Integrating Data-Parallel Analytics into Stream-Processing Using an In-Memory Data Grid

Dr. William L. Bain
ScaleOut Software, Inc.
Dr. William Bain, Founder & CEO of ScaleOut Software:

- Email: wbain@scaleoutsoftware.com
- Ph.D. in Electrical Engineering (Rice University, 1978)
- Career focused on parallel computing – Bell Labs, Intel, Microsoft
- 3 prior start-ups, last acquired by Microsoft and product now ships as Network Load Balancing in Windows Server

ScaleOut Software develops and markets **In-Memory Data Grids**, software for:

- Scaling application performance with in-memory data storage
- Analyzing live data in real time with in-memory computing

- Thirteen+ years in the market; 450+ customers, 12,000+ servers

---

In-Memory Computing Summit North America 2018
How In-Memory Computing Creates the Next Generation in Stream-Processing

• Goals and challenges for stream-processing
• Adding context: stateful stream-processing
• Overview of in-memory data grids (IMDGs)
• Digital twin model for stateful stream-processing
• Why use an IMDG: integrated event processing and data-parallel analysis
• Example use cases
• Detailed code sample: runners with smart watches
• Performance benefits
Goals for Stream-Processing

• **Goals:**
  • Process incoming data streams from many (1000s) of sources.
  • Analyze events for patterns of interest.
  • Provide timely (real-time) feedback and alerts.
  • Provide data-parallel analytics for aggregate statistics and feedback.

• **Many applications:**
  • Internet of Things (IoT)
  • Medical monitoring
  • Logistics
  • Financial trading systems
  • Ecommerce recommendations

Event Sources
Example: Ecommerce Recommendations

1000s of online shoppers:

- Each shopper generates a clickstream of products searched.
- Stream-processing system must:
  - Correlate clicks for each shopper.
  - Maintain a history of clicks during a shopping session.
  - Analyze clicks to create new recommendations within 100 msec.
- Analysis must:
  - Take into account the shopper’s preferences and demographics.
  - Use aggregate feedback based on collaborative shopping behavior.
Providing Recommendations in Real Time

• Requires scalable stream-processing to analyze each click and respond in <100ms:
  • Accept input with each event on shopper’s preferences.
  • Provide aggregate feedback on best-selling products.
Providing Aggregate Metrics

• Must aggregate statistics for all shoppers:
  • Track real-time shopping behavior.
  • Chart key purchasing trends.
  • Enable merchandizer to create promotions dynamically.

• Aggregate statistics can be shared with shoppers:
  • Allows shoppers to obtain collaborative feedback.
  • Examples include most viewed and best selling products.
Challenges for Stream-Processing Architectures

• Basic stream-processing architecture:

Data Sources

Source A ➔ ... ➔ Event 3 ➔ Event 2 ➔ Event 1

Source B ➔ ... ➔ Event 3 ➔ Event 2 ➔ Event 1

Source C ➔ ... ➔ Event 3 ➔ Event 2 ➔ Event 1

Events

Event 3 ➔ Event 2 ➔ Event 1

Stream Processing Pipeline

Source ➔ Step 1 ➔ Data Stream ➔ Step 2 ➔ Sink

• Challenges:
  • How efficiently correlate events from each data source?
  • How combine events with relevant state information to create the necessary context for analysis?
  • How embed application-specific analysis algorithms in the pipeline?
  • How generate feedback/alerts with low latency?
  • How perform data-parallel analytics to determine aggregate trends?
Adding Context to Stream-Processing

• Stateful stream-processing platforms add “unmanaged” data storage to the pipeline:
  • Pipeline stages perform transformations in a sequence of stages from data sources to sinks.
  • Data storage (distributed cache, database) is accessed from the pipeline by application code in an unspecified manner.
  • Examples: Apama (CEP), Apache Flink, Storm

• Problems:
  • There is no software architecture for managing state information.
  • This adds complexity to the application.
  • Creates a network bottleneck.
  • Does not address need for data-parallel analytics.
Lambda Architecture: Batch Parallel Analytics

• Lambda architecture separates stream-processing ("speed layer") from data-parallel analytics ("batch layer").

• Creates queryable state, but:
  • Does not enhance context for stateful stream processing.
  • Does not perform data-parallel analytics online for immediate feedback.
  • Does not lead to a "Hybrid Transactional and Analytics Processing" (HTAP) architecture.

How combine stream-processing with state to simplify design, maximize performance, and enable fast data-parallel analytics?

https://commons.wikimedia.org/w/index.php?curid=34963987
In-Memory Data Grid (IMDG)

IMDG provides a powerful platform for stateful stream-processing.

What is an IMDG?

• IMDG stores live, object-oriented data:
  • Uses a key/value storage model for large object collections.
  • Maps objects to a cluster of commodity servers with location transparency.
  • Has predictably fast (<1 msec.) data access and updates.
  • Designed for transparent scaling and high availability

• IMDG integrates in-memory computing with data storage:
  • Uses object-oriented execution model.
  • Leverages the cluster’s computing power.
  • Computes where the data lives to avoid network bottlenecks.
How an IMDG Can Integrate Computation

• Each grid host runs a worker process which executes application-defined methods on stored objects.
  • The set of worker processes is called an *invocation grid (IG)*.
  • IG usually runs language-specific runtimes (JVM, .NET).
  • IMDG can ship code to the IG workers.

• Key advantages for IGs:
  • Follows object-oriented model.
  • Avoids network bottlenecks by moving computing to the data.
  • Leverages IMDG’s cores & servers.
IMDG Runs Event Handlers for Stream-Processing

Event handlers run independently for each incoming event:

- IMDG directs event to a specific object using ReactiveX for low latency.
- IMDG executes multiple event handlers in parallel for high throughput.
IMDG Executes Data-Parallel Computations

Method execution implements a parallel op. on an object collection:

• Client runs a single method on all objects in a collection.
• Execution runs in parallel across the grid.
• Results are merged and returned to the client.
• Runs with lower latency than batch jobs.
A Basic Data-Parallel Execution Model

A fundamental model from parallel supercomputing:

• Run one method (“eval”) in parallel across many data objects.

• Optionally **merge** the results.
  • Binary combining is a special case, but...
  • It runs in logN time to enable scalable speedup
MapReduce Builds on This Model

• Implements “group-by” computations.

• Example: “Determine average RPM for all windmills by region (NE, NW, SE, SW).”

• Runs in two data-parallel phases (map, reduce):
  • **Map** phase repartitions and optionally combines source data.
  • **Reduce** phase analyzes each data partition in parallel.
  • Returns results for each partition (no merging).
Distributed ForEach: Another Data-Parallel Model

• Body code performs **eval** and iterative **merge** to reduce garbage collection:
Reduced GC Time with Distributed ForEach

PMI

Distributed ForEach
Stream-Processing with the Digital Twin Model

• Created by Michael Grieves; popularized by Gartner
• Represents each data source with an IMDG object that holds:
  • An event collection
  • State information about the data source
  • Logic for analyzing events, updating state, and generating alerts
• Benefits:
  • Offers a structured approach to stateful stream-processing.
  • Automatically correlates incoming events by data source.
  • Integrates all relevant context (events & state).
  • Enables easy deployment of application-specific logic (e.g., ML, rules engine, etc.) for analysis and alerting.
  • Provides domain for aggregate analysis and feedback.
Some Applications for Digital Twins

A digital twin correlates incoming events with context using domain-specific algorithms to generate alerts:

<table>
<thead>
<tr>
<th>Application</th>
<th>Context</th>
<th>Events</th>
<th>Logic</th>
<th>Alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoT devices</td>
<td>Device status &amp; history</td>
<td>Device telemetry</td>
<td>Analyze to predict maintenance.</td>
<td>Maintenance requests</td>
</tr>
<tr>
<td>Medical monitoring</td>
<td>Patient history &amp; medications</td>
<td>Heart-rate, blood-pressure, etc.</td>
<td>Evaluate measurements over time windows with rules engine.</td>
<td>Alerts to patient &amp; physician</td>
</tr>
<tr>
<td>Cable TV</td>
<td>Viewer preferences &amp; history, set-top box status</td>
<td>Channel change events, telemetry</td>
<td>Cleanse &amp; map channel events for reco. engine; predict box failure.</td>
<td>Viewer recommendations, repair alerts</td>
</tr>
<tr>
<td>Ecommerce</td>
<td>Shopper preferences &amp; buying history</td>
<td>Clickstream events from web site</td>
<td>Use ML to make product recommendations.</td>
<td>Product list for web site</td>
</tr>
<tr>
<td>Fraud detection</td>
<td>Customer status &amp; history</td>
<td>Transactions</td>
<td>Analyze patterns to identify probable fraud.</td>
<td>Alerts to customer &amp; bank</td>
</tr>
</tbody>
</table>
Why Use an IMDG to Host Digital Twins?

IMDG provides an excellent DT platform:

- Scalable, object-oriented data storage:
  - Offers a natural model for hosting digital twins.
  - Cleanly separates domain logic from data-parallel orchestration.

- Integrated, In-memory computing:
  - Automatically correlates incoming events for analysis.
  - Enables both stream and data-parallel processing.

- High performance:
  - Avoids data motion and associated network bottlenecks.
  - Fast and scales to handle large workloads.

- Integrated high availability:
  - Uses data replication designed for live systems.
  - Can ensure that computation is high av.
Scaling Event Ingestion with Kafka

- IMDG partitions digital twin objects across servers.
- Kafka offers partitions to scale out handling of event messages.
  - Partitions are distributed across brokers.
  - Brokers process messages in parallel.
- IMDG can map Kafka partitions to grid partitions:
  - IMDG specifies event-mapping algorithm to Kafka.
  - IMDG listens to appropriate Kafka partitions.
- **This minimizes event handling latency.**
  - Avoids store-and-forward within IMDG.
Integrating Event and Data-Parallel Processing

The IMDG:

• Posts incoming events to its respective digital twin object.

• Runs the twin’s event handler method with low latency.
  • Event handler manages the event collection and can use time windows for analysis.
  • Event handler uses and updates in-memory state.
  • Event handler can use/update off-line state.
  • Event handler optionally generates alerts and feedback to its data source.

• Runs data-parallel methods to analyze all digital twins in real-time.
  • Results can be used for both alerting and feedback.
Example: Ecommerce Shopping Site

Tracks web shoppers and provides real-time recommendations:

• Each DT object holds clickstream of browsed products, preferences, and demographics.

• Event handler analyzes this data and updates recommendations.

• Periodic data-parallel, batch analytics across all shoppers determine aggregate trends:
  • Examples include best selling products, average basket size, etc.
  • Used for analysis and real-time feedback

In-Memory Computing Summit North America 2018
Example: Tracking a Fleet of Vehicles

- **Goal**: Track telemetry from a fleet of cars or trucks.
  - Events indicate speed, position, and other parameters.
  - Digital twin object stores information about vehicle, driver, and destination.
  - Event handler alerts on exceptional conditions (speeding, lost vehicle).

- **Periodic data-parallel analytics** determines aggregate fleet performance:
  - Computes overall fuel efficiency, driver performance, vehicle availability, etc.
  - Can provide feedback to drivers to optimize operations (e.g., avoid congested areas).
Using Digital Twins in a Hierarchy

Tracks complex systems as hierarchy of digital twin objects:

• Leaf nodes receive telemetry from physical endpoints.

• Higher level nodes represent subsystems:
  • Receive telemetry from lower-level nodes.
  • Supply telemetry to higher-level nodes as alerts.
  • Allow successive refinement of real-time telemetry into higher-level abstractions.

Example: Hierarchy of Digital Twins for a Windmill
OOP Techniques Simplify Building Digital Twins

• Digital twin objects can use inheritance to create specialized behaviors:

  - Instances of objects can be organized in a hierarchy:
Detailed Example: Heart-Rate Watch Monitoring

Goal: Track heart-rate for a large population of runners.

- Heart-rate events flow from smart watches to their respective digital twin objects for analysis.
- The analysis uses wearer’s history, activity, and aggregate statistics to determine feedback and alerts.
Digital Twin Object (Java)

- Holds event collection and user’s context (age, medical history, current status, etc.):

```java
public class User implements Serializable {
    private int _id;
    private double _height;
    private double _bodyWeight;
    private Gender _gender;
    private int _age;
    private int _averageHr;
    private WorkoutProgress _status;
    private int _sessionAverageMax;
    private List<Medication> _medications;
    private List<Long> _heartIncidents;
    private List<HeartRate> _runningHeartRateTelemetry;
    private long _alertTime;
    private boolean _alerted;
    ...
}
```

User’s context

Event collection
Events & Alerts

• Event holds periodic telemetry sent from watch to IMDG:

```java
public class HeartRateEvent {
    private int _userId;
    private int _heartRate;
    private long _timestamp;
    private WorkoutType _workoutType;
    private WorkoutProgress _workoutProgress;
    private Event _event;
    ...
}
```

• Alert holds data to be sent back to wearer and/or to medical personnel:

```java
public class HeartRateAlert {
    private int _userId;
    private String _alertType;
    private String _params;
    ...
}
```
Define a ReactiveX observer that runs on every server in the IMDG:

```java
public class HeartRateObserver implements Observer<Event>, Serializable {
    @Override public void onNext(Event event) {
        HeartRateEvent hre = HeartRateEvent.fromBytes(event.getPayload());
        hre.setEvent(event);
        User.processRunningEvent(hre);
    }
}
```

Create an invocation grid that Initializes the ReactiveX observer at startup:

```java
Pipeline pipeline = new Pipeline("userCache", "userGrid");
GridAction action = pipeline.createRemoteObserverAction("userObserver",
            new HeartRateObserver());
InvocationGrid grid = new InvocationGridBuilder("userGrid")
    .addJar("./bin/appcode.jar")
    .addStartupAction(action)
    .load();
```
Event Handler and Event Posting

• Posting an event to the ReactiveX observer:
  • The key determines which server receives the event for posting.

```java
pipeline.postEvent(makeKey(UserId),"heartRateEvent", HeartRateEvent.toBytes(
    new HeartRateEvent(last, System.nanoTime(),
    WorkoutType.Running, WorkoutProgress)));
```

• Handling an event posted to the ReactiveX observer on DT twin’s server:

```java
private static void processRunningEvent(HeartRateEvent hre) {
    CachedObjectId id = hre.getId();
    User u = (User)cache.retrieve(id, false);
    ...
    executeRunningWorkoutAnalytics(hre, u);
    ...
    cache.update(id, u);}
```
Event Analysis

• Handles an event for an active user doing a running workout:

```java
private static void executeRunningWorkoutAnalytics(HeartRateEvent hre, User u) {
    long start = twoWeeksAgo();
    long sessionTimeout = threeHours();
    SessionWindowCollection<HeartRate> swc = new SessionWindowCollection<>(u.getRunningHeartRateTelemetry(),
        heartRate -> heartRate.getTimestamp(), start, sessionTimeout);
    swc.add(new HeartRate(hre.getHeartRate(), hre.getTimestamp()));

    int total = 0; int windowCount = 0;
    for(TimeWindow<HeartRate> window : swc) {
        int avg = 0;
        for(HeartRate hr : window) {avg += hr.getHeartRate();}
        total += (avg/window.size());
        windowCount++;
    }
    u.setAverageHr(total/windowCount);
    u.analyzeAndCheckForAlert(hre);
}
```
Analysis Techniques Enabled by Digital Twin

Enable detailed heart-rate monitoring for a high intensity exercise program:

• Example of data to be tracked:
  • **Exercise specifics**: type of exercise, exercise-specific parameters (distance, strides, altitude change, etc.)
  • **Participant background/history**: age, height, weight history, heart-related medical conditions and medications, injuries, previous medical events
  • **Exercise tracking**: session history, average # sessions per week, average and peak heart rates, frequency of exercise types
  • **Aggregate statistics**: average/max/min exercise tracking statistics for all participants

• Example of logic to be performed:
  • **Notify participant** if session history across time windows indicates need to change mix.
  • **Notify participant** if heart rate trends deviate significantly from aggregate statistics.
  • **Alert participant/medical personnel** if heart rate analysis across time windows indicates an imminent threat to health.
  • **Report** aggregate statistics to analysts and/or users.
Data Parallel Analysis Across all Digital Twins

- Uses IMDG’s in-memory compute engine to create aggregate statistics in real time.

- Results can be reported to analysts and updated every few seconds.

- Results can be used as feedback to event analysis in digital twin objects and/or reported to users.
Computing Aggregate Data

- Performs a data-parallel computation using the IMDG’s Eval and Merge methods:

```java
public class AggregateStatsInvokable implements Invokable<User, Integer,
    AggregateStats> {
    @Override
    public AggregateStats eval(User u, Integer numUsers) {
        AggregateStats userStats = new AggregateStats(numUsers);
        userStats.merge(u);
        return userStats;
    }

    @Override
    public AggregateStats merge(AggregateStats mergedStats,
        AggregateStats u) {
        mergedStats.merge(u);
        return mergedStats;
    }
}
```

Eval method

Binary merge method
Computing Aggregate Data (2)

• Computes running average of heart-rate by categories:

```java
public void merge(AggregateStats user) {
    numEvents += user.getNumEvents();
    totalHeartRate18to34 += user.getTotalHeartRate18to34();
    totalHeartRate35to50 += user.getTotalHeartRate35to50();
    totalHeartRateOver50 += user.getTotalHeartRateOver50();
    count18to34 += user.getCount18to34();
    count35to50 += user.getCount35to50();
    countOver50 += user.getCountOver50();

    totalHeartRateBmiUnderWeight += user.getTotalHeartRateBmiUnderWeight();
    totalHeartRateBmiNormalWeight += user.getTotalHeartRateBmiNormalWeight();
    totalHeartRateBmiOverweight += user.getTotalHeartRateBmiOverweight();
    countUnderweight += user.getCountUnderweight();
    countNormalWeight += user.getCountNormalWeight();
    countOverWeight += user.getCountOverWeight();
}
```
Running the Data-Parallel Computation

- Uses a single method to run a data-parallel computation and return results.
- Publishes merged results to an IMDG object for access by user objects and/or analysts.

```java
public void run() {
    NamedCache usersCache = CacheFactory.getCache("userCache");
    NamedCache statsCache = CacheFactory.getCache("statsCache");
    AggregateStats stats;

    InvokeResult<AggregateStats> result =
        usersCache.invoke(AggregateStatsInvokable.class, null, _numUsers,
                          TimeSpan.fromMilliseconds(10000));

    stats = result.getResult();
    statsCache.put("globalStats", stats);
}
```
Data-Parallel Execution Steps

• **Eval** phase: each server queries local objects and runs eval and merge methods:
  • Accessing local objects avoids data motion.
  • Completes with one result object per server.

• **Merge** phase: all servers perform binary, distributed merge to create final result:
  • Merge runs in parallel to minimize completion time.
  • Returns final result object to client.
Predictable, Scalable Performance

- Digital twin model enables the IMDG to scale both event-handling and integrated data-parallel analysis.
  - Correlating events to digital twin objects creates an automatic basis for performance scaling:
    - For event analysis
    - For data-parallel analysis
  - It enables access to each event’s context without requiring a network access.
  - It also co-locates and encapsulates application-specific code using o-o techniques.
Avoids Network Bottlenecks

• Digital twin model avoids network bottlenecks associated with using an IMDG as a networked cache in a stream-processing pipeline.
  • External data storage requires network access to obtain an event’s context.
  • Network bottleneck prevents scalable throughput.

Stream Pipeline

![Diagram of Stream Pipeline with Source, Operation, Data Stream, and Sink]

PMI vs. Random Access Throughput Comparison

![Graph comparing PMI and Random Access Throughput]
Digital Twins: The Next Generation in Stateful Stream-Processing

• **Challenge**: Current techniques for stateful stream-processing:
  • Lack a coherent software architecture for managing context.
  • Can suffer from performance issues due to network bottlenecks.

• **The digital twin model**:
  • Offers a flexible, powerful, scalable architecture for stateful stream-processing:
    • Associates events with context about their physical sources for deeper introspection.
    • Enables flexible, object-oriented encapsulation of analysis algorithms.
  • Provides a basis for aggregate analysis and feedback.

• **Scalable, data-parallel computing with an IMDG**:
  • Automatically correlates incoming events and processes them in parallel.
  • Implements integrated (real-time), aggregate analysis for immediate feedback.
In-Memory Computing for Operational Intelligence

www.scaleoutsoftware.com