での説明に用いた，Apache HTTP Server のアクセスログ解析処理のアプリケーションをコード B.1 に示す．この解析は，Web ページのトップページ (/ ページ) への GET アクセスのうち，現在 (2014 年 1 月 4 日を想定) の日にちから 4 日前までのものを対象に，日毎のアクセス数とレスポンス時のステータスコード分布 (図 3.2) を集計するためのものである．コード B.1 で表現しているジョブグラフは図 3.1 のものである．

アプリケーションの実行環境として，以下の構成を想定している．

ウェブサーバノード (アクセスログ出力先) WebServer0, WebServer1, WebServer2

ワーカノード (アクセスログ解析ノード) WebServer0, WebServer1, WebServer2, Worker0, Worker1

以下，動作を簡潔に説明する．

1. WebServer0, WebServer1, WebServer2 が Apache HTTP Server のアクセスログファイルをローカルファイルとして持つ．
2. WebServer0, WebServer1, WebServer2 がアクセスログ shellstreaming のレコードとして入力する．アクセスログへ追記が発生するたびに，下流のオペレータが実行される．
3. TextFileTail 入力ストリーム (16-19 行目)，ExternalTimeWindow オペレータ (73-77 行目)，LocalText 出力オペレータ (114-115, 151-152 行目) 以外のジョブは，WebServer0, WebServer1, WebServer2, Worker0, Worker1 の全てのワーカでインスタンスが実行される可能性がある．
4. ExternalTimeWindow オペレータは，WebServer0, WebServer1, WebServer2 の全てのログを集約する必要があるために，Worker0 のみでオペレータのインスタンスを実行するようにしている (77 行目)．
5. LocalFile 出力ストリームは，ログの集約結果を 1 箇所に出力するために， Worker1 に固定している (115, 152 行目)．

あとの詳細はコード中のコメントを参照のこと．

コード B.1　Apache HTTP Server アクセスログ解析アプリケーション

```
1 # -*- coding: utf-8 -*-
2 from os.path import join, abspath, dirname
3 import re
4 from shellstreaming import api
5 from shellstreaming.istream import TextFileTail
6 from shellstreaming.operator import ShellCmd, ExternalTimeWindow, CopySplit
7 from shellstreaming.ostream import LocalFile
```

```
8
9
10 APACHE_LOG    = '/var/log/apache2/access.log'
11 DAILY_ACCESS  = '/tmp/51_apache_log_analysis_daily.txt'
12 STATUS_CODES = '/tmp/51_apache_log_analysis_statuscode.txt'
13
14
15 def main():
16     log_stream = api.IStream(
17         TextFileTail, APACHE_LOG,
18         read_existing_lines=True,  # read not only newly appended lines
19         fixed_to=['WebServer0', 'WebServer1', 'WebServer2'])  # where logs exist
20
21     # filter lines in which '/' is 'GET' accessed
22     access_stream = api.Operator(
23         [log_stream], ShellCmd,
24         r'''grep -E '"GET / HTTP/[.0-9]+"' < IN_STREAM > OUT_STREAM''',
25         # grep commands returns  exitcode 1 when no input line matches to given pattern
26         success_exitcodes=(0, 1),
27         out_record_def=api.RecordDef([
28             {'name': 'ipaddr'     , 'type': 'STRING'},
29             {'name': 'timestamp'  , 'type': 'STRING'},
30             {'name': 'statuscode' , 'type': 'INT'},
31         ]),
32         # output line example:
33         # 151.217.31.218 - - [30/12/2013:10:29:50 +0900] "GET / HTTP/1.1" 200 265 "-"
               "-"
34         out_col_patterns={
35             'ipaddr'     : re.compile(r'^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+', re.MULTILINE),
36             'timestamp'  : re.compile(r'(?<= - - \[).+(?=\] )', re.MULTILINE),
37             'statuscode' : re.compile(r'[0-9]{3}', re.MULTILINE),
38         },
39     )
40
41     # change format of timestamp to fit shellstreaming's TIMESTAMP format
42     ts_access_stream = api.Operator(
43         [access_stream], ShellCmd,
44         (
45             r"sed -e 's/"
46             r"^\\([0-9]*\\.[0-9]*\\.[0-9]*\\.[0-9]*\\)|"    # ip address (|1)
47             r"\\([0-9]*\\)\\/\\([0-9]*\\)\\/\\([0-9]*\\):" # dd(|2),mm(|3),YYYY(|4)
48             r"\\([0-9]*\\):\\([0-9]*\\):\\([0-9]*\\).*|"    # HH(|5),MM(|6),SS(|7)
49             r"\\([0-9]*\\)"  # status code (|8)
50             r".*$"
51             r"/beg-ipaddr \\1 end-ipaddr "
52             r"beg-timestamp \\4-\\3-\\2 \\5:\\6:\\7 end-timestamp "
53             r"beg-statuscode \\8 end-statuscode/g' "
54             r"< IN_STREAM > OUT_STREAM"
55         ),
56         in_column_sep='|',
57         out_record_def=api.RecordDef([
58             {'name': 'ipaddr'     , 'type': 'STRING'},
59             {'name': 'timestamp'  , 'type': 'TIMESTAMP'},
60             {'name': 'statuscode' , 'type': 'INT'},
61         ]),
62         # output line example
63         # beg-ipaddr 151.217.31.218 end-ipaddr beg-timestamp 2013-12-30 10:29:59 end-
               timestamp beg-statuscode 200 end-statuscode
64         out_col_patterns={
65             'ipaddr'     : re.compile(r'(?<=beg-ipaddr ).+(?= end-ipaddr)'),
66             'timestamp'  : re.compile(r'(?<=beg-timestamp ).+(?= end-timestamp)'),
67             'statuscode' : re.compile(r'(?<=beg-statuscode ).+(?= end-statuscode)'),
68         },
69     )
70
71     # make window between 2014/01/01 and 2014/01/04.
```

```
72      # this operator collects records from all workers to Worker0
73      access_win = api.Operator(
74          [ts_access_stream], ExternalTimeWindow,
75          timestamp_column='timestamp',
76          size_days=4, latest_timestamp=api.Timestamp('2014-01-04 23:59:59'),
77          fixed_to=['Worker0'])
78
79      # copy input records and distribute them into 2-path
80      access_win0, access_win1 = api.Operator([access_win], CopySplit, 2)
81
82      ########################################
83      # path 1: group by date
84      ########################################
85
86      # projection: get timestamp column & retrieve date
87      date_win = api.Operator(
88          [access_win0], ShellCmd,
89          r'''awk '{print $2}' < IN_STREAM > OUT_STREAM''',
90          out_record_def=api.RecordDef([{'name': 'date'  , 'type': 'STRING'}]),
91          out_col_patterns={'date': re.compile(r'^.+$', re.MULTILINE)})
92
93      # sort date for `uniq -c` command
94      sorted_date_win = api.Operator(
95          [date_win], ShellCmd,
96          r'''sort < IN_STREAM > OUT_STREAM''',
97          out_record_def=api.RecordDef([{'name': 'date', 'type': 'STRING'}]),
98          out_col_patterns={'date': re.compile(r'^.+$', re.MULTILINE)})
99
100     # group by date
101     count_group_by_date = api.Operator(
102         [sorted_date_win], ShellCmd,
103         r'''uniq -c < IN_STREAM > OUT_STREAM''',
104         out_record_def=api.RecordDef([
105             {'name': 'count' , 'type': 'INT'},
106             {'name': 'date'  , 'type': 'STRING'},
107         ]),
108         out_col_patterns={
109             'count': re.compile(r'\d+', re.MULTILINE),
110             'date': re.compile(r'\d{4}-\d{2}-\d{2}', re.MULTILINE),
111         })
112
113     # write output records on Worker1
114     api.OStream(count_group_by_date, LocalFile, DAILY_ACCESS,
115                 output_format='json', fixed_to=['Worker1'])
116
117     ########################################
118     # path 2: group by status code
119     ########################################
120
121     # projection: get status code
122     statuscode_win = api.Operator(
123         [access_win1], ShellCmd,
124         r'''awk -F "|" '{print $3}' < IN_STREAM > OUT_STREAM''',
125         in_column_sep='|',
126         out_record_def=api.RecordDef([{'name': 'statuscode', 'type': 'INT'}]),
127         out_col_patterns={'statuscode': re.compile(r'^.+$', re.MULTILINE)})
128
129     # sort date for `uniq -c` command
130     sorted_statuscode_win = api.Operator(
131         [statuscode_win], ShellCmd,
132         r'''sort < IN_STREAM > OUT_STREAM''',
133         out_record_def=api.RecordDef([{'name': 'statuscode', 'type': 'INT'}]),
134         out_col_patterns={'statuscode': re.compile(r'^.+$', re.MULTILINE)})
135
136     # group by date
137     count_group_by_statuscode = api.Operator(
```

```
138         [sorted_statuscode_win], ShellCmd,
139         r'''uniq -c < IN_STREAM > OUT_STREAM''',
140         out_record_def=api.RecordDef([
141             {'name': 'count'      , 'type': 'INT'},
142             {'name': 'statuscode', 'type': 'INT'},
143         ]),
144         out_col_patterns={
145             'count'      : re.compile(r'\d+', re.MULTILINE),
146             'statuscode' : re.compile(r'\d{3}', re.MULTILINE),
147         },
148     fixed_to=['localhost'])
149
150     # write output records on Worker1
151     api.OStream(count_group_by_statuscode, LocalFile, STATUS_CODES,
152                 output_format='json', fixed_to=['Worker0'])
```

B.2　ワードカウント

ここでは，5.3.1 節の評価で用いたワードカウントアプリケーションを掲載する．アプリケーションのジョブグラフは図 5.8 のものである．

ワードカウントアプリケーションにおいては，ノードを跨いだ集約処理が発生しないように，図 4.2 を用いて説明したカラム値によるレコードの分散を行なっている (コード B.4 の 29 行目)

まず，図 5.8 での “split_sentence”, “word_count” に当たるシェルコマンドをそれぞれコード B.3, ??に示す．共に Python によって記述されているが，別の言語で記述することも可能である．

これらを用いた shellstreaming のアプリケーションはコード B.4 のものである．ただし，5.3.1 節の評価では，17 行目において “fixed_to” パラメータを指定し，全てのノードから英文センテンスが生成されるようにした[1]．

コード B.2　スペースによる単語分割シェルコマンド

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  import sys
4
5
6  if __name__ == '__main__':
7      sys.stderr.write('input any sentence (Ctrl-D to finish)\n')
8      while True:
9          sentence = sys.stdin.readline()
10         if sentence == '':  # EOF
11             sys.exit(0)
12         words = sentence.split(' ')
13         for word in words:
14             sys.stdout.write(word.strip('.,;:?!()[]"\'') + '\n')
15             sys.stdout.flush()
```

コード B.3　単語の出現回数を出力するシェルコマンド

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  import sys
4
5
```

[1]”fixed_to” パラメータを指定しない限り，入力ストリームは 1 つのワーカでのみインスタンス化される．これは，データソースによっては API 制限などがあるため，安全寄りの設計にしたためである．

```
if __name__ == '__main__':
    worddict = {}  # {'word0': count0, 'word1': count1, ...}
    sys.stderr.write('input any word (Ctrl-D to finish)\n')
    while True:
        word = sys.stdin.readline()
        if word == '':  # EOF
            sys.exit(0)
        word = word.strip()
        if ' ' in word:
            sys.stdout.write('single word is expected\n')
            sys.stdout.flush()
            continue
        if word not in worddict:
            worddict[word] = 0
        worddict[word] += 1
        sys.stdout.write('%s %d\n' % (word, worddict[word]))
        sys.stdout.flush()
```

コード B.4　shellstreaming によるワードカウントアプリケーション

```
# -*- coding: utf-8 -*-
from os.path import abspath, dirname
import re
from shellstreaming import api
from shellstreaming.istream import RandSentence
from shellstreaming.operator import ShellCmd
from shellstreaming.ostream import LocalFile


OUTPUT_FILE     = '/tmp/50_wordcount.txt'
SHELLCMD_DIR    = abspath(dirname(__file__))
SPLIT_SENTENCE = '/path/to/split_sentence'
WORD_COUNT      = '/path/to/word_count'


def main():
    sentence_stream = api.IStream(RandSentence)
    word_stream = api.Operator(
        [sentence_stream], ShellCmd,
        '%s < IN_STREAM > OUT_STREAM' % (SPLIT_SENTENCE),
        out_record_def=api.RecordDef([{'name': 'word', 'type': 'STRING'}]),
        out_col_patterns={'word': re.compile(r'^.+$', re.MULTILINE)},
        # for daemonizing
        daemon=True,
        msg_to_cmd='extraordinarylongword\n',
        reply_from_cmd='extraordinarylongword\n')

    # partition records by `word` column's value
    word_stream.partition_by_key('word')

    wc_stream = api.Operator(
        [word_stream], ShellCmd,
        '%s < IN_STREAM > OUT_STREAM' % (WORD_COUNT),
        out_record_def=api.RecordDef([
            {'name': 'word', 'type': 'STRING'},
            {'name': 'count', 'type': 'INT'}
        ]),
        out_col_patterns={
            'word'  : re.compile(r'^.+(?= )', re.MULTILINE),
            'count' : re.compile(r'\d+$', re.MULTILINE),
        },
        daemon=True,
        msg_to_cmd='not word\n',
        reply_from_cmd='single word is expected\n')

```

```
46      api.OStream(wc_stream, LocalFile, OUTPUT_FILE,
47                  output_format='json', fixed_to=['localhost'])
```