Scalable and Low-Latency Data Processing with StreamMapReduce

Andrey Brito\textsuperscript{1}, André Martin\textsuperscript{2}, Thomas Knauth\textsuperscript{2}, Stephan Creutz\textsuperscript{2}, Diogo Becker de Brum\textsuperscript{2}, Stefan Weigert\textsuperscript{2} and Christof Fetzer\textsuperscript{2}

\textsuperscript{1} Universidade Federal de Campina Grande, Brazil
\textsuperscript{2} Technische Universität Dresden, Germany

CLOUDCOM'11 – November, 29\textsuperscript{th} 2011
Outline

- Observations & Motivation
- Our Approach & Programming Model
- Evaluation
- Conclusion
- Q&A
MOTIVATION & OBSERVATIONS
Observations & Motivation

- **MapReduce** is a popular programming paradigm:
  - easy to understand: user provides simply a map and reduce function
  - scales well with problems sizes as well as computational resources
  - comes with implicit fault tolerance
  - **downside:** batch processing system – *response times > 30s*

- Recent trend:
  - Real Time Processing – requires *response times in (sub)seconds range*
  - Continues (infinite) stream of data
  - **Event Stream Processing** Systems seem to suitable as they provide
    - *response times in milliseconds range*, but
    - do not scale well
    - do not provide convenient programming interface such as MR
Observations & Motivation (cont.)

<table>
<thead>
<tr>
<th>Class</th>
<th>Response time</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>5 ms</td>
<td>Algorithmic trading</td>
</tr>
<tr>
<td>II</td>
<td>50 ms</td>
<td>Credit card fraud detection</td>
</tr>
<tr>
<td>III</td>
<td>500 ms</td>
<td>Mobile phone fraud detection</td>
</tr>
<tr>
<td>IV</td>
<td>5 s</td>
<td>Equipment/service monitoring</td>
</tr>
<tr>
<td>V</td>
<td>50 s</td>
<td>Continuous data analysis</td>
</tr>
<tr>
<td>VI</td>
<td>&gt; 500 s</td>
<td>Traditional data analysis (batch)</td>
</tr>
</tbody>
</table>
Observations & Motivation (cont.)

- **Challenges & Goals:** Combining MapReduce & ESP
  - simple programming interface as in MapReduce
  - scalable & high throughput data processing (as in MapReduce)
  - response times in sub-seconds range as in ESPs
OUR APPROACH & PROGRAMMING MODELL
Our Approach & Programming Model

- Keep the concept of Mappers and Reducers (plus Combiners)
  - extending it for ESP
- **Mappers** are stateless
- **Reducers** are statefull
  - sliding or jumping windows
  - windows are either time or counter based
- **Combiners** are statefull

- Different interfaces:
  - Simple MapReduce interface
  - Enhanced MapReduce interface for more efficient window computation
  - Enhanced MapReduce interface with time triggered reducer
Programming Model (cont’d)
Moving Average Example

Simple Interface

```java
// map1 := MAPPER
// reduce1 := REDUCER + JUMPING_WINDOW<1 minute>

type Key = ...
type Value = ...

map1(Key k, Value v) {
    // extract new key (k1) and value (v1) from (k,v)
    emit(k1, v1);
}

reduce1(Key k, List values) {
    int sum = 0, count = 0, v;
    for each v in values:
        sum += v; count++;
    emit(k, sum/count);
}
```

CLOUDCOM’11, Nov. 29th 2011   Scalable and Low-Latency Data Processing with Stream MapReduce
Programming Model (cont’d)
Moving Average Example

Simple Interface

```
// map1 := MAPPER
// reduce1 := REDUCER + SLIDING_WINDOW<1 minute, 6 sec>

map1(Key k, Value v) {
    // extract new key (k1) and value (v1) from (k,v)
    emit(k1, v1);
}

reduce1(Key k, List values) {
    int sum = 0, count = 0, v;
    for each v in values:
        sum += v; count++;
        emit(k, sum/count);
}
```
Programming Model (cont’d)

Moving Average Example

Enhanced Interface

```
reduce3 := REDUCER + SLIDING_WINDOW_INCREMENTAL<1 min, 6 seconds>

reduce3_init(k1) { // first time key k1 is received
    S = new State(); // custom user class
    S.sum = 0; S.count = 0;
    return S; // object S is now associated with key k1
}

reduce3(Key k, List newValues, List expiredValues, List commonValues, UserState S) {
    for v in expiredValues do:
        S.sum -= v; // Remove contribution of expired events
        S.count --;
    for v in newValues do: // Add contribution of new events
        S.sum += v
        S.count++;
    emit(k1, S.sum/S.count);
}
```
Programming Model (cont’d)
Moving Average Example

Enhanced Interface

```java
// reduce4:= REDUCER + STATEFUL
// ttreduce4:= TIMETRIGGERED_REDUCER<1 minute>

reduce4_init(Key k1) { // first time key k1 is received
    S = new State(); // custom user class
    S.sum = 0; S.count = 0;
    return S; // object S is now associated to key k1
}

reduce4(Key k, <Value v, UserState S>) {
    S.sum += v; S.count++;
}

ttreduce4(k1, <Value unused, UserState S>) {
    emit(k1, S.sum/S.count);
    S.sum = 0; S.count = 0;
}
```
Additional remarks

- Configuration annotations allows **optimization** regarding
  - fault tolerance (e.g. checkpointing on window boundaries)
  - load balancing (through operator migration)

- **Fault tolerance**
  - assume crash stop failure model
  - a combination of in-memory logging & checkpointing
  - precise recovery through **Virtual Synchrony** inspired deterministic exec:
    - events are assigned to epochs
    - within an epoch, events are processed out of order
    - checkpoints occur at epochs boundaries allowing precise recovery after a crash failure
    - see ICDCS2011 paper: *Low-overhead fault tolerance for high-throughput data processing systems*
EVALUATION
Experiments setup

- **48-nodes cluster** w/ 2x Intel Xeon E5405 (quad core) CPUs and **8GB** of RAM
- Gigabit Ethernet (1000BaseT full duplex) LAN

- Applications:
  - Community of Interests
  - SLA Conformance Monitoring
  - Word Count
Comparison Between Different Implementations of a Moving Average

![Comparison between different implementations of a moving average](chart)
Total Job Completion Times for Different Input Sizes of the Word Frequency Count Benchmark
Latency SLA & COI Application

![Graph 1](left): latency [ms] vs generator frequency [Hz]

![Graph 2](right): Δ vs frequency
Impact of Checkpointing the State in the Word Frequency Count Application

![Graph showing the impact of checkpointing on disk throughput per node in MB/s and aggregated throughput vs. number of nodes.](image-url)
Conclusion & Contribution

- Our approach combines responsiveness of ESP with scalability of MapReduce
- We introduced statefull reducers (as well as combiners)
- Provide low overhead fault tolerance with precise recovery due to virtual synchrony deterministic execution
THANK YOU
FOR YOUR UNDIVIDED ATTENTION