Master Thesis

Programmable scheduling in a stream processing system

Author(s): Lattuada, Andrea

Publication Date: 2016

Permanent Link: https://doi.org/10.3929/ethz-a-010648300

Rights / License: In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 144

Systems Group, Department of Computer Science, ETH Zurich

Programmable scheduling in a stream processing system

by
Andrea Lattuada

Supervised by
Prof. Timothy Roscoe
Dr. Frank McSherry

October 2015–April 2016
Abstract

The need for frameworks to express distributed computation in a safe and reliable manner has, in the recent years, resulted in renewed interest for the dataflow programming model, which represents a program as a graph of interconnected operators that perform data transformations.

Many research-oriented and industry-grade systems have employed this model to describe “streaming” transformations and aggregation for big-data analytics and machine learning tasks.

The growing expressive power of this kind of systems has brought back the need to devise efficient scheduling and resource-management strategies.

In this work we identify topologies and behaviours that are the source of scheduling challenges; we survey the existing streaming dataflow systems and the existing flow control techniques, their benefits and their shortcomings.

Building on this understanding, we explore a design that does not require advanced scheduling to be built into the system to adequately address the resource-management needs of a wide range of programs but allows the user to drive the execution schedule of a specific computation.

We report on a prototype implementation of programmable, user-level flow control pattern for Timely Dataflow, an highly-expressive distributed stream processing engine. We discuss how the pattern is effective in driving scheduling in real-world use-cases in a resource-efficient manner and with limited performance overhead.
I would like to thank Prof. Timothy Roscoe for the chance to work in the Systems Group and for his support of my efforts.

I am deeply grateful to Dr. Frank McSherry for the opportunity to work on Timely Dataflow, an extremely interesting system, for many eye-opening discussions and for his invaluable advice and support.

I am also indebted to Prof. Ankit Singla, Dr. Desislava Dimitrova, Dr. John Liagouris, Devesh Agrawal, Lucas Braun, and Zaheer Chothia for their advice and suggestions.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Publication</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Background</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Actor systems</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Dataflow</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Timely Dataflow</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Experimental configuration</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Computation edges and the backpressure problem</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>The <code>flatMap</code> operator</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Input/output rate</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Rate imbalance</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>An unstable computation, dataflow-join</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>Channel size as a system metric</td>
<td>22</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Known Techniques</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Source backpressure</td>
<td>25</td>
</tr>
<tr>
<td>5.2</td>
<td>Edge-by-edge backpressure</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Flow-control</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>Flow control strategy</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>System Primitives</td>
<td>32</td>
</tr>
<tr>
<td>6.3</td>
<td>The Faucet flow-control pattern</td>
<td>34</td>
</tr>
<tr>
<td>6.4</td>
<td>A more complex topology</td>
<td>37</td>
</tr>
<tr>
<td>6.5</td>
<td>Timestamp semantics</td>
<td>38</td>
</tr>
<tr>
<td>6.6</td>
<td>Summary</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>Interface changes to support Faucet</td>
<td>41</td>
</tr>
<tr>
<td>7.1</td>
<td>Progress tracking constraints on operators</td>
<td>41</td>
</tr>
<tr>
<td>7.2</td>
<td>Progress tracking behaviour for the batcher</td>
<td>41</td>
</tr>
</tbody>
</table>
### Contents

7.3 Generalised constraints, capabilities ........................................ 42
7.4 Implementation ................................................................. 42
7.5 Safety .................................................................................. 43
7.6 Summary ............................................................................... 44

8 Experiments ............................................................................. 45
8.1 Configuration ......................................................................... 45
8.2 Topologies ............................................................................ 46

9 Parameters ................................................................................ 47
9.1 Synchronization points introduced by the pattern ...................... 47
9.2 Concurrent batches, \( N_{\text{batches}} \) ........................................... 48
9.3 Batch size, \( B \) ................................................................. 49
9.4 Summary ................................................................................ 50

10 Evaluation ............................................................................... 51
10.1 Buffer size ............................................................................ 51
10.2 Overhead .............................................................................. 52
10.3 Peak memory consumption .................................................. 53
10.4 Summary ............................................................................... 54

11 Conclusion ............................................................................... 55
11.1 Main contributions .............................................................. 55
11.2 Future Work ........................................................................ 56

Appendix A Original operator interface constraints .......................... 57

Bibliography ................................................................................. 59
Chapter 1

Introduction

The need for frameworks to express distributed computation in a safe and reliable manner has, in the recent years, resulted in renewed interest for the dataflow programming model, which represents a program as a graph of interconnected operators that perform data transformations.

Many research-oriented and industry-grade systems have employed this model to describe "streaming" transformations and aggregation for big-data analytics and machine learning tasks. Support for distributed execution is provided out-of-the-box, and the programming interface proved to be general enough to express a wider variety of algorithms.

This increased generality and expressivity has brought back the need to devise efficient scheduling and resource-management strategies to deal with uneven rates for data production and consumption in various parts of the computation graph.

In this work we survey the existing streaming dataflow systems and flow-control techniques, their benefits and their shortcomings. We study the implementation of an algorithm that leverages the expressive power of the model and presents non-robust behaviour when a naïve execution schedule is selected.

We identify operator behaviours and classes of topologies that require advanced flow-control and resource management mechanisms and challenge existing techniques.

Building on this understanding, we explore a design that does not require advanced scheduling to be built into the system to adequately address the resource-management needs of a wide range of programs but allows the user to drive the execution schedule of a specific computation.

We describe how we modified and extended the programming interface of Timely Dataflow[36] to support Faucet, our programmable flow-control pattern implemented at the operator level by leveraging the expressive power of the system.

Finally, we evaluate the effectiveness of the pattern in providing reliable and programmable resource management in a real-world use-case and examine its performance characteristics.
Chapter 2

Publication

Part of the work included in this dissertation has been submitted and accepted for publication as an "extended abstract" at "Algorithms and Systems for MapReduce and Beyond (BeyondMR)"¹, "a workshop for research at the frontier of large-scale computations, with topics ranging from algorithms, to computational models, to the systems themselves". BeyondMR 2016 is held in conjunction with SIGMOD/PODS2016, in San Francisco, USA on Friday July 1, 2016.

3.1 Actor systems

The actor model has been introduced in 1973[25] as an “architecture and definitional method for artificial intelligence”. Its semantics have been extended to represent a generic system[24] and the model has blossomed into a host of derived models and definitions to fit various use cases. It generally describes a system as a collection of objects, called actors.

In the contemporary model[20] depicted in figure 3.1 and implemented by a multitude popular concurrent systems, actors are regarded as the units of computations and communicate by exchanging messages. Actors have the following properties:

- actors maintain local state that persists among invocations and cannot be shared directly with other actors;
- the actor logic is executed asynchronously on reception of a message; and, upon invocation, can mutate local state and send messages to other actors, sometimes referred to as “acquaintances”;
- actors can instantiate other actors and manage their life-cycle.

This message-passing based model has proven successful in describing concurrent computations without explicit, barrier-based synchronization.

The actor model has been encoded in various programming languages, such as Erlang[43], and libraries to express actor-based systems in general-purpose

Figure 3.1: Actor system
3. Background


3.2 Dataflow

Dataflow, originally introduced in [41], is a paradigm and computational model that expresses computations as a directed graph (figure 3.2):

- **nodes** (vertices) represent units of computation, the operators, originally restricted to deterministic, side-effect-free functions;

- **edges** edges identify channels along which pieces of data can flow; these can be anything from well-structured tuples to data-frames for DSP applications.

The computational model is particularly well-suited to express data-parallel, concurrent computation on a stream of incoming data. This led to its popularity for both real-time digital signal processing[31] and real-time analytics and stream processing for “big data” applications.

It is interesting to note how the dataflow model can be seen as a restriction of the actor model where:

- **operators** are limited **actors** that can be expressed via a function that can act on incoming data and, in some systems, read and write local state; dataflow operators are typically restricted from instantiating other operators;

- message exchanges between operators are only permitted on the dataflow graph edges.

Recent, well-known dataflow frameworks for data-analytics include Dryad[28], Spark Streaming (DStreams)[44], Flink[3] and Storm[4]. These systems expose a dataflow programming model, often in the form of a language-embedded DSL, and provide a runtime and scheduling environment for distributed execution.

Dataflow-based systems such as Tensorflow[19] are also becoming popular for machine-learning applications.

3.2.1 Message delivery

Message delivery semantics are different on a system-by-system basis, but physical computation edges are often implemented as FIFO channels. These
3.3. Timely Dataflow

Semantics affect how a downstream operator should expect to receive tuples generated by the upstream topology.

**Spark Streaming** [44] provides exactly-once semantics and emulates stream processing by performing a batch-like computation on “micro-batches” of the input data streams: operators are conceptually presented with a complete view of their inputs (“RDDs”) when scheduled.

**Storm** [5] provides best effort, unordered delivery and, optionally, at-least-once delivery out of the box.[6] Exactly-once semantics can be achieved via an alternative transactional API, Trident[7].

**Flink** [3][9] ensures exactly-once delivery in FIFO order. A checkpointing mechanism that relies on punctuation[42] is the basis for synchronization and fault tolerance in this system.

3.2.2 Operator model and execution strategies

Each system supports a different level of expressivity for the operator logic ranging from side-effect-free, deterministic transformations to completely custom logic that can either be invoked in response to a message being received, in a round-robin fashion, or via preemptive multi-threading.

**Spark Streaming** [44] emulates a streaming computation in a batch system by micro-batching the input and performing the batch-like computation on each micro-batch - one at a time. It supports[30] stateful operators in the form of functional-style incremental aggregation (with functions that receive previous state and return the updated state after processing a record);

**Storm** [5] executes an operator’s (Bolt) main method (`execute`) in response to a new input tuple; the logic is permitted to perform arbitrary computation and send output tuples to an “OutputCollector”. Worker threads can each drive one or multiple instances of an operator[17] and these are also allowed to freely spawn additional threads for asynchronous processing.

3.3 Timely Dataflow

Timely Dataflow[36] is a Rust[14]-based implementation of the cyclical dataflow programming model proposed by Naiad[38]. Its “progress tracking” mechanism enables the user to express fine-grained synchronization barriers while the system itself imposes almost no artificial synchronization: operators can ask to be notified once they have received all messages belonging to a certain logical grouping (identified by a “timestamp” associated with a set of tuples).

Unlike other systems, Timely Dataflow supports cycles in the dataflow topologies and complex timestamps. This makes it possible to express (nested) iterative computation via a feedback loop by encoding the loop counter in the timestamp.
3. Background

![Diagram of computation](image)

Figure 3.3: Topology of the computation defined by the code in Listing 3.1. Tuples are tagged with timestamps of the form \((t_1)\) and in the outer scope and \((t_1,t_2)\) in the inner scope.

### 3.3.1 Progress tracking

This section introduces the core concepts of the progress tracking subsystem, a core primitive that enables fine grained synchronization at each operator and permits parallel processing of data belonging to various logical groups (such as input time intervals).

In Timely Dataflow tuples are shipped within small batches that carry additional metadata: a logical timestamp is always associated to a batch of tuples and this information is used to keep track of the computation’s overall progress. When traversing an operator, input tuples can turn into output tuples which have an equal or greater timestamp.

Operators of the computation are organized in possibly nested scopes which enclose logical computation subgraphs, and the timestamps in each scope have a corresponding nested structure: often timestamps are tuples of integers, indicating an integer timestamp in each of its nested scopes. Scopes can have entry and exit nodes that respectively append or remove timestamp coordinates: these vertices adjust the metadata of tuples traversing them without affecting the contents. For example, the timestamp \((a_1,a_2)\) for a tuple entering a subscope would become \((a_1,a_2,a_3)\); conversely, the last coordinate would be stripped off when exiting the subscope.

Listing 3.1 and figure 3.3 show a simple topology that showcases nested scopes and the corresponding timestamp structure. In this case, the inner timestamp is used to keep track of the number of times a tuple has crossed the feedback edge.

Timely dataflow’s progress tracking machinery aggregates information about pending timestamps, corresponding to unconsumed messages and incomplete work in the system. This allows operators to understand when they have received all records that are annotated with a given timestamp. This information is aggregated into a per-input frontier at each operator. A frontier \(F = \{t_1, \ldots, t_x, t_i \in T\}\) implicitly defines the set \(S_F\) of logical timestamps that could be attached to future messages that could reach the operator’s inputs,

\[
S_F = \{ t_x \mid \exists t_f \in F \mid t_x \geq t_f \}.
\]

Conversely, any timestamp \(t_i \notin S_F\) is guaranteed to never appear at the operator’s input in the future.
3.3. Timely Dataflow

Listing 3.1: A simple timely dataflow computation showcasing nested scopes and feedback edges.

```rust
let mut input = root.scoped::<u64, _>::move | outer_scope | {
    let (input, input_stream) = outer_scope.new_input();

    let stream = input_stream.map(|x| x + 1);

    outer_scope.scoped(move | inner_scope | {
        let (handle, cycle) = inner_scope.loop_variable(5, 1);
        let forward = stream.enter(inner_scope).concat(&cycle);

        let times_ten = forward.map(|x| x * 10);
        times_ten.connect_loop(handle);
        times_ten.leave();
    });
}
```

Typically, this information is used to provide a high level signal, a notification at some logical timestamp \( t_i \) that an operator may assume that it is never going to see any other message at any timestamp \( t_x \leq t_i \). Timely provides an operator, `probe` that acts as a pass-through for tuples, but also permits user-level code to inspect progress tracking information.

3.3.2 Operator scheduling

Timely Dataflow computations are described by a logical computation graph composed by nodes - the operators - connected by edges - the communication channels.

At runtime each worker instantiates a full copy of the topology: there is an independent instance of each operator on each of the workers. This results in a fully symmetric and peer to peer system. Data sharding is performed by explicitely providing a sharding function for operators that need to perform aggregation or other key-based operation. These sharding functions distribute the tuples between buckets, which are then assigned to the available workers. For this reason, logical edges incident to an operator that declares a sharding strategy are instantiated as a full interconnect between workers.

A single thread is allocated to each worker and a simple cooperative scheduling approach is employed to run all the operator instances. Each worker, independently from the others and without synchronization, performs round-robin scheduling by calling into each operator’s control logic. When invoked, an operator has a chance to read any outstanding output, process any available notification and produce any amount of output.

3.3.3 Data layer

The unit of tuple shipping in Timely Dataflow is a chunk of allocated memory containing 1024 tuples: the actual memory footprint of these buffers depends on the data size for the tuple and does not need to be statically known. Operators read chunks of data from their input queue and can either forward
the obtained allocated chunk to the output or allocate new output chunks. This behaviour is irrespective of whether the operator ports are connected to local or remote workers.

A communication subsystem provides the necessary abstraction for tuple delivery. Tuple shipping is performed in-memory, with no serialization step, for worker-local communication; on the contrary, inter-process communication relies on a single TCP channel per worker pair: logical channels are multiplexed on this communication fabric. Serialization and deserialization is performed via Abomonation[33], an high performance serialization library that is designed to minimise copying.

All tuples within a single chunk are annotated with the same timestamp, \( t \), stored in the chunk header. No timestamp ordering is imposed on the shipped chunks and there are no synchronization barriers imposed on tuple delivery by the progress tracking mechanism: operators are required to handle asynchronous delivery of chunks with timestamps in non-monotonic order. Conversely, delivery of tuples affects the availability of notifications within an operator: when there are messages for \( t \) still on the input queue, no notifications for \( t \) can be delivered.

3.3.4 Granular synchronization

The lack of explicit synchronization between the progress tracking mechanism and tuple shipping differentiates Timely Dataflow from other dataflow systems. Spark Streaming[23] completes processing of all tuples pertaining to a certain timestamps before continuing with the next. Conversely, systems like Flink[3] rely on punctuation[42] for synchronization purposes: delivery to a specific operator is performed in timestamp order and special messages mark the boundaries of each timestamp.

This fine-grained tracking of granular units of progress, performed by Timely Dataflow without imposing constraints on the ordering of tuple shipping and delivery, provides greater flexibility for the user to only introduce synchronization points where necessary, minimizing resource waste due to system-imposed barriers. Operators can independently inspect their local frontier \( F \) at their inputs to determine whether they have seen all possible tuples annotated with a certain timestamp in order to perform operations that require a consistent or complete local view of the computation up to a certain point. Conversely, operators that can operate on tuples in isolation can do so without imposing artificial synchronization points on the system.

3.4 Experimental configuration

Unless otherwise specified, experiments described in this document are performed on a cluster of Intel Xeon E5-2650 @ 2.00GHz with 16 physical cores and connected by a 10Gbps link; memory consumption (rss) on each node is limited to 32GB.

The dataset in use is the LiveJournal social network graph[10]: statistics are listed in table 3.1.
3.4. Experimental configuration

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>4847571</td>
</tr>
<tr>
<td>Edges</td>
<td>68993773</td>
</tr>
<tr>
<td>Nodes in largest WCC</td>
<td>4843953 (0.999)</td>
</tr>
<tr>
<td>Edges in largest WCC</td>
<td>68983820 (1.000)</td>
</tr>
<tr>
<td>Nodes in largest SCC</td>
<td>3828682 (0.790)</td>
</tr>
<tr>
<td>Edges in largest SCC</td>
<td>65825429 (0.954)</td>
</tr>
<tr>
<td>Average clustering coefficient</td>
<td>0.2742</td>
</tr>
<tr>
<td>Number of triangles</td>
<td>285730264</td>
</tr>
<tr>
<td>Fraction of closed triangles</td>
<td>0.04266</td>
</tr>
<tr>
<td>Diameter (longest shortest path)</td>
<td>16</td>
</tr>
<tr>
<td>90-percentile effective diameter</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Table 3.1: LiveJournal social network graph dataset statistics
4.1 The flatMap operator

Due to the generality of the dataflow programming model, input and output rates for operators need not be tied by a simple, statically determined function; this prevents the construction of a static schedule which guarantees that the number of tuples in-flight on each communication channel is kept small.

We introduce flatMap, an existing[36] built-in Timely Dataflow operator, as a basic example of an operator that expresses a complex relation between input and output rate that is dependent on the content of the incoming tuples. flatMap is defined in terms of a function $I \rightarrow \text{Iterator<Item}=O$ that transforms an incoming tuple into a sequence of records on the output. For example, the operator instantiated by the code in listing 4.1 repeatedly reads an integer $n$ and generates $n$ output records.

Listing 4.1: A flatMap instantiation that repeatedly reads an integer $n$ and outputs the integers from 0 to $n-1$

```
incoming_stream.flatMap(|x| (0..x))
```

4.2 Input/output rate

Let $N_{in}(t, t')$ and $N_{out}(t, t')$ be the number of tuples respectively read and written by an operator during the (wall-clock) time interval $[t, t')$, we can define

$$O_{rate}(t, t') = \frac{N_{out}(t, t')}{N_{in}(t, t')}$$

to describe the relationship between the ingress and egress rate of the operator.

We cannot describe $N_{out}(t, t')$ as a function of $N_{in}(t, t')$ for the flatMap operator described in the previous section because $N_{out}(t, t')$ is the sum of the integers read in the interval $[t, t')$ and not a function of $N_{in}(t, t')$. 
4. Computation edges and the backpressure problem

4.3 Rate imbalance

In order to support a varied set of applications, dataflow systems such as Timely Dataflow cannot simply rely on the load shedding strategy employed by network routing systems. For this reason, when one of the operators produces data faster than its downstream operator is able to process it, data needs to be buffered on the edge connecting them.

In general, a basic buffering strategy can result in a significantly increased memory usage and instabilities if the processing rate imbalance continues and the buffer space required leads to resource exhaustion.

Operators that generate a large amount of output on their egress channel(s) in response to a small amount of input from its ingress channel(s) have a large instantaneous $O_{rate} \gg 1$. In particular, operators like `flatMap` have no static bound on their instantaneous rate and, as a result, may send an unpredictable and unbounded amount of data.

An $O_{rate} = 1$ identifies operators that typically emit a single tuple in response to a single input message and $O_{rate} < 1$ characterizes operators that perform filtering or aggregation.

4.4 An unstable computation, dataflow-join

To better illustrate the issue and as support for the examples and experiments in the remainder of the document we introduce a specific computation that presents non-stable behaviour in unmodified Timely Dataflow.

We consider an existing dataflow implementation[34] of a specialization of Ngo et al.’s GenericJoin[40] algorithm for relational joins.

4.4.1 Algorithm and dataflow graph

Our experiments consider the enumeration of 3- and 4-cliques in a directed graph $(N, E)$, viewed as a repeated self-join of the relation of edges

$$A_{ij} = \{(a_i, a_j)| a_i, a_j \in N, (a_i, a_j) \in E\}$$

The specific implementation of GenericJoin[35] operates attribute-by-attribute and produces intermediate streams of tuples that represent the $k$-length prefixes of the join results. This is initially the stream of the singleton tuples representing graph vertices, and for each $k$ the stream is extended to the stream of length-$(k+1)$ prefixes as follows:

- each relation declares how many records ("proposals") match the prefix,
- the stream of length-$k$ proposals that match the relation with the fewest proposals is emitted as a stream, and
- the proposals are intersected with the other relations to filter out tuples not supported by all relations in the join.
4.4. An unstable computation, dataflow-join

For example, a tuple \((a_1, a_2) \in A_{12}\) could indicate a 2-clique (an edge) and would be extended to a 3-tuple \((a_1, a_2, a_3)\) by having the relations \(A_{23}\) and \(A_{13}\) propose values for \(a_3\): for each \((a_1, a_2)\) the proposals for \(a_3\) come from the neighbors of the vertex \((a_1 \text{ or } a_2)\) with lower degree (thus, respectively, either from \(A_{23}\) or \(A_{13}\)). They are then intersected against the neighbors of the other vertex to filter out proposals that do not support a 3-clique.

Figure 4.1 depicts a simplified version of the logical dataflow topology for 3-clique enumeration. Each edge in the logical graph corresponds to a full interconnection between physical workers, each of which perform some fraction of the work for each operator. The edges are annotated with the tuples that cross them, at any point in the computation, as a result of injecting the two singleton tuples \((a_{11})\) and \((a_{12})\).

4.4.2 Operator behaviour

As shown in figure 4.1 operators \(P_a\), \(P_{b1}\) and \(P_{b2}\) propose viable candidates for the join result by looking up edges in the relation \(A_{i,j}\).

\(P_a\), upon reading tuple \((a_{11})\), outputs all edges \((a_{11}, a_{2x}) \in A_{12}\). Similarly, upon reading tuple \((a_{11}, a_{21})\), operator \(P_{b1}\) outputs all \((a_{11}, a_{21}, a_{3x}) \in A_{23}\).

These propose operators respond to a single input tuple with a sequence of output records, a behaviour similar to flatMap and characterized by a statistically unbounded and potentially large \(O_{rate}\). On the LiveJournal social graph dataset, \(P_a\) produces up to 20293 tuples in response to a single input record since vertex 10009 has degree 20293.

Conversely, intersect operators filter down their input to just the tuples that satisfy the relation being considered: in the example, operator \(I_1\) removes \((a_{11}, a_{22}, a_{33})\) because \((a_{22}, a_{33}) \notin A_{23}\). This operator type is characterized by \(O_{rate} \leq 1\).

An uncontrolled, naïve execution schedule may attempt running operators \(P_a\), \(C\), \(P_{b1}\) and \(P_{b2}\) to completion before the intersect steps \((I_1\) and \(I_2\)) start reading tuples. This would result in the entirety of the intermediate state produced by the two propose steps to be buffered on the edges from operators \(P_{b1}\) to \(I_1\).
In our example, all 3-tuples on the edges connecting \( P_{bi} \) and \( I_i \) would occupy buffer space at the same time, even though the intersect steps would have been able to process input tuple-by-tuple since they only perform stateless filtering. This buffered data corresponds to all 3-tuples \((a_1, a_2, a_3)\) such that \((a_1, a_2) \in A_{12}\) and either \((a_1, a_3) \in A_{13}\) or \((a_2, a_3) \in A_{23}\), depending on the routing performed by the count step. The cardinality of this set can be enormous and may require an extremely large amount of buffer space.

Conversely, a hypothetical memory-optimal single-threaded schedule for the computation would limit all the input operator buffer sizes to a single tuple: for example, \( P_1 \) would transfer control to \( C \) and \( P_2 \) immediately after producing tuple \((a_{11}, a_{21})\) and \( P_2 \) would, in turn, yield to \( C_i \) with \((a_{11}, a_{22}, a_{31})\) and so on. Such an approach, while minimizing memory usage, would, in turn, starve most workers in a distributed setting.

### 4.5 Channel size as a system metric

Instrumentation to measure buffer sizes for logical channels in the computation was put in place to diagnose unstable computations and observe the buffering behaviour. Buffer sizes are measured as the number of items on the dataflow edges. Because it represents allocated memory, this is a valid proxy for memory consumption and has the benefit of more accurately reflecting the unit of batching that Timely Dataflow employs for allocation and shipping: by default, tuples are stored and shipped in 1024-element buffers.

Figure 4.2 shows a full trace of the buffer sizes for various logical edges in a representative execution of the topology (depicted in figure 4.1) on the livejournal graph dataset. The experiment was performed with 8 workers on a single node but is representative of other configurations. The charts on top and on the bottom show the buffer sizes for the edges on the two distinct paths that tuples can take as a result of the partitioning performed at operator...
C. The fact that each incoming queue for the various operators grows and shrinks only once during the entire trace reveals that the default execution results in each operator consuming the entirety of its inputs and producing the full output in a single invocation: it can be seen how tuples flow in the topology as each operator depletes its upstream queue and buffers the entirety of its output on its outgoing edge. Additionally, the round-robin execution strategy is revealed by the fact that operator \( P_{b_2} \) does not begin consuming input until operators \( P_{b_1} \) and \( I_1 \) have terminated.

This confirms that an uncontrolled execution of the topology in Timely Dataflow corresponds to the naïve execution schedule described earlier. Such execution strategy, while advantageous in terms of scheduling efficiency, causes the entirety of the intermediate results to be buffered on the dataflow edges.

### 4.6 Summary

Both the unpredictability and the unboundedness of the relationship between an operator’s input and output rate, \( O_{rate} \), are the cause of be significant challenges for effective resource management:

- large instantaneous \( O_{rate} \) can result in a large amount of data to be buffered on an egress edge of the operator;
- unpredictable \( O_{rate} \) for operators across the computation makes it impossible to devise static execution strategies that result in bounded resource consumption;

A simple metric, buffer size of logical edges, has proven useful to analyse the behaviour of Timely Dataflow on a computation with multiple operators characterised by large rate imbalance and to diagnose memory consumption issues.

In the next chapter we will explore which flow-control techniques other systems employ to deal with rate imbalance.
In order to prevent memory exhaustion, most modern systems employ dynamic flow control techniques, often mandated by the most common use case for the system.

5.1 Source backpressure

Storm[4], Heron[29] and Spark Streaming[23] employ variations of a technique (depicted in figure 5.1) that relies on overflow signals from various components of the system to limit the data rate of the most upstream data sources, typically the inputs to the computation. This technique is effective for computations where the cause of resource exhaustion or operator overload is a significant surge in the ingress rate: this is frequently the case for big-data analytics workloads in which the heightened ingress rate is caused by external events. As an example, computations that analyse datacenter log events may experience surges caused by exceptionally high traffic or due to a large amounts of log events generated by a system failure.

Figure 5.1: Source backpressure. An overload signal at any operator in the topology (represented by its incoming buffer being full or reaching a threshold) triggers a reduction in ingestion rate at the computation source. Input/output rates for internal operators are unaffected.
5. Known Techniques

![Diagram](source)

Figure 5.2: Effect of increased output rate for internal operators in topologies with source back-pressure. A spike in the output rate for operator B causes a large number of tuples to be buffered on the edge to A; however, the overload signal only affects the "source" and does nothing to address the root cause: the heightened output rate of B.

5.1.1 Systems that rely on source backpressure

**Spark Streaming** [23] emulates a streaming computation in a batch system by micro-batching the input and performing the batch-like computation on each micro-batch - one at a time. In this system the maximum processing rate is estimated by looking at end-to-end latency, i.e. the time it takes to complete processing for a micro-batch. This information is used to choose an acceptable ingestion rate via a pluggable mechanism and backpressure is applied to computation inputs. By default, the input rate is dynamically adapted via a PID controller algorithm[11].

**Heron** [29] is Twitter’s internal implementation of **Storm**. In Heron, TCP-based back pressure could not be implemented naively because multiple logical channels are multiplexed on the same TCP channel. The "spout backpressure" approach was chosen because of being relatively easy to implement when compared to edge-by-edge back pressure, described in the next section. "Spout backpressure" is quite similar to the technique Spark Streaming uses: when operator instances slow down, worker-local computation inputs (spouts) are throttled immediately. An “initiate back pressure” message is sent to all other workers so that they can throttle their corresponding spouts. Another “all clear” message is sent when the throttling is no longer necessary. Message passing overhead is regarded as a potential disadvantage of this approach. Conversely, the ability to react quickly to ingress rate changes is mentioned as a supporting reason for this choice.

The Storm backpressure implementation[15] explicitly states that the implemented technique is similar to Heron’s and based on the Heron paper.

5.1.2 Robustness to increased output rate

While effective when dealing with an increased ingress data rate, this technique is not sufficient to protect the system from a surge in the number of messages produced by one of the operators within the dataflow graph (as opposed to a surge in ingress rate). A spike or other imbalance of the output rate for operator B in figure 5.2 when compared to its input rate, \( O_{rate} \gg 1 \)
can result in a large amount of data to be emitted on its output edge in a short amount of time. The system would interpret the growth of the edge buffer as an overload signal for \( A \) and attempt to slow down the most upstream data source. This flow control response is ineffective in this scenario because \( B \) is still permitted to output large amounts of data in response to minimal input and can result in resource exhaustion.

The dataflow join computation described in the previous chapter contains an operator, \( \text{propose} \), that can emit a large number of tuples in response to a single tuple in the input. A source backpressure mechanism would not remedy the risk of resource exhaustion in this topology because the \( \text{backpressure} \) signal is not applied to the operator that is emitting an excessive number of tuples.

### 5.2 Edge-by-edge backpressure

Apache Flink[3], IBM Streams[26], and Reactive Streams[13] employ a technique that relies on bounded-size buffers (or, equivalently, a credit-based system) for the dataflow edges. At each edge, the upstream operator can consume a predetermined number of credits to send data to the downstream operator. The latter can replenish the credit bucket by reading and processing tuples that were buffered on the edge. The limited number of credits guarantees that the buffer sizes for each edge in the system is kept bounded. In some system, the available space in the buffer is directly relied upon to determine how much data the upstream operator is allowed to send before being stalled or blocked. This technique closely resembles the flow control mechanism in TCP, where the \( \text{receiver window} \) represents the number of available credits for the channel.

When an operator processes data at a rate smaller than the arrival rate on one of its incoming edges, its upstream operator will consume credits faster than they are replenished: once these are depleted, the upstream operator is forced to block or decrease its processing rate in order to match the rate at which the downstream operator is processing data. This \( \text{backpressure} \) signal is propagated upstream, as operators are required to slow down to match the rate of the slowest operator in their downstream subgraph.

Figure 5.3 depicts a topology in which operator \( A \) is the bottleneck. The de-
5. Known Techniques

pletion of transmit credits for upstream operators (or, equivalently, depletion of available space in bounded-size buffers) affects the processing rate of operator B which, in turn, will cause C and D to slow down. If the rate imbalance at A persists, the backpressure signal would further propagate upstream.

Effective operation for this technique relies on a system-level mechanism that propagates information upstream on a backpressure graph which corresponds to the full dataflow graph with the edges inverted. Additionally, because of the required information propagation, correct behaviour is dependent upon a system-wide implementation that imposes the necessary constraints on all operators in the topology.

The edge-by-edge backpressure technique can guarantee bounded memory consumption, proportional to the number of available transmit credits within the system. A communication overhead is imposed by the additional information transferred on the backpressure graph and implementations need to manage inefficiencies due to fragmented scheduling: in fact, if only a small number of credits becomes available at a time, operators may be scheduled in rapid succession to process very small amount of data, which may result in a large cumulative effect even for small scheduling overhead.

Apache Flink [21] employs blocking queues between operators. If a receiver buffer is full, the sender is blocked. For edges that cross the network, there are separate sender and receiver blocking queues, and a watermark mechanism is used to avoid overflowing the receiver buffer: no data is read from the sender queue and put on the wire until the receiver buffer goes below the threshold. Flink also retains a buffer pool whose buffers can be temporarily assigned to channels that experience short bursts of increased data rate. SPL (IBM Streams)[27] employs blocking queues between operators, similarly to Flink.

Akka Streams [2] implement the Reactive Streams specification[13]. Reactive Streams employ a credit based mechanism: the receiver (downstream operator) can advertise additional capacity in the incoming buffers by sending request(n) messages to an operator upstream: these enable the upstream sender to send n new messages. All processing of the back-pressure signal is performed asynchronously and does not block the sender. The receiver controls its buffer size, which is typically bounded or static.

5.2.1 Liveness in cyclical topologies

The edge-by-edge technique can present issues when applied to dataflow topologies that allow cycles in the computation graph. Cycles in the dataflow graph also appear in the backpressure signaling graph. If no special precautions are taken, the overload of any operator within a cycle can trigger propagation of the backpressure signal and, if the overload scenario does not clear immediately, the overload signal may travel along the entirety of the cycle in the backpressure graph and effectively cause the overloaded operator itself to block, waiting for additional credits or buffer space before continuing. This scenario results in a permanent deadlock, as the overloaded operator would only be able to free up credits by processing tuples from its ingress edge but
5.3. Summary

In this chapter we have surveyed the benefits and pitfalls of various techniques to limit resource usage and prevent resource exhaustion in existing streaming systems. Operator data rate imbalance and cycles in the topology present significant challenges and are not fully addressed by these techniques: in particular,

- in the presence of operators with significant rate imbalance (large or unbounded $O_{rate}$) it is insufficient to only handle heightened ingress rate to the system;
- fine-grained flow control at the computation edges is a good general-purpose technique, but its focus on local behaviour presents issues for cyclical topologies and may result in inefficiencies due to the latency in signal propagation.

Figure 5.4: Deadlock scenario and differentiated queues. On the left: in a computation with cycles, the backpressure signal originated from $A$ can travel around the cycle and cause $A$ itself to stop, resulting in a permanent deadlock. On the right: differentiated input buffers at the feedback merge point can help preventing the deadlock by prioritizing data from the feedback edge.

it is blocked by a lack of available credits for its egress edge. The left-hand side of figure 5.4 shows how the overload of operator $A$ results in a back-pressure effect on operator $C$, which in turn forces $B$ to slow down. Credit depletion for the ingress edge to $B$ results in $A$ being blocked and unable to release resources.

An unresolved issue[22] in Flink’s issue tracker states how “[...] during increased load (or even short bursts) the increased amount of records in transit inside the loop can cause the limited number of input/output buffer-pools to fill up and deadlock the system”. The documentation for Akka Streams[18] warns about the risk of deadlocks in cyclical topologies and suggests selecting a preferred input at the cycle merge point (operator $C$ in the example): in this configuration, processing of incoming data from the feedback edge is prioritized, facilitating the release of resources and reducing the deadlock risk.

Separate queues for each operator input at the feedback merge point, as depicted on the right-hand side of figure 5.4 can be effective in limiting the deadlock risk in scenarios where the operator overload is caused by a surge in the rate of tuples entering the feedback loop. Conversely, if one of the operators in the cycles can emit data at rate higher then its input rate, it can deplete all the available buffer size within the cycle and still cause a deadlock.
Chapter 6

Flow-control

The programming model for Timely Dataflow supports a wide variety of programs ranging from real-time analytics to highly efficient query processing. Some interesting computations, such as the one described in the previous section, require a flow control and scheduling mechanism to be executed reliably and efficiently on the engine. A mechanism for resource-aware scheduling would enable the user to express this wider variety of programs with better resource management guarantees.

In this chapter, we summarize why the existing flow control techniques are either not naively applicable to the Timely Dataflow system or would not achieve the goal of increasing the expressive power of the paradigm.

**Source backpressure** (section 5.1) can deal with increased ingress rate in real-time analytics systems but is ineffective in resource-limiting computations that contain operators that have unpredictable or unbounded rate imbalance.

**Edge-by-edge backpressure** (section 5.2) introduces the risk of deadlock in cyclical topologies which enable iterative computations to be expressed in the programming model. In addition to the difficulties discussed in the previous chapter, this technique presents further challenges when considered as a resource-management method for Timely Dataflow. As discussed in section 3.3.3, tuple batches are not ordered by timestamp when shipped on computation edges and, as a result, batches with lower timestamps may be queued behind tuples with higher timestamps at the feedback merge point. If any operator in the cycle needs to wait for the delivery of all tuples with a certain timestamp before proceeding with the consumption of further input, the system may be unable to release resources and deadlock. An approach that relies on differentiated, per-timestamp queues can be envisioned: it carries significant implementation complexity and does not intrinsically provide better guarantees because the number of active timestamps and associated buffers is not guaranteed to be small or bounded.
6. Flow-control

6.1 Flow control strategy

Due to these difficulties with implementing existing, well-known flow control techniques in Timely Dataflow, we elect to devise a scheduling and flow control pattern that leverages the existing primitives and abstractions made available by the system. In order to ensure stability, the number of in-flight tuples within the computation needs to be kept under control. We introduce a computation pattern and supporting library that enables fine-grained control of the amount of tuples that are allowed into a certain computation section. Additionally, to remain true to the spirit of the system, we strive to avoid introducing unnecessary synchronization points.

6.2 System Primitives

The resource-management pattern is built upon the following core primitives of Timely Dataflow:

1. The ability to organize the computation graph with nested scope that introduce finer-grained synchronization units (timestamps).

2. The fact that progress tracking information does not flow along dataflow edges, and an operator can inspect the progress state of other operators, even if distant in the dataflow graph. The probe operator exposes this information to the user.

3. The expressive power available to write complex custom operator logic and the fact that operators are scheduled in a round-robin fashion, even when no tuples are queued on their inputs.

6.2.1 Scopes

The Timely Dataflow programming model enables intervening on the structure of the logical timestamps to convey further information on the state and progress of a well-defined subgraph of the computation.

By designating a section of the dataflow graph as a scope, timestamps for tuples crossing the scope boundary can be manipulated: a new coordinate can be appended, transforming a $k$-element timestamp $(t_1, \ldots, t_k)$ into a $(k+1)$-element timestamp $(t_1, \ldots, t_k, t_{k+1})$. Figure 6.1 shows a basic example: tuples are annotated with timestamps of the form $(t_1)$ outside the subscope and $(t_1, t_2)$ inside the subscope. The enter and leave operators are tasked with forwarding the tuples that cross the boundary while transforming the timestamps as necessary; the progress tracking system is aware of the relationship between timestamps inside and outside the scope and is able to aggregate information about finer-grained timestamps in the subscope and report them to the rest of the dataflow graph and vice-versa. In most cases, if a tuple with timestamp $(t_1a, \ldots, t_ka, t_{k+1})$ can still appear at the ingress edge of the enter operator, operators within a subscope have to assume that tuples tagged with a timestamp of the form $\forall t_{k+1} \in (t_1a, \ldots, t_ka, t_{k+1})$ can still appear at their inputs. Conversely, if tuples tagged with any timestamp of the form $\forall t_{k+1} \in (t_1a, \ldots, t_ka, t_{k+1})$ can still appear at the ingress edge of the exit opera-
6.2. System Primitives

Figure 6.1: Timestamp transformation at scope boundaries

tor, the remainder of the dataflow graph downstream of the subscope has to expect a tuple with a timestamp \((t_1, \ldots, t_k)\).

6.2.2 Probe

Timely Dataflow exposes progress tracking information to the user via the probe operator. When inserted in the topology, it exposes the local frontier at its input: it can be queried to determine whether a certain timestamp can still appear at its input, i.e., whether \(t_j \in F\) where \(F\) represents the frontier at the probe input, as described in section 3.3.1.

6.2.3 Scheduling and expressivity

Within a single worker, scheduling for operators is performed cooperatively on a single thread, in a round-robin fashion, as described in more detail in section 3.3.2. This means that operators are woken up periodically and given a chance to read input, send output or request notifications even when there are no outstanding notifications or tuples on the input.

Listing 6.1: Simplified interface available to operator logic.

```rust
/// Obtains the next batch of input tuples of type D and annotated with a timestamp T.
/// Returns 'None' if no data is available.
fn next_batch(input: usize) -> Option<(T, Vec<D>)>

/// Obtains the next notification, or None if not available.
fn next_notification() -> Option<T>

/// Sends a tuple annotated with timestamp t to the output.
fn send(output: usize, t: T, data: D) -> ()

/// Requests a notification for a certain timestamp.
fn notify_at(t: T) -> ()
```

Listing 6.1 contains a simplified summary of the interface available to custom operator logic. The only constraints imposed are those necessary to guarantee the correctness of the progress tracking protocol: a complete overview of these constraints is available in appendix A. Additionally, local operator state can be persisted between invocations.

This generality, the fact that operators are scheduled periodically and the ability to persist state between invocations enable the operators to regulate the flow of data from its input to its output.
6.3 The Faucet flow-control pattern

Faucet, the computation pattern we propose, wraps a subgraph of the computation by introducing an additional scope, an operator with control logic (the batcher) at the subgraph ingress edge, and a probe at the egress edge. Figure 6.2 shows the resulting topology.

The probe handle, that permits inspecting the frontier at the probe, is injected in the batcher, which uses that information to regulate the rate of admission of tuples into the controlled subgraph.

When entering a new scope, data has to cross an enter ingress operator that appends a coordinate to the logical timestamp. Thus, all data with outer timestamp \( \vec{t} = (t_1, ..., t_n) \) appears at the batcher’s input with timestamp \( \vec{\tau} = (t_1, ..., t_n, t_b) \). The new coordinate, \( t_b \) will be used to annotate batches of data in the inner topology in order to track their progress through the controlled subgraph.

The batcher processes its input and forwards small batches of the data for each incoming logical timestamp \( \vec{\tau}_e \) and annotates them with a batch counter \( t_b \), unique within the same \( \vec{\tau}_e \).

By inspecting the probe at the egress of the flow-control subgraph, the batcher is able to maintain a constant number \( N_{\text{batches}} \) of batches in-flight within the controlled subgraph. If the input data rate is higher than the rate at which the subgraph is able to process data, the batcher stops forwarding data and buffers it.

The number of batches \( N_{\text{batches}} \) and the number of tuples per batch \( B \) allowed in the controlled subgraph are configurable by the user. The reasoning behind this design choice is discussed in section 9.1.

It is important to note how, due to the structure of the pattern, this flow-control mechanism can be applied in a nested fashion to guard subgraphs or operators that present significant data rate imbalance. In most cases, the point of entry to a controlled subgraph is the operator that proves to be the cause for resource exhaustion and the probe is applied after the operator that reduces or filters the large intermediate results. This way, the pattern can ensure that the number of in-flight tuples is kept small for a subgraph in which early operators produce a large number of tuples and late operators consume them.
6.3. The Faucet flow-control pattern

Figure 6.3: A minimal computation (a.) without flow-control, (b.) with Faucet applied to the subgraph containing A and B, and (c.) with the batcher wrapping a flatMap operator. A has $O_{rate} > 1$ and B is a filter or aggregation, $O_{rate} < 1$

6.3.1 Support for flatMap operators

Chapter 4 showed how operators with unbounded output rate ($O_{rate} \gg 1$) may produce a large number of tuples in response to a small amount input. We would like to be able to control their execution in order to limit the number of tuples injected in the downstream topology.

For this purpose, the batcher can also support a custom transformation of the data traversing it as long as the operator logic can be expressed as a function from an incoming tuple to an iterator: a function $A \rightarrow$ Iterator<Item=B>. This makes it possible to express the logic of a flatMap operator directly in the batcher. With a lazy iterator, execution can be suspended after a set number of tuples has been produced and only resumed once the batch has reached the end of the controlled subgraph.

The instance of flatMap in listing 4.1, `.flatMap(|x| (0..x))`, would produce 1 billion elements in response to $x = 1'000'000'000$. However, if the iterator-based version of the batcher is in use, only $N_{batches} \times B$ elements are materialized from the iterator and execution is suspended until one of the batches reaches the end of the controlled subgraph.

6.3.2 Usage example

The topology depicted in figure 6.3.a contains operators characterised by input/output rate imbalance. In particular, A has $O_{rate} > 1$ and B has $O_{rate} < 1$. A naive schedule may attempt to buffer the entirety of the output of A on its output edge.

Faucet can be applied to limit the number of tuples that enter the subgraph of A and B as depicted in figure 6.3.b. When $O_{rate}$ for A is not extremely large, this setup is sufficient to control the amount of intermediate results being buffered on the internal computation edges and ensure that B is scheduled to consume tuples produced by A before the batcher injects another batch.

When A can be expressed as $C \rightarrow$ Iterator<Item=D> and if its $O_{rate} \gg 1$, the topology in figure 6.3.c will cause the execution of A to be suspended as necessary to limit the number of tuples injected in the controlled topology.

Configuration 6.3.b addresses the scheduling and memory management concerns for operators with unpredictable but limited $O_{rate}$, while 6.3.c can deal
6. Flow-control

Figure 6.4: Dataflow-join topology to enumerate 3-cliques with the Faucet pattern applied.

with operators that have unbounded instantaneous $O_{rate} \gg 1$.

6.3.3 Real-world example and effects on scheduling

Figure 6.4 depicts the dataflow join topology to enumerate 3-cliques described in section 4.4 with Faucet applied to guard the two nested subgraphs originating from the two `propose` steps: these are characterised by $O_{rate} > 0$ but, in the LiveJournal social graph dataset, the number of proposals cannot exceed the maximum degree, which is 20293. Therefore $O_{rate}$ for $P_a$, $P_b1$ and $P_b2$ is bounded. For this reason it is not necessary to express the logic of the `propose` steps directly in the batcher.

The most downstream operators in the controlled subgraph are those in the `intersect` step, which has a significantly reduced egress rate when compared to the rate of input tuples ($O_{rate} < 1$).

In the diagram, edges are again annotated with tuples that cross them, and each of the black rectangles represents a batch of tuples tagged with a certain
6.4. A more complex topology

Thanks to additional *propose* and *intersect* steps the dataflow-join topology depicted in figure 6.5 builds upon the results of the 3-clique computation to enumerate 4-cliques.

Because operator $A_{12}$ (the first *propose* step) has a fairly large but bounded $\max(O_{\text{rate}})$, the Faucet pattern can be applied as shown in figure 6.6. Experiments in chapters 9 and 10 will show that this is a viable topology.

**Figure 6.5:** Logical topology for the dataflow-join computation to enumerate 4-cliques.

**Figure 6.6:** Logical topology for the dataflow-join computation to enumerate 4-cliques, with the Faucet pattern applied.

synthetic logical timestamp (in square brackets) and for which completion can be detected by the probe at the end of the flow-control block.

We assume $N_{\text{batches}} = 1$ and $B = 1$ for this example: the input tuple $(a_{11})$ is the only one allowed in the outer scope at first, as part of a batch with timestamp $(..., u_1)$. $(a_{12})$ will only be injected once processing for $(a_{11})$ is complete.

The inner instantiation of Faucet only admits $(a_{11}, a_{21})$ in the inner scope as part of the batch annotated with timestamp $(..., u_1, v_1)$. Once this batch is completed (resulting in no tuples on the output), $(a_{11}, a_{21})$ is admitted into the inner scope with timestamp $(..., u_1, v_1)$. It should be noted how a batch can only be marked as completed after the *intersect* operators, with $O_{\text{rate}} < 1$, have read and processed the tuples belonging to the batch.

The process repeats till exhaustion of the input. If $N_{\text{batches}}$ was 2, both batch $(..., u_1)$ and $(..., u_2)$ would be active at the same time in the outer scope; similarly, both $(..., u_1, v_1)$ and $(..., u_1, v_2)$ would both be active in the inner scope.

The example shows how delaying injection of tuples into a controlled sub-graph has the effect of limiting the amount of intermediate state being generated and stabilizing the buffer sizes.

6.4 A more complex topology
6. Flow-control

6.5 Timestamp semantics

The batch identifier is encoded as an integer appended as a timestamp coordinate: if the incoming logical timestamp has structure \( \vec{t} = (t_1, ..., t_n) \), the timestamp within a subgraph controlled with Faucet will be \( \vec{t} = (t_1, ..., t_n, t_b) \) where \( t_b \) is the batch identifier.

When nested, the pattern needs a separate coordinate for keeping track of batches within each nested subgraph: this results in multiple integer values being appended to the timestamp:

\[
\vec{t} = (t_1, ..., t_n, t_{b_1}, ..., t_{b_i})
\]

where \( t_{b_i} \) retains the batch identifier for each nested instantiation (\( i = 1..k \)) of the pattern. In the example in figure 6.4, \( u_1 \) and \( u_2 \) represent values for \( t_{b_1} \), the outer batcher identifier, whereas \( v_1 \) and \( v_2 \) represent values for \( t_{b_2} \) in the inner instantiation.

It is important to note how identifiers for batches in an instantiation are only unique within a single outer timestamp: let \( \vec{t}_e = (t_1, ..., t_n, t_{b_1}) \); \( t'_{b_2} \) identifies one batch within \( \vec{t}'_e \) and a different batch within \( \vec{t}''_e \), in the same way as \( t'_{b_1} \) identifies separate batches depending on the outer timestamp \( \vec{t}_e = (t_1, ..., t_n) \). Full timestamps are required to uniquely identify specific batches.

By default, in Timely Dataflow, a partial order for timestamps is defined as follows: let \( \vec{t}' = (t'_1, ..., t'_n) \) and \( \vec{t}'' = (t''_1, ..., t''_n) \),

\[
\vec{t}' \leq \vec{t}'' \iff \forall i, t'_i < t''_i
\]

Also known as the product partial order[39].

The system coalesces progress updates and only permits inspection of the antichain of the frontier at the input of the \text{probe} operator, that is \( F = \{t_1, ..., t_k : t_i \in T; \exists b, j : t_i \leq t_j \} \). This definition for partial order, while effective for expressing data dependencies in iterative computations, imposes an artificial ordering on nested batches.

In a computation with two nested Faucet subgraphs with inner timestamps of the form \( \vec{t} = (t_e, t_{b_1}, t_{b_2}) \) and for a certain \( \vec{t}'_e \), the presence of an in-flight batch \( (\vec{t}'_e, t'_{b_1}, t'_{b_2}) \) prevents any batch \( (\vec{t}'_e, t''_{b_1}, t''_{b_2}) \) where \( t'_{b_1} \leq t''_{b_1} \land t'_{b_2} \leq t''_{b_2} \) from being marked as complete.

In a nested setting, completion messages for inner batches will be delayed by the completion of inner batches belonging to a previous outer batch - even in the absence of a data dependency. This effectively limits the number of in-flight batches that can be kept tracked to \( N_{\text{batches}} \) instead of the theoretical \( (N_{\text{batches}})^k \), where \( k \) is the nesting depth.

Unwanted synchronization barriers are produced by this restriction, which also has the beneficial effect of retaining a roughly linear relationship between the \( N_{\text{batches}} \) parameter and the overall peak queue length and memory consumption, as shown by the experiments described in section 9.2.
6.6 Summary

We introduced Faucet, a flow control pattern for Timely Dataflow designed to cause minimal unwanted synchronization and overhead. When applied to computation subgraphs that can generate large amount of intermediate state, the pattern affects scheduling in order to stabilise and control buffering on the computation edges. Additionally, a specific configuration of the pattern can control the execution of operators with extreme rate imbalance ($O_{rate} \gg 1$) and an Iterator-based interface without the need to resort to multi-threading and blocking calls. The pattern can be applied in a composable fashion to control scheduling and resource consumption of multiple nested subgraphs.
Chapter 7

Interface changes to support Faucet

This chapter describes the changes to the Timely Dataflow operator interface that we implemented to support Faucet.

7.1 Progress tracking constraints on operators

Operators in Timely Dataflow typically perform computations and produce output in response to incoming data or when notified they’ve received all data for a certain timestamp $t$: the correctness of progress tracking is predicated on a set of constraints for the timestamps that can be produced on the output of an operator or for which notifications can be requested.

Originally, these constraints were encoded as follows at the operator interface: an operator is permitted to send or request notifications at any $t_o$ if:

- during the same invocation of the operator logic, it has received data or notification at some $t_i$, such that $t_o \geq t_i$;
- there’s an undelivered, outstanding requested notification for some $t_i$, such that $t_o \geq t_i$;

Precise semantics are more nuanced and a complete write-up is available in appendix A.

7.2 Progress tracking behaviour for the batcher

Timely Dataflow’s data layer will deliver tuples to operators as soon as they become available on a dataflow edge. For this reason, the batcher needs to be able to buffer such input within the operator and deliver it at an appropriate timestamp at a later invocation, in response to the out-of-band signal provided by the probe. To achieve this, it needs to retain the ability to send data at timestamps $(\overrightarrow{t}, t_{b1})$, for various $t_{b1}$, on multiple invocations of the operator logic and not only in response to incoming data or notification.

This desired behaviour was atypical for Timely Dataflow operators and could not be cleanly implemented within the constraints described. An initial prototype solution relied on ensuring that undelivered notifications for the timestamps of future batches were consistently re-introduced at the end of
7. Interface changes to support Faucet

Each operator invocation. This resulted in unclear logic and inefficient processing because the operator was forced to process all outstanding notifications and re-request some of them just to apply a filter on notification requests that were needed to retain the ability to act at later invocations.

7.3 Generalised constraints, capabilities

Notifications in Timely Dataflow are not core system primitives and are generated locally for each operator by relying upon system-level updates that are exchanged with the rest of the systems. The library code that wraps an operator’s logic coalesces the system-level progress updates, determines which notifications need to be delivered and prepares an updated summary of the local progress to be shared with the rest of the system.

We partially redesigned the operator interface in order to support a wider range of valid operator logic with a set of constraints that more closely matched the requirements of the progress tracking protocol.

The ability to send data at timestamp $t_c$ is encoded into a capability for $t_c$: a data structure that represents a permit to act at timestamp $t_c$. The concept of capabilities is borrowed from system security[32].

In the revised interface, capabilities for $t_c$ are obtained upon reading input or when processing notifications at $t_c$ and are required to be provided as evidence of the ability to act when attempting to send data or request notifications at some $t_o$, where $t_o > t_c$.

The revised constraints are encoded as follows:

1. a capability for timestamp $t_c$ represents a permit for an operator that holds the capability to send data and request notifications at timestamp $t_c$;
2. the operator receives a capability at $t_c$ when it reads data tagged with that timestamp or processes a notification for that timestamp;
3. retaining a capability at $t_c$ means that the operator retains the ability to send data at that timestamp even when, in a future invocation, it has not read data or received notifications at $t_c$.

7.4 Implementation

The Rust programming language provides convenient features for implementing this interface. Capabilities are encoded as structures parametrized by the local timestamp type $T$ as shown in listing 7.1.

```
Listing 7.1: Capability struct

struct Capability<T> {  
    time: T,
    // ...
}
```
7.5 Safety

Functions of the operator interface can return capabilities alongside the data being read or notifications being delivered. Additionally, endpoints to request further notifications and send data to the output channels can request a reference to a capability as evidence that the operator is permitted to act at the associated timestamp.

The listing 7.2 shows a simplified version of the resulting operator interface. It is important to note how `send` and `notify_at` require a capability as evidence that the operator logic is permitted to send data or request notifications at \( t \).

Retaining a capability may be simply accomplished by persisting its data structure across invocations. The `Drop` special-purpose trait\(^1\) is used to execute code when a capability is relinquished by the user. This code informs the operator wrapper that the user does not intend to retain the ability to act at the associated timestamp and such information should be forwarded to the progress tracking subsystem.

The ability to retain a capability is leveraged in the batcher operator, which stashes a capability for the timestamp that will identify the next batch to be sent. The capability will be then dropped once the batch has been sent, to inform the progress tracking mechanism that the frontier at downstream operators can advance beyond the associated timestamps: this will let the probe determine when the batch is completed.

7.5 Safety

By more explicitly encoding the constraints in a way that can be statically verified by the compiler, the capability-based operator interfaces prevents the user from writing code that would violate the operator contract. This is beneficial to reliability and correctness.

---

\(^1\)The `Drop` trait is used to run some code when a value goes out of scope. This is sometimes called a `destroyor`.\([16]\)
7. Interface changes to support Faucet

7.6 Summary

The implementation of user-level flow control required some interface changes to Timely Dataflow. These not only enabled a cleaner implementation for Faucet but also more clearly exposed the core primitives of the system; this resulted in a simplified operator contract and in additional safety via compile-time checks.
Chapter 8

Experiments

This chapter introduces the experimental setting that yields the results discussed in chapter 9 and 10.

8.1 Configuration

Due to the fact that progress tracking information propagation time will affect the delivery of the signal that a certain Faucet batch has completed and to ensure performance experiments are as close to real-world deployment as possible, all evaluations are performed in a distributed setting.

Experiments are performed by starting one process on each of two nodes; each process, in turn, spawns the same number of worker threads. Charts in the following chapters indicate the total number of worker threads.

The hosts are Intel Xeon E5-2650 @ 2.00GHz with 16 physical cores and connected by a 10Gbps link.

Memory consumption (rss) on each node is limited to 32GB in order to prevent the hosts from relying on swap space. Early experiments have proven how swapping results in extreme performance losses and failure in completing execution of the workloads.

The dataset in use is the LiveJournal social network directed graph[10] introduced in section 3.4.
Figure 8.1: Dataflow-join topologies for 3- and 4-clique enumeration with and without the Faucet pattern applied.

8.2 Topologies

The topologies in use for the experiments are shown in figure 8.1:

(1.a.) plain dataflow-join computation to enumerate 3-cliques, introduced in section 4.4;

(1.b.) dataflow-join computation to enumerate 3-cliques, with the Faucet pattern applied as described in section 6.3.3;

(2.a.) plain dataflow-join computation to enumerate 4-cliques, introduced in section 6.4;

(2.b.) dataflow-join computation to enumerate 4-cliques, with the Faucet pattern applied.
Chapter 9

Parameters

When devising an execution schedule, the trade-off between minimizing the peak message queue size between operators and the likelihood that no single worker is subject to starvation can have a critical effect on overall throughput, latency and resource usage. In this chapter we analyse the effects of parameter choice in Faucet and we show how large enough values for $N_{batches}$, the number of batches, and $B$, the batch size, achieve optimal performance for the pattern with limited effect on memory consumption.

9.1 Synchronization points introduced by the pattern

The batching approach employed by Faucet introduces a synchronization point at the subgraph input: the batcher will only forward additional tuples to the downstream operators when the existing in-flight batches have completed. The synchronization is performed via the progress tracking mechanism, which intrinsically requires inter-worker communication. For this reason, workers can be starved while (i) waiting for other workers to complete work on $T_S(t)$, (ii) waiting for the time it takes for the completion message to cross the network and reach the relevant probe.

Data skewness is a significant factor for (i), as uneven data distribution at the shuffle point would cause significantly different execution times at various workers for each chunks. Because of timely dataflow’s ability to handle multiple concurrent active timestamps at the same time, we can rely on this mechanism to minimise the effect of skewness and progress tracking propagation time.

By maintaining multiple concurrent units of synchronization active at the same time we can ensure that workers do not need to wait for each batch to complete before injecting the next. Additionally, if we assume that the sharding functions yield a statistically uniform distribution of data between the workers, we can rely on large batches to minimize the impact of imbalance in processing a chunk of data on different workers (stragglers).
9. Parameters

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\noalign{\smallskip}
$B$ & \multicolumn{4}{c|}{$N_{\text{batches}}$} \\
\cline{2-5}
\noalign{\smallskip}
\hline
\noalign{\smallskip}
400 & 1.26 & 1.27 & 1.19 & 1.09 \\
880 & 1.17 & 1.18 & 1.11 & 1.03 \\
1937 & 1.13 & 1.13 & 1.11 & 1.06 \\
4263 & 1.12 & 1.12 & 1.09 & 1.04 \\
9382 & 1.10 & 1.09 & 1.03 & 1.00 \\
20648 & 1.07 & 1.06 & 1.01 & 1.02 \\
45440 & 1.04 & 1.04 & 1.01 & 1.02 \\
100000 & 1.00 & 1.00 & 1.01 & 1.02 \\
\hline
\end{tabular}
\caption{Effect of $N_{\text{batches}}$ on runtime for dataflow-join with Faucet, 3-cliques (topology 1.b.), 4 workers. Times shown are normalized against the best execution within the same configuration.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\noalign{\smallskip}
$B$ & \multicolumn{4}{c|}{$N_{\text{batches}}$} \\
\cline{2-5}
\noalign{\smallskip}
\hline
\noalign{\smallskip}
400 & 1.26 & 1.27 & 1.19 & 1.09 \\
880 & 1.17 & 1.18 & 1.11 & 1.03 \\
1937 & 1.13 & 1.13 & 1.11 & 1.06 \\
4263 & 1.12 & 1.12 & 1.09 & 1.04 \\
9382 & 1.10 & 1.09 & 1.03 & 1.00 \\
20648 & 1.07 & 1.06 & 1.01 & 1.02 \\
45440 & 1.04 & 1.04 & 1.01 & 1.02 \\
100000 & 1.00 & 1.00 & 1.01 & 1.02 \\
\hline
\end{tabular}
\caption{Effect of $N_{\text{batches}}$ on runtime for dataflow-join with Faucet, 4-cliques (topology 2.b.), 4 workers. Times shown are normalized against the best execution within the same configuration.}
\end{table}

9.2 Concurrent batches, $N_{\text{batches}}$

Table 9.1 and table 9.1 show execution times for various values of $N_{\text{batches}}$ and $B$, normalized against the best execution time within the same configuration.

In our experiments, switching from $N_{\text{batches}} = 1$ to $N_{\text{batches}} = 2$ improves performance significantly by minimizing the adverse effect of stragglers which delay injection of future batches by inhibiting the completion of a previous batch. Notably, values of $N_{\text{batches}} \geq 2$ have limited effect on runtime, especially when running with a larger number of threads or with larger batch size $B$.

Tables 9.3 and 9.4 show memory consumption for various values of $N_{\text{batches}}$ and $B$.

In the topologies under test, that have two nested Faucet instances, the relationship between $N_{\text{batches}}$ and peak memory consumption is roughly linear, especially for executions where the batch size $B$ is large enough. This confirms the behaviour predicted by section 6.5: additional synchronization barriers limit the memory impact for larger values of $N_{\text{batches}}$.

In the following experiments on the effects of batch size $B$, we select $N_{\text{batches}} = 4$ as a reasonable default, to reduce the variable space.
9.3 Batch size, $B$

Figures 9.1 and 9.2 show the effects of batch size $B$ on total runtime and memory consumption, when $N_{\text{batches}} = 4$.

When a $N_{\text{batches}} \geq 2$ is selected, an exceedingly small value for $B$ causes dramatic performance losses likely due to inefficient processing and synchronization overhead.

For smaller values of $B$, performance improves significantly as the batch size

---

### Table 9.3: Effect of $N_{\text{batches}}$ on peak memory consumption (in MB) for dataflow-join with Faucet, 3-cliques (topology 1.b.), 4 workers

<table>
<thead>
<tr>
<th>$B$</th>
<th>$N_{\text{batches}}$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td></td>
<td>50.35</td>
<td>54.74</td>
<td>62.87</td>
<td>94.35</td>
</tr>
<tr>
<td>880</td>
<td></td>
<td>51.95</td>
<td>57.45</td>
<td>71.86</td>
<td>120.41</td>
</tr>
<tr>
<td>1937</td>
<td></td>
<td>52.09</td>
<td>63.19</td>
<td>89.87</td>
<td>157.89</td>
</tr>
<tr>
<td>4263</td>
<td></td>
<td>55.54</td>
<td>76.41</td>
<td>124.83</td>
<td>247.98</td>
</tr>
<tr>
<td>9382</td>
<td></td>
<td>65.63</td>
<td>106.01</td>
<td>145.44</td>
<td>313.41</td>
</tr>
<tr>
<td>20648</td>
<td></td>
<td>84.93</td>
<td>141.90</td>
<td>246.88</td>
<td>698.88</td>
</tr>
<tr>
<td>45440</td>
<td></td>
<td>134.11</td>
<td>200.10</td>
<td>408.15</td>
<td>1,097.56</td>
</tr>
<tr>
<td>9382</td>
<td></td>
<td>255.84</td>
<td>403.45</td>
<td>999.38</td>
<td>2,372.35</td>
</tr>
</tbody>
</table>

### Table 9.4: Effect of $N_{\text{batches}}$ on peak memory consumption (in GB) for dataflow-join, 3-cliques (topology 2.b.), 4 workers

<table>
<thead>
<tr>
<th>$B$</th>
<th>$N_{\text{batches}}$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td></td>
<td>0.51</td>
<td>0.52</td>
<td>0.55</td>
<td>0.63</td>
</tr>
<tr>
<td>880</td>
<td></td>
<td>0.52</td>
<td>0.54</td>
<td>0.58</td>
<td>0.81</td>
</tr>
<tr>
<td>1937</td>
<td></td>
<td>0.54</td>
<td>0.56</td>
<td>0.67</td>
<td>1.12</td>
</tr>
<tr>
<td>4263</td>
<td></td>
<td>0.56</td>
<td>0.62</td>
<td>0.90</td>
<td>2.36</td>
</tr>
<tr>
<td>9382</td>
<td></td>
<td>0.60</td>
<td>0.74</td>
<td>1.28</td>
<td>3.22</td>
</tr>
<tr>
<td>20648</td>
<td></td>
<td>0.70</td>
<td>1.02</td>
<td>2.32</td>
<td>6.19</td>
</tr>
<tr>
<td>45440</td>
<td></td>
<td>0.95</td>
<td>1.75</td>
<td>4.65</td>
<td>9.99</td>
</tr>
<tr>
<td>100000</td>
<td></td>
<td>1.70</td>
<td>3.57</td>
<td>9.54</td>
<td>23.47</td>
</tr>
</tbody>
</table>

---

Figure 9.1: Effect of batch size $B$ on total runtime and memory consumption, with $N_{\text{batches}} = 4$, in dataflow-join, 3-cliques (topology 1.b.).
9. Parameters

Figure 9.2: Effect of batch size $B$ on total runtime and memory consumption, with $N_{\text{batches}} = 4$, in dataflow-join, 4-cliques (topology 2.b.)

$B$ increases. Conversely, values of $B$ above a certain threshold have limited effect on total runtime and a roughly linear relationship with peak memory consumption. For the topologies under test, the threshold value is roughly $1000 \cdot C$ where $C$ is the number of worker threads. This is consistent with the fact that Timely Dataflow ships tuples in small batches of 1024-tuples (as described in section 3.3.3): to prevent starving and ensure efficiency, each worker should receive full buffers.

Graphs for other values of $N_{\text{batches}} \geq 2$ display a similar pattern.

The effects of batch size on memory consumption are negligible for small values of $B$. Larger values have a roughly linear effect on peak memory utilization (with a small slope) which still remains very small when compared to uncontrolled executions, as shown in chapter 10.

9.4 Summary

In Faucet, large enough values for $N_{\text{batches}}$ (the number of parallel batches) and $B$ (the batch size) result in close-to-optimal performance with a roughly linear relationship between $N_{\text{batches}}$ or $B$ and peak memory consumption, rendering the selection of optimal values for the pattern’s parameters less of a concern for the end user.
Chapter 10

Evaluation

The evaluation focuses on two main metrics: first, the ability of the technique to control the operator input queue sizes and, consequently, the overall memory usage; secondly, the amount of overhead introduced by the additional load on the progress tracking machinery and the loss of efficiency due to the synchronization imposed by the batching.

10.1 Buffer size

Figure 10.1 shows a full trace of the buffer sizes for various logical edges in a representative execution of the dataflow-join topology to enumerate 3-cliques (figure 6.4), with the Faucet pattern applied. The charts on top and on the bottom show the buffer sizes for the edges on the two distinct paths that tuples can take as a result of the partitioning performed at operator C.

Figure 10.1: Channel sizes for 3-clique dataflow-join with Faucet applied (topology 1.b.), 8 workers. \( N_{\text{batches}} = 4, B = 10000 \). The operator names refer to figure 6.4.
10. Evaluation

Figure 10.2: Overhead and peak memory footprint of Faucet when compared to a non-controlled execution, 3-cliques (topologies 1.a. and 1.b.). The left chart shows runtimes for maximum, minimum and average execution times of the flow-controlled topology with close-to-optimal parameters chosen for \( N_{\text{batches}} \) and \( B \). Values are normalised against the runtime of the non-controlled execution. An execution with \( N_{\text{batches}} = 2 \) and \( B = 2048 \) (lowest values for the parameters) resulted in 2.02 and was removed from the chart for clarity. The right chart shows peak memory consumption for the two topologies when run on various fractions of the input dataset, with 8 workers, and \( N_{\text{batches}} = 4 \) and \( B = 16384 \) for Faucet.

When compared with the trace for the uncontrolled execution in figure 4.2, this trace shows how Faucet is effective in limiting and stabilising the buffer sizes for the logical computation edges.

10.2 Overhead

In order to estimate the overhead caused by the additional synchronization and inefficiencies introduced by the Faucet pattern, we compare the total execution time of the computation with and without the pattern applied. We chose values for the parameters that achieve close-to-optimal performance based on the results of the experiments described in chapter 9.

Multiple experiments were performed with every combination of \( N_{\text{batches}} \) and \( B \) such that

\[
N_{\text{batches}} \in \{2, 4, 8\}
\]

and

\[
B \in \{1024 \cdot W_c \cdot x | x \in \{1, 2, 4, 8\}\}
\]

where \( W_c \) is the number of workers. This methodology was selected to show how a wide range of parameter choices result in similar runtime overhead.

To ensure that the non-controlled topology could reliably complete execution without incurring in resource overruns, for the 4-clique topology (2.a. and 2.b.) the search was limited to 4-cliques originating from the first 100'000 vertices, or about 2% of the full dataset.

The left charts in figures 10.2 and 10.3 show overhead as the ratio of total execution time with and without the pattern applied: we display minimum, maximum and average overhead for the various parameter choices.

In distributed settings the measured runtime overhead ranges from 10% to 30% depending on the topology and parameter choice.
10.3 Peak memory consumption

A primary objective of flow-control is to limit and manage resource consumption. We evaluate its effectiveness by comparing peak memory consumption for un-controlled executions with topologies with the Faucet pattern applied.

The right charts in figures 10.2 and 10.3 show peak memory consumption for the 3-clique and 4-clique topologies with and without Faucet, when the search is limited to cliques originating from a fraction of the full vertex set. The experiments were performed with 8 workers, $N_{\text{batches}} = 4$ and $B = 2 \cdot W_c = 16384$: a representative configuration roughly in the middle of the examined variable space.

It is critical to note that the vanilla dataflow-join topology for 4-cliques cannot complete within 64GB of physical memory (32GB per host) when run on the full dataset: the datapoint for the largest dataset size in the right chart of figure 10.3 is for the enumeration of 4-cliques originating from just 200'000 vertices, or about 4.1% on the full dataset. While we do not have peak memory consumption values for the full uncontrolled execution, other experiments showed that the topology with Faucet peaks only at about 1.4 GB (see figure 9.2).
10.4 Summary

In this chapter we have reported on how Faucet is effective in controlling and stabilising edge buffer sizes. Thanks to this, it provides at least 10-100x savings on peak memory usage in configurations that result in limited runtime overhead, across a wide range of parameter values.

Controlled resource consumption results in greater expressivity: by applying the Faucet pattern the user is able to write well-behaved implementations of programs that have multiple operators with significant input/output rate imbalance.
11.1 Main contributions

In this work we have shown how increased expressivity in the dataflow programming model, such as the support for iterative computations and general operator logic, introduces significant challenges in devising and effective and memory-constrained scheduling strategy. This is important for systems that expose core primitives for the implementation of a wide range of concurrent and distributed algorithms and are not limited to a specific "big data" analytics or machine learning use-case: the timely dataflow programming model implemented by Naiad[38] and Timely Dataflow[36] imposes minimal constraints on programmer expressivity in order to provide flexible, programmable and fine-grained synchronization signals.

In chapter 4 we have identified operator behaviours and classes of topologies that require advanced flow-control and resource management mechanisms and which challenges they present to existing backpressure techniques. Again, increased generality of the programming interface, such as support for operators with significant rate imbalance ($O_{rate} \gg 1$) or cycles in the computation topology, translates in the need for more complex scheduling strategies.

Chapter 5 presented a survey of the existing system-level techniques: basic throttling of the ingress rate is insufficient for a robust implementation of many algorithms and fine-grained, general-purpose, edge-by-edge backpressure, while effective in stabilising memory consumption, can introduce inefficiencies and the risk of deadlock.

While introducing additional challenges and support for greater expressivity, the more powerful primitives exposed by the Timely Dataflow system enable flow-control logic to be implemented as user-level code at the operator level. In chapter 6 we introduced a flow-control pattern that follows this design principle and imposes minimal additional synchronization to the final program. The pattern is modular and programmable, and the user has the burden and flexibility to properly apply it to the topology where necessary. This way, the user has the ability to drive and affect scheduling for a specific computation instead of having to rely on a generic system-wide flow-control and scheduling mechanism.
11. Conclusion

In chapter 7 we described how our partial redesign of Timely Dataflow’s operator interface helped provide better support for user-level scheduling and resulted in a simpler operator contract that more clearly exposes the core primitives of the system.

Chapters 8, 9 and, 10 reported on how the proposed flow-control pattern is effective in multiple different configurations and results in limited overhead, showing how custom, programmable flow-control is feasible and effective in practice.

11.2 Future Work

Some research and engineering questions remain open and would require additional work:

- although not many systems provide the low level abstractions and expressive power of Timely Dataflow, it is possible (albeit uncertain) that the building blocks for user-level, programmable flow control may be made available in other systems so that variations of the Faucet pattern can be implemented: defining the exact set of features required and evaluate the applicability of the technique in other system can be of interest;

- Timely Dataflow currently drops allocated memory employed for tuple shipping: because Faucet limits and stabilizes the edge buffer sizes in the topology, a buffer reuse strategy may yield performance improvements by amortizing the allocation cost;

- the batcher’s support for embedded operator logic can be extended to be more expressive: the backpressure signal can be exposed directly as a bucket of capabilities that represent a permit to send a record or batch at a specific timestamp;

- the pattern assigns specific semantics to the timestamps used to track completion of batches of data and this likely does not yield correct results when the pattern is mixed naively with higher level libraries built on the timely dataflow model, such as differential dataflow[37]: this interaction can be studied to determine how multiple semantics for the timestamps can be mixed in the same program.

Additionally, the generality of the progress tracking mechanism may enable the implementation of other flow-control and scheduling strategies: for example completion times on various workers can be used as a signal to affect and re-balance sharding for datasets with significant skewness.
Appendix A

Original operator interface constraints

The following describes the original Timely Dataflow operator contract for custom operators written using the `unary_notify` and `binary_notify` helper functions.

**Listing A.1:** Example usage for `unary_notify` and `binary_notify` - functions that support creation of operators with custom logic

```cpp
stream.unary_notify(Pipeline, "example", Vec::new(),
                    | input, output, notificator | {
                   // logic
                    });
stream.binary_notify(&stream2, Pipeline, Pipeline, "example",
                      Vec::new(),
                      | input1, input2, output, notificator | {
                     // logic
                      });
```

During the following, the closure passed to `binary_notify` or `unary_notify` is referred to as `logic`.

The requirements for sending data at timestamp \( t \) and for requesting a notification at timestamp \( t \) are the same: both

- obtaining an output session for time to (by calling `output.session(t_o)`) and sending data at that timestamp and;
- requesting a notification at time to (by calling `notificator.notify_at(t_o)`) are only permitted when one (or more) of the following is true.

1. within the same invocation of the logic, a piece of data was read from one of the inputs (via `input.next()`) and its timestamp was \( t_i \) such that \( t_i \leq t_o \);
2. one or more notifications have been requested at timestamp \( t_n \) in a previous invocation of the logic (by calling `notificator.notify_at(t_n)`) and either

   - not all of the requested notifications for timestamp \( t_n \) have been delivered yet (each call to `notification.next()` returns the tuple \((t_n, count)\), where `count` indicates the number of notifications being delivered) or
A. Original operator interface constraints

- it was delivered during the current invocation of the logic.

The number of outstanding notifications at timestamp $t_n$ is increased by 1 on each call to notificator.notify_at($t_n$) and decreased by count when notificator.next() returns the tuple ($t_n$, count).


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PROGRANTABLE SCHEDULING IN A STREAM PROCESSING SYSTEM

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

LATIUDA

First name(s):

ANDREA

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 27.04.2016

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.