Stream Processing Optimizations

Scott Schneider
IBM Thomas J. Watson Research Center
New York, USA

Martin Hirzel
IBM Thomas J. Watson Research Center
New York, USA

Buğra Gedik
Computer Engineering Department
Bilkent University
Ankara, Turkey
Agenda

• 9:00-10:30
  - Overview and background (40 minutes)
  - Optimization catalog (50 minutes)

• 11:00-12:30
  - SPL and InfoSphere Streams background (25 minutes)
  - Fission (40 minutes)
  - Open research questions (25 minutes)
DEBS’13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 1: Overview and Background
Stream Processing

- Streaming sources are plenty
  - Volume, Velocity, Variety
- Online analysis is paramount
  - Quickly process and analyze data, derive insights, and take timely action

Telco analyses streaming network data to reduce hardware costs by 90%

Utility avoids power failures by analysing 10 PB of data in minutes

Hospital analyses streaming vitals to detect illness 24 hours earlier
Catalog of Streaming Optimizations

- Streaming applications: graph of streams and operators
- Performance is an important requirement
- Different communities → different terminology
  - e.g. operator/box/filter; hoisting/push-down
- Different communities → different assumptions
  - e.g. acyclic graphs/arbitrary graphs; shared memory/distributed
- Catalogue of optimizations
  - Uniform terminology
  - Safety & profitability conditions
  - Interactions among optimizations
Fission Optimization

• High throughput processing is a critical requirement
  - Multiple cores and/or host machines
  - System and language level techniques
• Application characteristics limit the speedup brought by optimizations
  - pipeline depth (# of ops), filter selectivity
• Data parallelism is an exception
  - number of available cores (can be scaled)
• Fission
  - Data parallelism optimization in streaming applications
  - How to apply transparently, safely, and adaptively?
Background

• **Operator graph**
  - Operators connected by streams

• **Stream**
  - A series of data items

• **Data item**
  - A set of attributes

• **Operator**
  - Generic data manipulator
  - Has input and output ports
  - Streams connect output ports to input ports
    - FIFO semantics
  - **Source** operator, no input ports
  - **Sink** operator, no output ports

• **Operator firing**
  - Perform processing, produce data items
State in Operators

- **Stateful operators**
  - Maintain state across firings
  - E.g., *deduplicate*: pass data items not seen recently
- **Stateless operators**
  - Do not maintain state across firings
  - E.g., *filter*: pass data items with values larger than a threshold

- **Partitioned stateful operators**
  - Maintain independent state for non-overlapping sub-streams
  - These sub-streams are identified by a *partitioning attribute*
  - E.g.: For each stock symbol in a financial trading stream, compute the volume weighted average price over the last 10 transactions.
    The partitioning attribute: stock symbol.
Selectivity of Operators

- **Selectivity**
  - the number of data items produced per data item consumed
  - e.g., selectivity=0.1 means
    - 1 data item is produced for every 10 consumed
  - used in establishing safety and profitability

- **Dynamic selectivity**
  - selectivity value is
    - *not known at development time*
    - *can change at run-time*
  - e.g., data-dependent filtering, compression, or aggregates on time-based windows
Selectivity Categories

- Selectivity categories (single input/output operators)
  - Exclusively-once (=1): one in; one out [always]
  - At-most-once (≤1): one in; zero or one out [always]
  - Prolific (≥1): one in; one, or more out [sometimes]

- Synchronous data flow (SDF) languages
  - Assume that the selectivity of each operator is fixed and known at compile time
  - Provide good optimization opportunities at the cost of reduced application flexibility
  - Typically used for signal processing applications

- Unlike SDF, we assume dynamic selectivity
  - Support general-purpose streaming

- Selectivity categories are used to fine-tune optimizations
Streaming Programming Models

Synchronous
- Static selectivity
  - e.g., 1 : 3
    ```python
    for i in range(3):
        result = f(i)
        submit(result)
    ```
  - In general, \( m : n \) where \( m \) and \( n \) are statically known
  - Always has static schedule

Asynchronous
- Dynamic selectivity
  - e.g., 1 : [0,1]
    ```python
    if input.value > 5:
        submit(result)
    ```
  - In general, 1 : *
  - In general, schedules cannot be static
Flavors of Parallelism

- There are three main forms of parallelism in streaming applications
  - Pipeline, task, and data parallelism
    
    **Pipeline**
    
    An operator processes a data item at the same time its upstream operator processes the next data item.

    **Task**
    
    Different operators process a data item produced by their common upstream operator, at the same time.

- Pipeline and task parallelism are inherent in the graph
Data Parallelism

- Data parallelism needs to be extracted from the application
  - Morph the graph
    - Split: distribute to replicas
    - Replicate: do data parallel processing
    - Merge: put results back together
- Requires additional mechanisms to preserve application semantics
  - Maintaining the order of tuples
  - Making sure state is partitioned correctly

"different data items from the same stream are processed by the replicas of an operator, at the same time"
Safety and Profitability

• **Safety**: an optimization is *safe* if applying it is guaranteed to maintain the semantics
  - State (stateless & partitioned stateful)
    • Parallel region formation, splitting tuples
  - Selectivity
    • Result ordering, splitting and merging tuples
• **Profitability**: an optimization in profitable if it increases the performance (throughput)
  - Transparency: Does not require developer input
  - Adaptivity: Adapt to resource and workload availability
Adaptive Optimization

- When the workload increases, more resources should be requested
- In the context of data parallelism
  - How many parallel channels to use at a given time
- Maintaining SASO properties is a challenge
  - Stability: do not oscillate wildly
  - Accuracy: eventually find the most profitable operating point
  - Settling time: quickly settle on an operating point
  - Overshoot: steer away from disastrous settings
Publications


DEBS’13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 2: Optimization Catalog
Motivation

• Catalog = survey, but organized as easy reference

• Use cases:
  – User: understand optimized code; hand-implment optimizations
  – System builder: automate optimizations; avoid interference with other features
  – Researcher: literature survey (see paper); open research issues
Stream Optimization Literature

Conflicting terminology
- Operator = filter = box = stage = actor = module
- Data item = tuple = sample
- Join = relational vs. any merge
- Rate = speed vs. selectivity

Unstated assumptions
- Missing safety conditions
- Missing profitability trade-offs
- Any graph vs. forest vs. single-entry, single-exit region
- Shared-memory vs. distributed

DSP (digital signal processing)
Operating systems and networks
DB (databases)
CEP (complex event processing)
Optimization Name

Key idea.

- Preconditions for correctness

Safety

- Most influential published papers

Profitability

- Micro-benchmark
- Runs in SPL
- Relative numbers
- Error bars are standard deviation of 3+ runs

Variations

- How to optimize at runtime

Central trade-off factor

Graph before

Graph after
List of Optimizations

Graph changed
- Operator reordering
- Redundancy elimination
- Operator separation
- Fusion
- Fission

Graph unchanged
- Placement
- Load balancing
- State sharing
- Batching
- Algorithm selection
- Load shedding
Operator Reordering

Change the order in which operators appear in the graph.

• Commutative
• Attributes available

Safety

Profitability

Variations

• Algebraic
• Commutativity analysis
• Synergies, e.g. fusion, fission

Dynamism

• Eddy
Redundancy Elimination

Eliminate operators that are redundant in the graph.

- Dup Split
- A
- B
- C

Safety

- Same algorithm
- Data available

Profitability

Profitability

Variations

- Many-query optimization
- Eliminate no-op
- Eliminate idempotent operator
- Eliminate dead subgraph

Dynamism

- In many-query case: share at submission time
Operator Separation

Separate an operator into multiple constituent operators.

- Ensure $A_1(A_2(s)) = A(s)$

<table>
<thead>
<tr>
<th>Safety</th>
<th>Profitability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Separating Aggregation</td>
</tr>
<tr>
<td></td>
<td>Throughput</td>
</tr>
<tr>
<td></td>
<td>Selectivity of Aggregation</td>
</tr>
</tbody>
</table>

Variations

- Algebraic
- Using special API
- Dependency analysis
- Enables reordering

Dynamism

- N/A
Fusion

*Fuse multiple separate operators into a single operator.*

**Safety**
- Have right resources
- Have enough resources
- No infinite recursion

**Profitability**
- Online recompilation
- Transport operators

**Variations**
- Single vs. multiple threads
- Fusion enables traditional compiler optimizations

**Dynamism**
Fission

Replicate an operator for data-parallel execution.

Safety
- No state or disjoint state
- Merge in order, if needed

Profitability

Variations
- Round-robin (no state)
- Hash by key (disjoint state)
- Duplicate

Dynamism
- Elastic operators (learn width)
- STM (resolve conflicts)
Placement

Place the logical graph onto physical machines and cores.

Safety

- Have right resources
- Have enough resources
- Obey license/security
- If dynamic, need migratability

Variations

- Based on host resources vs. network resources, or both
- Automatic vs. user-specified

Profitability

Based on host resources vs. network resources, or both
- Automatic vs. user-specified

Dynamism

- Submission-time
- Online, via operator migration

Variations

- Submission-time
- Online, via operator migration
Load Balancing

Avoid bottleneck operators by spreading the work evenly.

- Easier for routing than placement
- Balancing work while placing operators
- Balancing work by re-routing data

Safety
- Avoid starvation
- Ensure each worker is equally qualified
- Establish placement safety

Profitability

Variations

Dynamism
- Easier for routing than placement
State Sharing

*Share identical data stored in multiple places in the graph.*

- Common access (usually: fusion)
- No race conditions
- No memory leaks

Safety

Profitability

Variations

Dynamism

- N/A
Batching

*Communicate or compute over multiple data items as a unit.*

**Safety**
- No deadlocks
- Satisfy deadlines

**Profitability**

**Variations**
- Batching enables traditional compiler optimizations

**Dynamism**
- Batching controller
- Train scheduling
Algorithm Selection

Replace an operator by a different operator.

Safety

- $A_\alpha(s) \equiv A_\beta(s)$
- May not need to be safe

Profitability

Variations

- Algebraic
- Auto-tuners
- General vs. specialized

Dynamism

- Compile both versions, then select via control port
Load Shedding

*Degrade gracefully during overload situations.*

- **Safety**
  - By definition, not safe!
  - QoS trade-off

- **Profitability**
  - By definition, not safe!
  - QoS trade-off

- **Variations**
  - Filtering data items (variations: where in graph)
  - Algorithm selection

- **Dynamism**
  - Always dynamic
To Learn More

• DEBS’13 proceedings: “Tutorial: Stream Processing Optimizations”
DEBS’ 13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 3: InfoSphere Streams Background
Streams Programming Model

• Streams applications are data flow graphs that consist of:
  – **Tuples**: structured data item
  – **Operators**: reusable stream analytics
  – **Streams**: series of tuples with a fixed type
  – **Processing Elements**: operator groups in execution
composite Main {
  type
  Entry = int32 uid, rstring server, rstring msg;
  Sum = uint32 uid, int32 total;
  graph
  stream<Entry> Msgs = ParSource() {
    param servers: "logs.*.com";
    partitionBy: server;
  }

  stream<Sum> Sums = Aggregate(Msgs) {
    window Msgs: tumbling, time(5), partitioned;
    param partitionBy: uid;
  }

  stream<Sum> Suspects = Filter(Sums) {
    param filter: total > 100;
  }

  () as Sink = FileSink(Suspects) {
    param file: "suspects.csv";
  }
}
SPL: Immutable by Default

```
stream<Data> Out = Custom(In) {
  logic state: int32 factor_ = 42;
  onTuple In: {
    ++count_;  // immutable by default
    submit({result=In.val*factor_}, Out);
  }
}
```

```
stream<Data> Out = Custom(In) {
  logic state: mutable int32 count_ = 0;
  onTuple In: {
    ++count_;  // explicitly mutable
    submit({count=count_}, Out);
  }
}
```

straight-forward to statically determine this is a stateless operator

straight-forward to statically determine this is a statelful operator
**SPL: Generic Primitive Operators**

*an Aggregate invocation*

```plaintext
stream<Sum> Sums = Aggregate(Msgs) {
  window Msgs: tumbling, time(5),
  partitioned;
  param partitionBy: uid;
}
```

*the Aggregate operator model*

```
{Aggregate
  {parameters {groupBy optional Expression}
   {partitionBy optional Expression}}
  {inputPorts 1 required windowed}
  {outputPorts 1 required}
}
```

---

**SPL compiler**

**Aggregate definition**

**Aggregate instance code**
Source ➔ Compilation ➔ Execution
Source ➔ Compilation ➔ Execution
Source ➔ Compilation ➔ Execution

SPL compiler

Streams Runtime
(Job management, Security, Continuous Resource Management)

Source ➔ Compilation ➔ Execution

x86 host

Source ➔ PE ➔ PE ➔ PE ➔ PE ➔ Sink

SPL compiler

Streams Runtime
(Job management, Security, Continuous Resource Management)
DEBS’ 13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 4: Fission Deep Dive
Fission Overview

```plaintext
composite Main {
  type
  Entry = int32 uid, rstring server,
            rstring msg;
  Sum = uint32 uid, int32 total;

  graph
    stream<Entry> Msgs = ParSource() {
      param servers: "logs.*.com";
      partitionBy: server;
    }

    stream<Sum> Sums = Aggregate(Msgs) {
      window Msgs: tumbling, time(5),
                  partitioned;
      param partitionBy: uid;
    }

    stream<Sum> Suspects = Filter(Sums) {
      param filter: total > 100;
    }

    () as Sink = FileSink(Suspects) {
      param file: "suspects.csv";
    }
}
```
Technical Overview

**Compiler:**
- Apply parallel transformations
- Pick routing mechanism (e.g., hash by key)
- Pick ordering mechanism (e.g., seq. numbers)

**Runtime:**
- Replicate segment into channels
- Add split/merge/shuffle as needed
- Enforce ordering
Transformations

<table>
<thead>
<tr>
<th>Parallelize non-source/sink</th>
<th>Parallelize sources and sinks</th>
<th>Combine parallel regions</th>
<th>Rotate merge and split</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Examples:
- OPRA source
- Database sink

Also known as “shuffle”

Do all of the above as much as possible, wherever it is safe to do so.
Core Challenges

• State
  – **Problem**: No shared memory between channels (partitioned local state)
  – **Solution**: Only parallelize if stateless or partitioned (i.e., separate state into channels by keys)

• Order
  – **Problem**: Race conditions between channels (independent threads of control)
  – **Solution**: Only parallelize if merge can guarantee same tuple order as without parallelization
## Safety Conditions

<table>
<thead>
<tr>
<th>Parallelize non-source/sink</th>
<th>Parallelize sources and sinks</th>
<th>Combine parallel regions</th>
<th>Rotate merge and split</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- **stateless or partitioned state**
- **simple chain**

- **stateless or partitioned state**

- **stateless**
  - compatible keys
  - forwarding

- **incompatible keys**
  - selectivity ≤ 1
Select Parallel Segments

• Can't parallelize
  – Operators with >1 fan-in or fan-out
  – Punctuation dependency later on

• Can't add operator to parallel segment if
  – Another operator in segment has co-location constraint
  – Keys don't match
Constraints & Fusion

Infer partition colocation → Select parallel segments → Fusion

Compile-time

Expand parallel segments → Check placement constraints → Place partitions on hosts

Submission-time

ADL
Compiler to Runtime

Compiler

Graph + unexpanded parallel regions

Fully expanded graph

Runtime graph fragment
Runtime graph fragment
Runtime graph fragment

PE
PE
PE

compile-time
submission-time
run-time
## Runtime

<table>
<thead>
<tr>
<th>state</th>
<th>selectivity</th>
<th>gaps</th>
<th>dups</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>round-robin</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>1 : 1</td>
</tr>
<tr>
<td>seqno</td>
<td>partitioned</td>
<td>✗</td>
<td>✗</td>
<td>1 : 1</td>
</tr>
<tr>
<td>strict seqno &amp; pulse</td>
<td>partitioned</td>
<td>✓</td>
<td>✗</td>
<td>1 : [0,1]</td>
</tr>
<tr>
<td>relaxed seqno &amp; pulse</td>
<td>partitioned</td>
<td>✓</td>
<td>✓</td>
<td>1 : [0,∞]</td>
</tr>
</tbody>
</table>

Operators in parallel segments:
- Forward seqno & pulse

Split:
- Insert seqno & pulse
- Routing

Merge:
- Apply ordering policy
- Remove seqno (if there) and drop pulse (if there)
Merger Ordering

Round-Robo

Sequence Numbers

Strict Sequence Number & Pulses

Relaxed Sequence Number & Pulses
Application Kernel Performance

Speedup vs. 1 channel

Number of parallel channels

Network monitoring
Twitter NLP
PageRank
Twitter CEP
Finance

(a) Network monitoring
(b) PageRank
(c) Twitter NLP
(d) Twitter CEP
(e) Finance
Elasticity: The Problem

• What is $N$? We want to:
  – find it dynamically, at runtime
  – automatically, with no user intervention
  – in the presence of stateless and partitioned stateful operators
  – maximize throughput
Elasticity: Solution Sketch

local control, adaptation

global storage, synchronization
DEBS’13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 6: Open Research Questions
Programming Model Challenges

High-level
Easy to use
Optimizable

CEP patterns
StreamDatalog
StreamSQL
StreamIt (MIT)
Graph GUI
SPL
Java API
Annotated C
C/Fortran

Low-level
General
Predictable

Other challenges
• Foreign code
• Familiarity
Interaction of SPL and C++

At compile time
Application source code (SPL)

At run time
Application model (XML)

SPL Compiler

Operator model (XML)

Operator instance model (XML)

Operator code generator

Operator instance (C++)

Streaming platform

Stream of input data items

Stream of output data items
Optimization Combination

- Operator reordering
- Redundancy elimination
- Placement
- State sharing
- Fission
- Operator separation
- Fusion
- Load balancing
- Algorithm selection
- Load shedding

Challenges
- If separate: order
- If combined: profitability model
Interaction with Traditional Compiler Analysis

Traditional compiler analyses

- Operator reordering
- Operator separation
- Fission
- Redundancy elimination
- Placement
- Fusion
- Load shedding
- Load balancing
- Algorithm selection
- Batching
- State sharing
- Challenges:
  - State
  - Ordering
  - Selectivity
  - Key forwarding
Interaction with Traditional Compiler Optimizations

- Traditional compiler analyses
  - Operator reordering
  - Operator separation
  - Fission
  - Algorithm selection
  - Load shedding
- Redundancy elimination
- Placement
- Fusion
- Load balancing
- State sharing
- Batching
- Traditional compiler optimizations

Challenges:
- Inlining
- Loop unrolling
- Deforestation
- Scalarization
# Dynamic Optimization

<table>
<thead>
<tr>
<th>Compile time</th>
<th>Submission time</th>
<th>Runtime discrete</th>
<th>Runtime continuous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator separation</td>
<td>Redundancy elimination</td>
<td>Load balancing</td>
<td>Operator reordering</td>
</tr>
<tr>
<td>Fusion</td>
<td>Fission</td>
<td></td>
<td>Batching</td>
</tr>
<tr>
<td>State sharing</td>
<td>Placement</td>
<td></td>
<td>Load shedding</td>
</tr>
<tr>
<td>Algorithm selection</td>
<td></td>
<td>Other challenges:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Settling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Accuracy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Stability</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Overshoot</td>
<td></td>
</tr>
</tbody>
</table>

Other challenges:
- Settling
- Accuracy
- Stability
- Overshoot
Dynamic Operator Reordering

Approach: Emulate graph change via data-item routing.
Example: Eddies [Avnur, Hellerstein SIGMOD’00]
Benchmarks

**Wish List**
- Representative
  - ... of real code
  - ... of real inputs
- Fast enough to conduct many experiments
- Fully automated / scripted
- Self-validating
- Open-source with industry-friendly license

**Literature**
- LinearRoad
  [Arasu et al. VLDB’04]
- BiCEP
  [Mendes, Bizarro, Marques TPC TC’09]
- StreamIt
  [Thies, Amarasinghe PACT’10]
Generality of Optimizations

Challenges
- Expand "Supported"
- In the right direction
Generality of Fission

<table>
<thead>
<tr>
<th>State</th>
<th>Ordering</th>
<th>Topology</th>
<th>User code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateless</td>
<td>Static selectivity</td>
<td>Single operator</td>
<td>Built-in operators</td>
</tr>
<tr>
<td>Partitioned</td>
<td>Dynamic selectivity</td>
<td>Simple pipeline</td>
<td>Streaming language</td>
</tr>
<tr>
<td>Arbitrary</td>
<td></td>
<td>Arbitrary subgraph</td>
<td>Foreign language</td>
</tr>
<tr>
<td>Stateful</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Challenges
- Expand “Supported”
- In the right direction