Scaling Ordered Stream Processing on Shared-Memory Multicores

Guna Prasaad

G. Ramalingam, Kaushik Rajan

26 Aug, 2019
Realtime Stream Processing

• In today’s world, data is of utmost value as it “arrives”

• Ability to process data in realtime is key to enabling several applications

• Stream processing has a very long history both inside and outside the database community

• New use-cases: surveillance, fraud detection, ad-serving, shopping cart analysis, online multiplayer games, live video streaming and distribution

• Processing large volumes of high-speed data in realtime is a challenge
Stream Processing Engines

- Allow users to define a computation pipeline that operates on a continuous stream of incoming data

- Architectures vary from a single core to shared-memory multicores to distributed shared-nothing

- Predominantly adopt the micro-batch architecture
Shared-Memory Parallelism

- Streaming pipelines generally have a bounded memory footprint
- Tremendous growth in memory sizes and core counts

Single shared-memory machine is “often” sufficient

Building block for distributed stream processing engines

- Treat each core in a multicore machine as an individual node
- Fail to exploit low-overhead shared-memory parallelism
Ordered Stream Processing

Semantically equivalent to executing the stream computation on input stream serially one after another.
Ordered Stream Processing

Semantically equivalent to executing the stream computation on input stream serially one after another

• Streams of events/tuples already have a notion of *temporal ordering*

• In many scenarios application logic depends on the event order

• For instance, timeout based sessions in a clickstream
Ordered Stream Processing

Semantically equivalent to executing the stream computation on input stream serially one after another

- Streams of events/tuples already have a notion of temporal ordering
- In many scenarios application logic depends on the event order
- For instance, timeout based sessions in a clickstream
- Easy deployment with fault-tolerance in the distributed setting
- Active replication requires deterministic processing guarantee
Overview
Parallelism in a Stream Computation

map → filter → average-by-key → filter → average

filter → sum
Parallelism in a Stream Computation

Data Parallelism
Parallelism in a Stream Computation

Data Parallelism
Parallelism in a Stream Computation

Data Parallelism

Pipeline Parallelism

map → filter → average-by-key

filter → average

filter → sum
Parallelism in a Stream Computation

- map
- filter
- average-by-key
- filter
- average
- filter
- sum

Data Parallelism

Pipeline Parallelism

Task Parallelism
Order & Data Parallelism

\[ \ldots, i_3, i_2, i_1 \rightarrow \blacklozenge \rightarrow \ldots o_3, o_2, o_1 \]
Order & Data Parallelism

\[ \ldots, i_3, i_2, i_1 \rightarrow \text{\textbullet} \rightarrow \ldots o_3, o_2, o_1 \]

Value of Output

\[ i_n \rightarrow \text{\textbullet} \rightarrow o_n \]
Order & Data Parallelism

\[ \ldots, i_3, i_2, i_1 \rightarrow \blacklozenge \rightarrow \ldots o_3, o_2, o_1 \]

Value of Output

\[ i_n \rightarrow \blacklozenge \rightarrow o_n \]

Order of Outputs

\[ \ldots, o_3, o_2, o_1 \]
Ordering Semantics

If $i < i'$, when can $i$ and $i'$ be processed out-of-order?
Ordering Semantics

If $i < i'$, when can $i$ and $i'$ be processed out-of-order?

$$(o_n, S_n) = \text{operate}(S_{n-1}, i_n)$$
Ordering Semantics

If $i < i'$, when can $i$ and $i'$ be processed out-of-order?

$$(o_n, S_n) = \text{operate}(S_{n-1}, i_n)$$

Stateless
Ordering Semantics

If $i < i'$, when can $i$ and $i'$ be processed out-of-order?

$$(o_n, S_n) = \text{operate}(S_{n-1}, i_n)$$

Stateless

Always
Ordering Semantics

If \( i < i' \), when can \( i \) and \( i' \) be processed out-of-order?

\[
(o_n, S_n) = \text{operate}(S_{n-1}, i_n)
\]

*Stateless*

*Stateful*

Always
Ordering Semantics

If $i < i'$, when can $i$ and $i'$ be processed out-of-order?

$$(o_n, S_n) = \text{operate}(S_{n-1}, i_n)$$

Stateless: Always
Stateful: Never
Ordering Semantics

If $i < i'$, when can $i$ and $i'$ be processed out-of-order?

$$(o_n, S_n) = \text{operate}(S_{n-1}, i_n)$$

- Stateless: Always
- Stateful: Never
- Partitionable Stateful
Ordering Semantics

If $i < i'$, when can $i$ and $i'$ be processed out-of-order?

$$(o_n, S_n) = \text{operate}(S_{n-1}, i_n)$$

- **Stateless**: Always
- **Stateful**: Never
- **Partitionable Stateful**: $\mathbb{P}(i) \neq \mathbb{P}(i')$
Stream Dataflow Graph

1 → 2 → 3 → 4 → 5 → 6
Async Executable

Decouple operators by allowing inputs to be processed “asynchronously”
Decouple operators by allowing inputs to be processed “asynchronously”
Decouple operators by allowing inputs to be processed “asynchronously”
Each operator executable must individually provide the ordering guarantee when multiple workers are allotted.
Key Requirement

Each operator executable must individually provide the ordering guarantee when multiple workers are allotted.

Clear separation of concerns between correctness and optimisation.
Outline

• Reordering Outputs
• Partitionable Stateful Operators
• Scheduling Runtime
• Evaluation
• Conclusion
Reordering Outputs
Reordering Outputs

- Each input is assigned a “sequence number” based on their arrival
- Concurrent workers are operating on inputs to produce outputs
- We want to reorder and send them downstream in the input arrival order
- Output $o_{i+1}$, even if produced earlier, can only be sent downstream after all of $o_1, o_2, \ldots, o_i$ have been sent
void send(o_t) {
    lock();
    if (t ≠ next) {
        add o_t to buffer
    } else {
        send_downstream(o_t);
        next++;
        while(buffer has o_{next}) {
            send_downstream(o_{next});
            next++;
        }
    }
    unlock();
}
Lock-Based Reordering

```
void send(o_t) {
  lock();
  if (t \neq next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_next) {
      send_downstream(o_next);
      next++;
    }
  }
  unlock();
}
```
Lock-Based Reordering

```c
void send(o_t) {
    lock();
    if (t != next) {
        add o_t to buffer
    } else {
        send_downstream(o_t);
        next++;
        while(buffer has o_next) {
            send_downstream(o_next);
            next++;
        }
    }
    unlock();
}
```
Lock-Based Reordering

```c
void send(o_t) {
    lock();
    if (t ≠ next) {
        add o_t to buffer
    } else {
        send_downstream(o_t);
        next++;
        while (buffer has o_{next}) {
            send_downstream(o_{next});
            next++;
        }
    }
    unlock();
}
```
Lock-Based Reordering

```c
void send(o_t) {
    lock();
    if (t != next) {
        add o_t to buffer
    } else {
        send_downstream(o_t);
        next++;
        while(buffer has o_next) {
            send_downstream(o_next);
            next++;
        }
    }
    unlock();
}
```
void send(o_t) {
    lock();
    if (t ≠ next) {
        add o_t to buffer
    } else {
        send_downstream(o_t);
        next++;
        while(buffer has o_{next}) {
            send_downstream(o_{next});
            next++;
        }
    }
    unlock();
}
void send(o_t) {
  lock();
  if (t \neq next) {
    add o_t to buffer
  } else {
    send_downstream(o_t);
    next++;
    while(buffer has o_{next}) {
      send_downstream(o_{next});
      next++;
    }
  }
  unlock();
}
Lock-Based Reordering

- Each output already has a designated location on the buffer
- Decouple adding to buffer from sending downstream
- If a worker is already sending outputs downstream, delegate the work to it and return to do more useful work
Our Solution: Non-Blocking Reordering

```c
// data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

// invoked by workers
bool send(o_t) {
    bool success = try_add(o_t);
    while (not flag.test_and_set()) {
        send_ready_outputs_downstream();
        flag.clear();
        if (!more_ready_outputs()) {
            break;
        }
    }
    return success;
}
```
Our Solution: Non-Blocking Reordering

```c
//data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

//invoked by workers
bool send(o_t) {
    bool success = try_add(o_t);
    while (not flag.test_and_set()) {
        send_ready_outputs_downstream();
        flag.clear();
        if (!more_ready_outputs()) {
            break;
        }
    }
    return success;
}
```
Our Solution: Non-Blocking Reordering

```c
// data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

// invoked by workers
bool send(o_t) {
    bool success = try_add(o_t);
    while (!flag.test_and_set()) {
        send_ready_outputs_downstream();
        flag.clear();
        if (!more_ready_outputs()) {
            break;
        }
    }
    return success;
}
```
Our Solution: Non-Blocking Reordering

```c
1 // data fields
2 atomic_long next;
3 atomic<output*> buffer[s];
4 atomic_flag flag;
5
6 // invoked by workers
7 bool send(o_t) {
8     bool success = try_add(o_t);
9     while (not flag.test_and_set()) {
10         send_ready_outputs_downstream();
11         flag.clear();
12         if (!more_ready_outputs()) {
13             break;
14         }
15     }
16     return success;
17 }
```
Our Solution: Non-Blocking Reordering

```c
1 // data fields
2 atomic_long next;
3 atomic<output*> buffer[s];
4 atomic_flag flag;
5
6 // invoked by workers
7 bool send(o_t) {
8     bool success = try_add(o_t);
9     while (not flag.test_and_set()) {
10         send_ready_outputs_downstream();
11         flag.clear();
12         if (!more_ready_outputs()) {
13             break;
14         }
15     }  
16     return success;
17 }
```
Our Solution: Non-Blocking Reordering

```c
// data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

// invoked by workers
bool send(o_t) {
  bool success = try_add(o_t);
  while (not flag.test_and_set()) {
    send_ready_outputs_downstream();
    flag.clear();
    if (!more_ready_outputs()) {
      break;
    }
  }
  return success;
}```
Our Solution: Non-Blocking Reordering

```c
// data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

// invoked by workers
bool send(o_t) {
    bool success = try_add(o_t);
    while (not flag.test_and_set()) {
        send_ready_outputs_downstream();
        flag.clear();
        if (!more_ready_outputs()) {
            break;
        }
    }
    return success;
}
```
Our Solution: Non-Blocking Reordering

1 //data fields
2 atomic_long next;
3 atomic<output*> buffer[s];
4 atomic_flag flag;
5
6 //invoked by workers
7 bool send(o_t) {
8     bool success = try_add(o_t);
9     while (not flag.test_and_set()) {
10         send_ready_outputs_downstream();
11         flag.clear();
12         if (!more_ready_outputs()) {
13             break;
14         }
15     }
16     return success;
17 }
Our Solution: Non-Blocking Reordering

```c
1 // data fields
2 atomic_long next;
3 atomic<output*> buffer[s];
4 atomic_flag flag;
5
6 // invoked by workers
7 bool send(o_t) {
8     bool success = try_add(o_t);
9     while (not flag.test_and_set()) {
10         send_ready_outputs_downstream();
11         flag.clear();
12         if (!more_ready_outputs()) {
13             break;
14         }
15     }
16     return success;
17 }
```
Our Solution: Non-Blocking Reordering

```c
// data fields
atomic_long next;
atomic<output*> buffer[s];
atomic_flag flag;

// invoked by workers
bool send(o_t) {
    bool success = try_add(o_t);
    while (not flag.test_and_set()) {
        send_ready_outputs_downstream();
        flag.clear();
        if (!more_ready_outputs()) {
            break;
        }
    }
    return success;
}
```
Our Solution: Non-Blocking Reordering

1 //data fields
2 atomic_long next;
3 atomic<output*> buffer[s];
4 atomic_flag flag;
5
6 //invoked by workers
7 bool send(o_t) {
8     bool success = try_add(o_t);
9     while (not flag.test_and_set()) {
10         send_ready_outputs_downstream();
11         flag.clear();
12         if (!more_ready_outputs()) {
13             break;
14         }
15     }
16     return success;
17 }
Our Solution: Non-Blocking Reordering

```c
1 //data fields
2 atomic_long next;
3 atomic<output*> buffer[s];
4 atomic_flag flag;
5
6 //invoked by workers
7 bool send(o_i) {
8    bool success = try_add(o_i);
9    while (!flag.test_and_set()) {
10       send_ready_outputs_downstream();
11       flag.clear();
12       if (!more_ready_outputs()) {
13          break;
14       }
15    }
16    return success;
17 }
```
Our Solution: Non-Blocking Reordering

1 //data fields
2 atomic_long next;
3 atomic<output*> buffer[s];
4 atomic_flag flag;
5
6 //invoked by workers
7 bool send(o_t) {
8     bool success = try_add(o_t);
9     while (not flag.test_and_set()) {
10        send_ready_outputs_downstream();
11        flag.clear();
12        if (!more_ready_outputs()) {
13            break;
14        }
15    }
16    return success;
17 }
Partitionable Stateful Operators
Partitionable Stateful Operators

We have been working on group-by-aggregates for several decades, what’s new?
Partitionable Stateful Operators

We have been working on group-by-aggregates for several decades, what’s new?

Latency-Critical  Ordering Requirement
Partitionable Stateful Operators

We have been working on group-by-aggregates for several decades, what’s new?

If $i < i'$, they can be processed out-of-order (or concurrently) only when $\mathcal{P}(i) \neq \mathcal{P}(i')$
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue
- Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue

Algorithm
1. Dequeue input
2. Lock partition
3. Operate on input
4. Unlock partition
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue

• Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue

• Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition

Dequeue $i_1$
Lock $p_2$
Operate on $i_1$
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue

• Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue

• Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition
Shared Queue

- Each tuple has an associated partition
- All inputs are added into a single shared linearizable concurrent queue

Algorithm
1. Dequeue input
2. Lock partition
3. Operate on input
4. Unlock partition
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue

• Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition
Shared Queue

• Each tuple has an associated partition

• All inputs are added into a single shared linearizable concurrent queue

• Algorithm
  1. Dequeue input
  2. Lock partition
  3. Operate on input
  4. Unlock partition
Partitioned Queues

- Consider each partition as a stateful operator with its own queue

- At most one worker can process a partition

- Most commonly used strategy in all stream processing engines

- Unnecessary blocking of outputs in the reordering buffer
Partitioned Queues

- Consider each partition as a stateful operator with its own queue.

- At most one worker can process a partition.

- Most commonly used strategy in all stream processing engines.

- Unnecessary blocking of outputs in the reordering buffer.

\[
\begin{array}{cccc}
 i_7 & : & p_1 & \\
 i_3 & : & p_1 & \\
 i_1 & : & p_1 & \\
 i_5 & : & p_2 & \\
 i_2 & : & p_2 & \\
 i_6 & : & p_3 & \\
 i_4 & : & p_3 & \\
\end{array}
\]
Partitioned Queues

- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer
Partitioned Queues

- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer
Partitioned Queues

- Consider each partition as a stateful operator with its own queue
- At most one worker can process a partition
- Most commonly used strategy in all stream processing engines
- Unnecessary blocking of outputs in the reordering buffer
Solutions

Shared Queue

• Almost ordered processing
• Partition guarantee violation

Partitioned Queues

• Partition guarantee
• Output blocking due to out-of-order processing
Our Solution: Hybrid Strategy

- Master Queue: Contains only partition ids in the order of arrival
- Partition Queues:
  - One for each partition
  - Each queue contains inputs belonging to a single partition
- Non-blocking strategy in the ordered setting!
//invoked by producers
void addInput(tuple) {
    p = getPartition(tuple);
    partitionQueues[p].enqueue(msg);
    masterQueue.enqueue(p);
}
Our Solution: Hybrid Queue

```java
// invoked by producers
void addInput(tuple) {
    p = getPartition(tuple);
    partitionQueues[p].enqueue(msg);
    masterQueue.enqueue(p);
}
```

Master Queue

Partition Queues
Our Solution: Hybrid Queue

```
// invoked by producers
void addInput(tuple) {
  p = getPartition(tuple);
  partitionQueues[p].enqueue(msg);
  masterQueue.enqueue(p);
}
```
Our Solution: Hybrid Queue

//invoked by producers
void addInput(tuple) {
  p = getPartition(tuple);
  partitionQueues[p].enqueue(msg);
  masterQueue.enqueue(p);
}
Our Solution: Hybrid Queue

```java
//invoked by producers
void addInput(tuple) {
    p = getPartition(tuple);
    partitionQueues[p].enqueue(msg);
    masterQueue.enqueue(p);
}
```
//invoked by producers
void addInput(tuple) {
  p = getPartition(tuple);
  partitionQueues[p].enqueue(msg);
  masterQueue.enqueue(p);
}
Our Solution: Hybrid Queue

```java
// invoked by producers
void addInput(tupple) {
    p = getPartition(tupple);
    partitionQueues[p].enqueue(msg);
    masterQueue.enqueue(p);
}
```

![Diagram showing hybrid queue structure with master queue and partition queues]
Our Solution: Hybrid Queue

```java
//invoked by producers
void addInput(t tuple) {
    p = getPartition(t tuple);
    partitionQueues[p].enqueue(msg);
    masterQueue.enqueue(p);
}
```
void consumeInputs() {
    while (masterQueue.tryDequeue(p)) {
        if (count[p].fetch_add(1) == 0) {
            do {
                partitionQueues[p].tryDequeue(tuple);
                operate(tuple);
            } while (count[p].fetch_sub(1) > 1);
        }
    }
}
Our Solution: Hybrid Queue

```c
//invoked by workers
void consumeInputs() {
    while(masterQueue.tryDequeue(p)) {
        if(count[p].fetch_add(1) == 0) {
            do {
                partitionQueues[p].tryDequeue(tuple);
                operate(tuple);
            } while(count[p].fetch_sub(1) > 1);
        } }
```
Our Solution: Hybrid Queue

```cpp
//invoked by workers
void consumeInputs() {
    while(masterQueue.tryDequeue(p)) {
        if(count[p].fetch_add(1) == 0) {
            do {
                partitionQueues[p].tryDequeue(tuple);
                operate(tuple);
            } while(count[p].fetch_sub(1) > 1);
        }
    }
}
```

Counts

Dequeue $p_2$

$counts[p_2]: 0 \rightarrow 1$
Our Solution: Hybrid Queue

//invoked by workers
void consumeInputs() {
    while (masterQueue.tryDequeue(p)) {
        if (count[p].fetch_add(1) == 0) {
            do {
                partitionQueues[p].tryDequeue(tup); // Dequeue $p_2$
                operate(tup);
            } while (count[p].fetch_sub(1) > 1);
        }
    }
}

Counts:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dequeue $p_2$

counts[$p_2$]: $0 \rightarrow 1$

Dequeue $i_1$
Our Solution: Hybrid Queue

//invoked by workers
void consumeInputs() {
    while(masterQueue.tryDequeue(p)) {
        if(count[p].fetch_add(1) == 0) {
            do {
                partitionQueues[p].tryDequeue(tuple);
                operate(tuple);
            } while(count[p].fetch_sub(1) > 1);
        }
    }
}

Counts

Dequeue $p_2$

counts[$p_2$] : 0 $\rightarrow$ 1

Dequeue $i_1$

Operate on $i_1$
Our Solution: Hybrid Queue

```c
// invoked by workers
void consumeInputs() {
    while(masterQueue.tryDequeue(p)) {
        if(count[p].fetch_add(1) == 0) {
            do {
                partitionQueues[p].tryDequeue(ttuple);
                operate(ttuple);
            } while(count[p].fetch_sub(1) > 1);
        }
    }
}
```

Counts

- Dequeue $p_2$
- $\text{counts}[p_2] : 0 \rightarrow 1$
- Dequeue $i_1$
- Operate on $i_1$
- $\text{counts}[p_2] : 1 \rightarrow 0$
Our Solution: Hybrid Queue
Our Solution: Hybrid Queue

Deque $p_1$

Deque $p_1$

Counts

Tiles:
- $p_1$
- $p_3$
- $p_2$
- $p_3$

Tiles:
- $i_7$
- $i_3$
- $i_5$
- $i_6$
- $i_4$
Our Solution: Hybrid Queue

Dequeue $p_1$

counts[$p_1$]: 0 $\rightarrow$ 1

Dequeue $p_1$

counts[$p_1$]: 1 $\rightarrow$ 2

Counts

0 2 0
Our Solution: Hybrid Queue

Dequeue $p_1$

$\text{counts}[p_1] : 0 \rightarrow 1$

Dequeue $p_1$

$\text{counts}[p_1] : 1 \rightarrow 2$

Counts

0 2 0

Dequeue $i_2$

Dequeue $i_2$
Our Solution: Hybrid Queue

- Dequeue $p_1$
  - counts[$p_1$]: $0 \rightarrow 1$
- Dequeue $i_2$
- Dequeue $p_1$
  - counts[$p_1$]: $1 \rightarrow 2$
- Dequeue $p_3$

Counts:

0 2 0

Elements:

- $p_1$
- $i_7$
- $i_6$
- $i_5$
- $i_4$
- $i_3$
- $p_3$
- $p_2$
Our Solution: Hybrid Queue

- Dequeue $p_1$
  - `counts[p_1]`: $0 \rightarrow 1$
- Dequeue $i_2$
- Operate on $i_2$

- Dequeue $p_1$
  - `counts[p_1]`: $1 \rightarrow 2$

- Dequeue $p_3$
  - `counts[p_3]`: $0 \rightarrow 1$

Counts: $0 \rightarrow 2 \rightarrow 1$
Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
<th>Dequeue $i_2$</th>
<th>Dequeue $p_3$</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{counts}[p_1] : 0 \rightarrow 1$</td>
<td>$\text{counts}[p_1] : 1 \rightarrow 2$</td>
<td>$\text{Dequeue} \ i_2$</td>
<td>$\text{Dequeue} \ p_3$</td>
<td>$0 \ 1 \ 1$</td>
</tr>
<tr>
<td>$\text{counts}[p_1] : 2 \rightarrow 1$</td>
<td>$\text{counts}[p_3] : 0 \rightarrow 1$</td>
<td>$\text{Operate on} \ i_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>counts[$p_1$]: 0 → 1</td>
<td>counts[$p_1$]: 1 → 2</td>
</tr>
<tr>
<td>Dequeue $i_2$</td>
<td>Dequeue $p_3$</td>
</tr>
<tr>
<td>counts[$p_1$]: 2 → 1</td>
<td>counts[$p_3$]: 0 → 1</td>
</tr>
<tr>
<td>Operate on $i_2$</td>
<td>Dequeue $i_4$</td>
</tr>
</tbody>
</table>

Counts

$p_1$ $p_2$ $p_3$ $i_7$ $i_3$ $i_5$ $i_6$
Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>counts[$p_1$]: 0 → 1</td>
<td>counts[$p_1$]: 1 → 2</td>
<td>counts[$p_1$]: 1 → 2</td>
</tr>
<tr>
<td>Dequeue $i_2$</td>
<td>Dequeue $p_3$</td>
<td>Dequeue $i_4$</td>
</tr>
<tr>
<td>Operate on $i_2$</td>
<td>counts[$p_3$]: 0 → 1</td>
<td></td>
</tr>
<tr>
<td>counts[$p_1$]: 2 → 1</td>
<td>Dequeue $i_4$</td>
<td></td>
</tr>
<tr>
<td>Dequeue $i_3$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Counts

$\begin{array}{c}
0 \\
1 \\
1 \\
\end{array}$
## Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{counts}[p_1] : 0 \rightarrow 1$</td>
<td>$\text{counts}[p_1] : 1 \rightarrow 2$</td>
</tr>
<tr>
<td>Dequeue $i_2$</td>
<td>Dequeue $p_3$</td>
</tr>
<tr>
<td>$\text{counts}[p_1] : 2 \rightarrow 1$</td>
<td>$\text{counts}[p_3] : 0 \rightarrow 1$</td>
</tr>
<tr>
<td>Dequeue $i_3$</td>
<td>Dequeue $i_4$</td>
</tr>
<tr>
<td>Operate on $i_2$</td>
<td>Operate on $i_4$</td>
</tr>
</tbody>
</table>

**Counts**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

- Dequeue $p_1$
- Dequeue $p_3$
- Dequeue $i_4$
Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{counts}[p_1]: 0 \rightarrow 1$</td>
<td>$\text{counts}[p_1]: 1 \rightarrow 2$</td>
</tr>
<tr>
<td>Dequeue $i_2$</td>
<td>Dequeue $p_3$</td>
</tr>
<tr>
<td>$\text{counts}[p_1]: 2 \rightarrow 1$</td>
<td>$\text{counts}[p_3]: 0 \rightarrow 1$</td>
</tr>
<tr>
<td>Operate on $i_2$</td>
<td>Dequeue $i_4$</td>
</tr>
<tr>
<td>Operate on $i_3$</td>
<td>Operate on $i_4$</td>
</tr>
<tr>
<td>$\text{counts}[p_3]: 1 \rightarrow 0$</td>
<td>$\text{counts}[p_3]: 1 \rightarrow 0$</td>
</tr>
</tbody>
</table>

Counts:

- $p_1$
- $p_3$
- $p_2$
- $i_7$
- $i_5$
- $i_6$
## Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dequeue $p_1$</td>
<td></td>
<td>0-2-0</td>
</tr>
<tr>
<td>Counts[$p_1$] : 0 → 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dequeue $i_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operate on $i_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counts[$p_1$] : 2 → 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dequeue $i_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operate on $i_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counts[$p_1$] : 1 → 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dequeue $p_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counts[$p_1$] : 1 → 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dequeue $p_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counts[$p_3$] : 0 → 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dequeue $i_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operate on $i_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counts[$p_3$] : 1 → 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $i_7$: Blue
- $i_5$ and $i_6$: Green
- $p_1$, $p_2$, $p_3$: Yellow
# Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>counts[$p_1$]: 0 $\rightarrow$ 1</td>
<td>counts[$p_1$]: 1 $\rightarrow$ 2</td>
</tr>
<tr>
<td>Dequeue $i_2$</td>
<td>Dequeue $p_3$</td>
</tr>
<tr>
<td>Operate on $i_2$</td>
<td>counts[$p_3$]: 0 $\rightarrow$ 1</td>
</tr>
<tr>
<td>counts[$p_1$]: 2 $\rightarrow$ 1</td>
<td>Dequeue $i_4$</td>
</tr>
<tr>
<td>Dequeue $i_3$</td>
<td>Operate on $i_4$</td>
</tr>
<tr>
<td>Operate on $i_3$</td>
<td>counts[$p_3$]: 1 $\rightarrow$ 0</td>
</tr>
<tr>
<td>counts[$p_1$]: 1 $\rightarrow$ 0</td>
<td>Dequeue $p_2$</td>
</tr>
</tbody>
</table>

Counts

0 0 0
# Our Solution: Hybrid Queue

<table>
<thead>
<tr>
<th>Dequeue $p_1$</th>
<th>Dequeue $p_1$</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{counts}[p_1]: 0 \rightarrow 1$</td>
<td>$\text{counts}[p_1]: 1 \rightarrow 2$</td>
<td>0 0 0</td>
</tr>
<tr>
<td>Dequeue $i_2$</td>
<td>Dequeue $p_3$</td>
<td></td>
</tr>
<tr>
<td>$\text{counts}[p_1]: 2 \rightarrow 1$</td>
<td>$\text{counts}[p_3]: 0 \rightarrow 1$</td>
<td></td>
</tr>
<tr>
<td>Operate on $i_2$</td>
<td>Dequeue $i_4$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operate on $i_4$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{counts}[p_3]: 1 \rightarrow 0$</td>
<td></td>
</tr>
<tr>
<td>Dequeue $i_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dequeue $p_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operate on $i_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dequeue $p_3$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{counts}[p_1]$ indicates the number of times $p_1$ has been dequeued.
Operator Implementations

MPMC Queue
- Stateless
  - Reordering Buffer

SPSC Queue
- Stateful

MPMC Master Queue
- Partitionable Stateful
  - Reordering Buffer
  - SPSC Partition Queues
Scheduling Runtime
Dynamic Scheduling

Monitor the state of the pipeline and operator characteristics to answer

Which operator should a worker next work on?
Parameters of Interest

\[ I_i \]  Input queue size

\[ O_i \]  Output queue size

\[ S_i \]  Average selectivity i.e. Number of outputs per input

\[ C_i \]  Operator cost i.e. Time taken to process each input

\[ W_i \]  Number of workers allotted currently

\[ M_i \]  Maximum allowed number of workers
4 Heuristics

- Queue-Size Throttling (QST)
- Last-In-Pipeline (LIP)
- Estimated Time (ET)
- Current Throughput (CT)
Queue-Size Throttling (QST)

- Apply pressure from ingress towards egress
- Focus on one operator at a time
- Each operator has an upper bound on output queue size
- Normalize for selectivity
- Pick earliest operator in the pipeline with output queue size less than its threshold

$$cS_i = \prod_{k=1}^{i} s_i$$
Queue-Size Throttling (QST)

- Apply pressure from ingress towards egress
- Focus on one operator at a time
- Each operator has an upper bound on output queue size
- Normalize for selectivity
- Pick earliest operator in the pipeline with output queue size less than its threshold

\[
cs_i = \prod_{k=1}^{i} s_i
\]

\[
T_i = \frac{C \ast cs_i}{\sum_{i=1}^{n} cs_i}
\]
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)

1 2 3 4 5 x 2
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Queue-Size Throttling (QST)
Last In Pipeline (LIP)

- Complementary to QST
- Provide suction to pull tuples from ingress towards egress
- Prioritizes operators later in the pipeline
- Operator is “schedulable” if it has
  - Less than maximum allowed workers assigned
  - Minimum input queue size
Last In Pipeline (LIP)
Last In Pipeline (LIP)
Last In Pipeline (LIP)
Last In Pipeline (LIP)
Last In Pipeline (LIP)
Last In Pipeline (LIP)
Last In Pipeline (LIP)
Last In Pipeline (LIP)
Estimated Time (ET)

• Priority-based: Compute a priority score for each operator and assign worker to the one with highest score

• Priority score is estimated time to process the current input queue to completion if an additional worker is assigned

• Intuition: Operator that will take more time to complete needs additional worker time

\[ p_i = \frac{I_i \times c_i}{w_i + 1} \]
Estimated Time (ET)

\[ p_i = \frac{I_i \times c_i}{w_i + 1} \]
Estimated Time (ET)

\[ p_i = \frac{I_i \cdot c_i}{w_i + 1} \]

\[ p_1 = 40\mu s \quad p_2 = 10\mu s \quad p_3 = 15\mu s \quad p_4 = 70\mu s \quad p_5 = 100\mu s \]
Estimated Time (ET)

\[ p_i = \frac{I_i \ast c_i}{w_i + 1} \]

\[
p_1 = 40\mu s \quad p_2 = 10\mu s \quad p_3 = 15\mu s \quad p_4 = 70\mu s \quad p_5 = 50\mu s
\]
Estimated Time (ET)

\[ p_i = \frac{I_i \cdot c_i}{w_i + 1} \]

\( p_1 = 40\mu s \)
\( p_2 = 10\mu s \)
\( p_3 = 15\mu s \)
\( p_4 = 35\mu s \)
\( p_5 = 50\mu s \)
Estimated Time (ET)

\[ p_i = \frac{I_i \times c_i}{w_i + 1} \]

\[ p_1 = 40\mu s \quad p_2 = 10\mu s \quad p_3 = 15\mu s \quad p_4 = 35\mu s \quad p_5 = 33.3\mu s \]
Current Throughput (CT)

- Schedule the operator with lowest throughput as it is likely to be bottleneck in the pipeline
- Normalize for selectivity
- Divide time into windows of size $w$, compute “effective” number of tuples processed by operator in $w$
- Choose operator with smallest $n_i^w$

\[ n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cS_{i-1}} \]
Current Throughput (CT)

\[ n^w_i = \frac{T^w_i + (w_i \times s)}{c_i \times c s_{i-1}} \]
Current Throughput (CT)

\[ n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cs_{i-1}} \]

- \( n_1^w = 2M \)
- \( n_2^w = 0.33M \)
- \( n_3^w = 5M \)
- \( n_4^w = 0.1M \)
- \( n_5^w = 10M \)
Current Throughput (CT)

\[
n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times c_{s_i-1}}
\]

\[
\begin{align*}
  n_1^w &= 2M \\
  n_2^w &= 0.33M \\
  n_3^w &= 5M \\
  n_4^w &= 1M \\
  n_5^w &= 10M
\end{align*}
\]
Current Throughput (CT)

\[ n_i^w = \frac{T_i^w + (w_i \times s)}{c_i \times cS_{i-1}} \]

\[ n_1^w = 2M \]
\[ n_2^w = 0.75M \]
\[ n_3^w = 5M \]
\[ n_4^w = 1M \]
\[ n_5^w = 10M \]
Evaluation
Evaluation

Experimental Setup

• Intel Xeon E5 Family 2698B v3 series
• Windows Server 2012 R2 Datacenter
• 16 Physical Cores
• Cache sizes: 32KB, 256KB & 40MB
• Steady-state throughput, latency

Workloads

• TPCx-BB (Big Bench) Benchmark
• Modern Big Data Benchmark
• Q1-4, Q15 are streaming queries
• Eg. “Find top 30 products that are viewed together online”
• Micro-benchmarks
Scheduling - Throughput

The diagram shows the throughput in tuples per second for different query numbers (Query 1 to Query 15) and varying numbers of cores (2 to 16). The throughput is measured in tuples per second and is plotted on a logarithmic scale. The queries are evaluated using different scheduling methods labeled as CT, ET, QST, and LP.
Scheduling - Latency

The diagram shows the latency for different queries ranging from Query 1 to Query 15. The y-axis represents the log of latency in microseconds, while the x-axis represents the number of cores. Different lines and colors correspond to different scheduling techniques:

- CT (blue line and markers)
- ET (orange line and markers)
- QST (green line and markers)
- LP (red line and markers)

Each query's latency decreases as the number of cores increases, indicating efficient scaling of the scheduling algorithms.
Scheduling - Analysis

• Even when total worker time distribution is same, the “throughput” is different for different heuristics!

• Heuristics that distribute workers across the operators
  • Establish a continuous pipelined flow
  • Yields better throughput and latency

• Heuristics that focus on a single operator at a time
  • Prioritizes data parallelism over pipeline parallelism
  • Suffer from overheads of exploiting data parallelism in ordered setting
Partitioned Stateful Schemes

Load Imbalance Across Partitions

Hybrid strategy can afford finer partitions and hence better load balance!
Partitioned Stateful Schemes

Latency for different operator costs

Partitioned scheme blocks outputs, increasing latency!
Output Reordering Schemes

Lightweight Operators

High Selectivity Operators

Non-blocking reordering scheme prevents unnecessary worker blocking
Conclusion
Conclusion

• Framework for parallelizing ordered stream computations on shared-memory multicores

• Implementation of data-parallel operators in the ordered setting
  • Reordering outputs without worker blocking
  • Processing partitioned stateful operators in almost arrival order

• Proposed heuristics for dynamically scheduling stream operators and compared them empirically
Questions?