Event Processing over a Distributed JSON Store: Design and Performance

Miki Enoki[1], Jérôme Siméon[2], Hiroshi Horii[1], and Martin Hirzel[2]

[1] IBM Research, Tokyo, Japan
{enomiki,horii}@jp.ibm.com
[2] IBM Research, New York, USA
{simeon,hirzel}@us.ibm.com

Abstract. Web applications are increasingly built to target both desktop and mobile users. As a result, modern Web development infrastructure must be able to process large numbers of events (e.g., for location-based features) and support analytics over those events, with applications ranging from banking (e.g., fraud detection) to retail (e.g., just-in-time personalized promotions). We describe a system specifically designed for those applications, allowing high-throughput event processing along with analytics. Our main contribution is the design and implementation of an in-memory JSON store that can handle both events and analytics workloads. The store relies on the JSON model in order to serve data through a common Web API. Thanks to the flexibility of the JSON model, the store can integrate data from systems of record (e.g., customer profiles) with data transmitted between the server and a large number of clients (e.g., location-based events or transactions). The proposed store is built over a distributed, transactional, in-memory object cache for performance. Our experiments show that our implementation handles high throughput and low latency without sacrificing scalability.

Keywords: Events Processing, Analytics, Rules, JSON, In-Memory Database.

1 Introduction

In-memory computing reverses traditional data processing by embedding the compute where data is stored, instead of moving the data to where the compute happens. Analysts predict the market for in-memory computing to grow 35% between now and 2015 [18]. This is in part because in-memory computing can radically change the time-frame for completing some data analysis tasks. In addition, it allows for systems to combine on-the-fly processing of events with analytics, which are increasingly important for business applications. We describe a system designed around a distributed in-memory data store that supports such applications. Our work is part of a broader project, *Insight to Action* (I2A), whose goal is to support high-throughput transaction processing along with decision capabilities based on large-scale distributed analytics.

Event streaming systems have been a subject of intense research and development in the last 10 years. Existing distributed event streaming systems include both research prototypes [1,5] and commercial products [12,20]. Those systems usually operate with a limited amount of state, and allow users to combine operators that process one or

B. Benatallah et al. (Eds.): WISE 2014, Part II, LNCS 8787, pp. 395–404, 2014.

more input events before passing a new event to the next operator. The I2A architecture is organized around an agent model in which each agent can process a fraction of the incoming events. One key difference compared to prior event-based systems is the presence of a distributed in-memory data store that can be accessed by those agents. The store is used both to allow agents to hold on to some state (e.g., information about a specific customer or product useful when processing a specific event), as well as to enable analytics that can inform decisions on those events (e.g., by computing the average order amount for customers in the same area).

This paper focuses on I2A's distributed store, which is one of the main novel parts of the overall system. The store handles both event storage as well as data obtained from systems of record (e.g., relational backends). Another novelty is the store's ability to handle flexible data through the JSON format. This is particularly important in this context as information is usually obtained from existing and usually heterogeneous data sources. Furthermore, it allows to easily blend events with stored data.

This paper makes the following technical contributions:

- It describes a system that supports combined workloads for events and analytics over an in-memory distributed JSON store.
- It illustrates the benefits of the JSON data model to unify representations on the client, on the wire, and in the store, and to support better flexibility in the system.
- It describes the architecture for the distributed JSON store, which leverages the MongoDB API [14] on top of an in-memory distributed object cache (WXS [22]).
- It reports experimental results on throughput, latency, and scalability.

2 Insight to Action

This section provides background on the Insight to Action (I2A) system on top of which the contributions of this paper rest.

Processing events with analytics. The starting point for this paper is the I2A system, which is currently under development at IBM. I2A combines event processing with analytics over a distributed store. The idea behind the name is that analytics discover insights and event processing performs actions. Both analytics and event processing use the same distributed in-memory store. Combining both events and analytics in a single system fosters ease of use (no need to configure multiple systems) and performance (no need to move data back and forth).

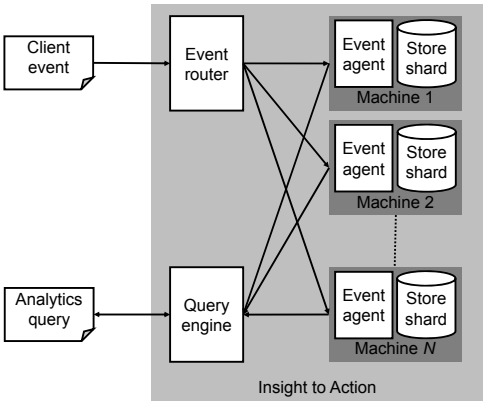

Fig. 1. Architecture for processing events and analytics over a distributed in-memory store

```
rule NormalTakeOff {
  when {
    toe: TakeOffEvent(sched+20 >= now);
    flight: toe.flight();
  } then {
    update flight.departed = true;
    update flight.on_time = true;
  }
}
rule LateTakeOff {
  when {
    toe: TakeOffEvent(sched+20 < now);
    flight: toe.flight();
  } then {
    update flight.departed = true;
    update flight.on_time = false;
    emit new FlightDelayEvent(
      flight.fl_no());
  }
}
```

Fig. 2. Sample rules for I2A event processing

```
rule RAvg {
  when {
    late_count: aggregate {
        f: Flight(on_time = false);
        fn: f.fl_no();
      }
      groupby { fn }
      do { count { f }; }
  } then {
    insert new FlightStat(
      fn, late_count);
  }
}
```

Fig. 3. Sample rule for I2A analytics

The top part of Fig. 1 illustrates event processing in I2A. Each arriving client event contains a key associating it with an entity. Each machine runs a store shard storing entities, as well as an event agent. The system uses the key in the event to route it to the machine where the corresponding entity is stored. The event agent acts upon the event by reading and writing entities in the local store shard, and by emitting zero or more output events (not shown in the figure).

The bottom part of Fig. 1 illustrates analytics in I2A. Analytics can be either user-initiated, or scheduled to repeat periodically. In the latter case, the time period between analytics recomputation is on the order of minutes, in contrast to the expected latency for event processing, which is in the sub-second range. That is because an analytics query typically scans all entities on all machines, unlike an event, which only accesses a few entities on a single machine. A query engine coordinates the distributed analytics, and combines the results (i.e., insight). The results are then either reported back to the user, or saved in the store for use by future events (i.e., action). While this section deliberately does not specify the data formats used, Section 3 will argue that JSON is a good choice.

Scenario. To illustrate I2A, we consider a simple example inspired by a real scenario from the airline industry. For each flight, the system receives events, such as when passengers book or cancel, when the crew is ready, when the flight takes off, etc. For each of those events, an agent updates the flight entity in its local store shard. Agents are written using JRules [21]. Due to space limitations, we only show simple rules that illustrate the main features: how to process events, and how to run analytics.

The rules in Fig. 2 implement the agent handling events for a flight taking off. JRules uses a condition-action model similar to that of production systems [10]. The condition is inside the `when` clause of the rule, and the action in the `then` clause. The first example applies to take-off events that were scheduled at most 20 minutes before their actual time (`sched+20 >= now`), while the second applies when the take-off was at least 20 minutes late. In both cases, the action part of the rule updates the flight information with the

corresponding status, storing it in the shard so that other rules may access it. The second rule also emits a new `FlightDelayEvent` that can be used by other rules, for instance, to notify passengers or handle connecting flights.

Periodic analytics jobs can be run over all the data in the store, e.g., to compute summary information about all flights. For instance, the rule in Fig. 3 computes the number of late departures for every flight number. The rule can access all flight information in the store and compute aggregation using the **`aggregate`** expression, which here groups information by flight number, for flights that have been marked as delayed. This summary information is then replicated back out to each of the shards, making it available for future event processing. For instance, if a particular flight is frequently delayed, the can trigger a review, or it can be taken into account when re-booking passengers. As a more complex example, assume that the system receives an event about a passenger missing their flight. The agent handling this event can consult available summary status information to look for alternative flights. The system reports those alternatives back to the passenger, who can then re-book, leading to another event.

While this particular example focuses on the airlines industry, I2A is obviously not restricted to this domain. There are many use cases in different industries where insights from distributed analytics can drive actions in low-latency event processing.

3 Leveraging JSON

One important aspect of our work is the built-in support for JSON as a data model throughout the system, and how it allows to more easily integrate new information as it becomes available.

JSON End-to-End. The JavaScript Object Notation (JSON) has gained broad acceptance as a format for data exchange, often replacing XML due to its relative simplicity and compactness. Originally used for its easy integration with the client through JavaScript, it is now increasingly also used in both the back-end, e.g., through JSON databases [14], and the middleware, e.g., through Node.js [15].

Fig. 4 shows where the MongoDB API is integrated inside the I2A architecture. Both the event agent and the query engine can access data in the store through that JSON-centric API. From an application's perspective, the system can be seen as a JSON-centric development platform, which can load JSON data into the store, process incoming JSON events (e.g., through a REST API), execute rules on those events, and respond with JSON events. Fig. 4 also

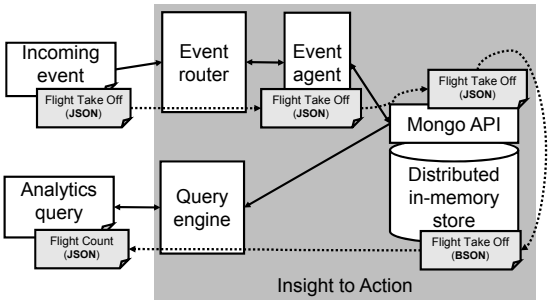

Fig. 4. JSON support and usage inside I2A

shows the life-cycle of the JSON data from an incoming event (flight take-off), which is then routed to the appropriate event agent, stored, and possibly queried back.

Because both external data and internal representation use JSON, we can pass most of the data through the system without expensive conversions common in similar architectures (to Java POJOs, to records in a relational store, etc.). Such conversions are particularly costly in distributed systems, which often need to (de)serialize objects for communication. Our architecture avoids most of that cost.

Flexibility. Another benefit of JSON is its flexibility, building on previous approaches for semi-structured data and Web data exchange formats such as XML. Because it does not require the user to declare the content of the documents or messages ahead of time, JSON supports data with some level of heterogeneity (e.g., missing or extra fields), and to extend the data with new information as it becomes available without having to recompile or deploy the application, or without having to reload the data.

In our example, the following JSON documents may correspond respectively to the delayed departure event and to the entities for the airports of departure and arrival. Some of those events and entities may include distinct information, for instance, the US airports include the state. If an event has extra fields not needed by a rule, the rule fires as normal. If an event lacks fields required by a rule, the rule simply does not fire.

```
event { fl_no: "II32", from: "JFK", to: "CDG", departed: "21:43 GMT", on_time: false }
event { fl_no: "II32", from: "JFK", to: "CDG", landed: "04:56 GMT", on_time: false }
airport { code: "JFK", country: "USA", state: "NY", city: "New York" }
airport { code: "CDG", country: "France", city: "Paris" }
```

Besides entities required for the applications, such as airport data in our example, the store records events as they are being processed, providing historical records that can be useful for analytical purposes. As the application evolves, there is often a need to integrate more information. In our example application, we can imagine that we might want to include weather information, or twitter data that may be used for notification. External services often provide data in an XML or JSON format, which can be easily integrated into the I2A architecture, and stored directly in the I2A distributed repository. For instance, the following JSON document may be provided by a location-based service providing weather alerts.

```
alert { weather: "Snow", location: "New York", until: "19:00 GMT" }
```

That information can then be used to augment events being returned by the system. For instance, the delayed departure event could be enriched with information indicating that the reason for the delay is weather-related, with the corresponding alert information transmitted back to the user.

4 Integrated Distributed JSON Store

This section introduces the architecture of the distributed JSON store for I2A.

4.1 MongoDB

MongoDB is an open-source document-oriented database for JSON [14]. Internally it uses BSON (Binary JSON), a binary-encoded serialization of JSON documents. Developers interact with MongoDB by using a language driver (supported languages include Java, C/C++, Ruby, and NodeJS). Clients communicate with the database server through a standard TCP/IP socket. MongoDB supports only atomic transactions on individual rows, and it scales horizontally based on auto-sharding across multiple machines.

4.2 Integration with Insight to Action

JSON Store for I2A. Though MongoDB supports a distributed JSON store, it does not have the full transactional semantics that I2A requires for consistent event processing. Our challenge is how to reuse MongoDB's API without losing the transactionality and scalability advantages. Therefore, instead of using MongoDB directly, we used WebSphere eXtreme Scale (WXS [22]) as a distributed in-memory store with the MongoDB API, as shown in Fig. 5 (a), which depicts more details of each of the machines shown in Fig. 1. WXS is an elastic, scalable, in-memory key/value data store. It dynamically caches, shards, replicates, and manages the application data and business processing across multiple servers.

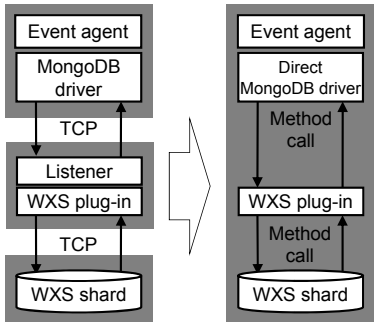

Fig. 5. From external to embedded interactions using the MongoDB interface over the WXS store

Fig. 6 shows a sequence diagram for a read query by an event agent. A query is described using a JSON format such as {"name": "Miki Enoki", "department": "S77" }. The query selects the JSON documents that contain "Miki Enoki" as the name and "S77" as the department. The event agent communicates with the listener using the MongoDB driver for TCP and sends the query as serialized BSON data to the listener. The listener is a server application on WXS that intercepts the MongoWire Protocol messages. The query is deserialized as BSON data by the listener and then converted into a query for the WXS Object Query API to read the response data from the WXS shards by using the WXS plug-in. In the WXS shards, each query agent processes a query and then returns the requested data via TCP. The results are collected by the WXS plug-in, and serialized for transmission to the event agent.

WXS supports sharding for a distributed store, which allows running multiple WXS shards within multiple machines in an I2A instance for scaling out with sharded data. With a distributed store, each query of the event agent has to be routed to the corresponding WXS shard to access the appropriate data.

Embedded JSON Store. In I2A, since high-performance data processing is important for both event processing and analytics, we developed the embedded store shown in Fig. 5 (b) by eliminating the TCP communications from Fig. 5 (a). Even without eliminating the TCP communications, performance is improved when the event agent

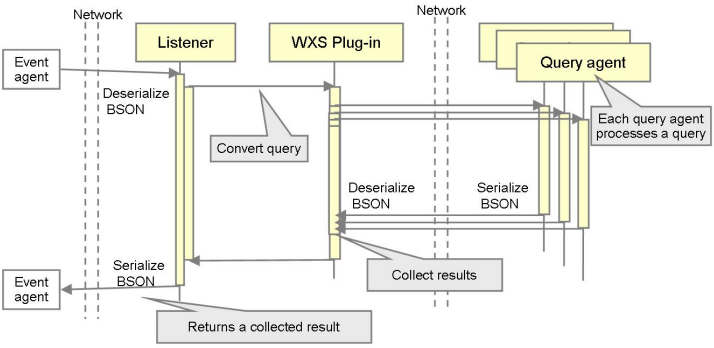

Fig. 6. Sequence Diagram of Query Processing

accesses only the WXS shards on its own machine. This assumption is true thanks to the fundamental design of I2A: since the I2A architecture includes an event router, we do not need to fall back on WXS for routing to the proper shard. In addition to insuring the communications are local, additional performance improvement comes from eliminating the TCP connections between the clients and the listener, and between the listener and the WXS store. Without the TCP connections, we can avoid the data serialization, which was needed to convert the BSON objects into byte array objects and vice versa. We wrote a direct MongoDB driver (class `DirectDB`) as a replacement for class `com.mongodb.DB`. The original class `com.mongodb.DB` sends query request to the Listener via TCP. Our replacement class `DirectDB`, in contrast, directly hands query requests to the WXS plug-in via a method call. WXS is also written in Java, so we can run all of the components in one JVM by using method calls in each machine in the cluster, as shown in Fig. 5 (b). Similarly, the event agent can handle low-latency read and write events by using method calls.

5 Experimental Evaluation

This section explores the throughput, latency, and scale-out of our JSON store for I2A.

Evaluation Methodology. We evaluated the effectiveness of our embedded JSON store with YCSB (Yahoo! Cloud Serving Benchmark) [8]. YCSB is a framework and common set of workloads for evaluating the performance of various key-value stores. I2A routes events to WXS shards storing the corresponding entity based on an entity key in the event. Therefore, we use YCSB to emulate an event processing workload. Some parameters are configurable by the user, such as the number of records, number of operations, or the read/update ratio. We fixed the number of records and operations to 100,000 and 5,000,000, respectively, and experimented with varying read/update ratios. As described in Section 4, events are processed through the MongoDB API in the JSON store, so we used the MongoDB driver included in YCSB for the benchmark client. The key and value are document ID and its JSON document respectively. For the embedded JSON store, we used our Direct MongoDB driver to directly access WXS with an internal method call. The environment was as follows: a 2-CPU Xeon X5670 (2.93GHz,

L1=32KB, L2=256KB, L3=12MB, 6 cores) with 32 GB of RAM and Red Hat Linux 5.5. We installed WXS version 8.6.0.2 as the JSON store.

Throughput and Latency. We compared the throughput and latency of the embedded JSON store (Fig. 5 (b)) to the original (Fig. 5 (a)) with a single shard. The number of client threads was changed from 1 to 200. The highest throughput is shown in the Fig. 7. We experimented with three read/update ratios 50/50, 95/5, and 100/0, which correspond to the YCSB official workloads a, b, and c, respectively. I2A handles the events for event processing and analytics. The event agent updates the entities in the JSON store corresponding to the incoming events, and then reads the related entities or the analytics results. For the analytics, the events in the store are read periodically. Therefore, the data access pattern of I2A is read intensive. As seen in Fig. 7, the throughput of the embedded store was better for all read/update ratios. In particular, the result for the read-only scenario was about 4.8 times higher than the original.

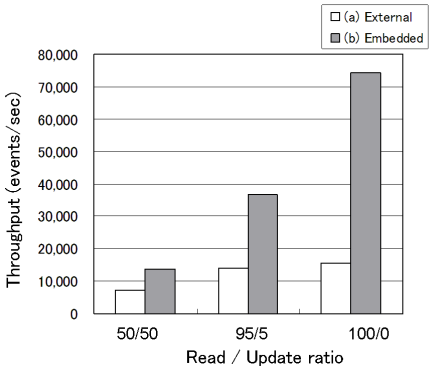

Fig. 7. Throughput

Fig. 8 shows latency results for read accesses. The embedded store reduced the average latency by 87%. The 95-percentile embedded latency was also much smaller than the external latency. This indicates that eliminating the TCP/IP communication is highly effective for I2A.

	External	Embedded
Average	640 μs	48 μs
Minimum	302 μs	32 μs
Maximum	66,639 μs	40,731 μs
95-percentile	623 μs	66 μs

Fig. 8. Read latency

Scalability. Next, we evaluated the scalability of our embedded store in a distributed environment. We measured the maximum throughput while changing the number of machines (nodes) for each workload. As described in Section 4, each event agent accesses only a local WXS shard in the same node, so each benchmark client runs in its own WXS node. Fig. 9 shows the results. In all of the workloads, the throughputs scaled well as the number of nodes increased. The throughputs with four nodes with read/update ratios of 50/50, 95/5, and 100/0 are respectively 3.9, 3.6, and 4.1 times higher than that with one node. This demonstrates the good scalability property of our proposed distributed JSON store for I2A. Since we do not need to fall back on WXS for routing to a data shard in our proposed store, it made a contribution for scalability.

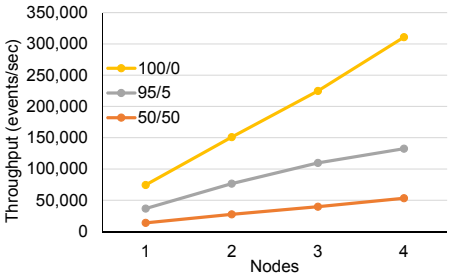

Fig. 9. Scale-out results for distributed store

6 Related Work

Stream processing systems, such as StreamBase [20], or InfoSphere Streams [12], support high-speed aggregation, enrichment, filtering, and transformation of streams of events. They analyze data in motion, as we do, but in addition, we can also analyze data at rest in a distributed store. Complex event processing (CEP) uses patterns over sequences of simple events to detect complex events. Distributed CEP engines include Cayuga [5], and CEP engines can also be distributed by embedding them in general streaming systems [11].

Several projects from the database area are closely related to our work. Ceri and Widom use production rules in a distributed database [6]. JAM uses Java agents to run batch analytics over a distributed database [19]. Kantere et al. discuss using triggers in databases that are not just distributed, but even (unlike our work) federated [13]. None of these works consider JSON support, or directly address the issue of performing analytics of data in motion alongside batch analytics of data at rest.

I2A is based on production rule languages, similar to [10,21], but allows integration with a distributed store. A related area is business process management systems (BPMSs), which use an event-driven architecture to coordinate human activities along with automated tasks. In contrast to distributed BPMSs [4], we also address the integration with a store and with batch analytics. From a database perspective, Datalog is the most commonly used rules language. It is storage centric and can be distributed [2]. To the best of our knowledge no work around Datalog addresses the integration of events with batch analytics, or includes support for JSON. A notable exception is JSON rules [16]; however, that work embeds rules in a browser rather than a distributed store.

The Percolator [17] runs distributed observers (similar to database triggers) over BigTable [7] (a distributed store). However, it does not directly address low-latency event processing; furthermore, it does not use JSON. Distributed stores that support JSON include MongoDB [14] and CouchDB [3]. Our work goes one step further by also offering rule-based event processing. Finally, our work is orthogonal to the question of query languages for JSON (e.g., JSONiq[9]) which could be easily integrated in our approach.

7 Conclusion

The *Insight to Action* (I2A) system embeds event processing into a distributed in-memory store. This enables event processing on large amounts of data without paying the penalty of going to disk or multiplexing all computation on a single machine. This paper is about the store component of I2A, specifically, about using JSON for this store. The advantages of JSON for the I2A store are that it helps simplify the system by using the same data model in all layers, and that it increases flexibility when schemas change. The challenge was how to reuse familiar APIs without losing the scalability advantages. We present our architecture for solving these challenges, along with performance results. Overall, I2A with a JSON store enables simple, flexible, and scalable stateful event processing. We are still actively developing I2A, and investigating several improvements, notably efficient execution strategies for aggregations, and how to improve freshness for the analytics without interfering with transaction performance.

References

1. Abadi, D.J., Ahmad, Y., et al.: The design of the Borealis stream processing engine. In: Conference on Innovative Data Systems Research (CIDR), pp. 277–289 (2005)
2. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.: BOOM analytics: Exploring data-centric, declarative programming for the cloud. In: European Conference on Computer Systems (EuroSys), pp. 223–236 (2010)
3. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The definitive guide. O'Reilly (2010)
4. Bonner, A.J.: Workflow, transactions and Datalog. In: Symposium on Principles of Database Systems (PODS), pp. 294–305 (1999)
5. Brenna, L., Gehrke, J., Johansen, D., Hong, M.: Distributed event stream processing with non-deterministic finite automata. In: Conference on Distributed Event-Based Systems (DEBS) (2009)
6. Ceri, S., Widom, J.: Production rules in parallel and distributed database environments. In: Conference on Very Large Data Bases (VLDB), pp. 339–351 (1992)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: Operating Systems Design and Implementation (OSDI), pp. 205–218 (2006)
8. Cooper, B.F., Silberstein, A., et al.: Benchmarking cloud serving systems with YCSB. In: Symposium on Cloud Computing (SoCC), pp. 143–154 (2010)
9. Florescu, D., Fourny, G.: JSONiq: The history of a query language. IEEE Internet Computing 17(5), 86–90 (2013)
10. Forgy, C.L.: OPS5 user's manual. Technical Report 2397, Carnegie Mellon University (CMU) (1981)
11. Hirzel, M.: Partition and compose: Parallel complex event processing. In: Conference on Distributed Event-Based Systems (DEBS), pp. 191–200 (2012)
12. Hirzel, M., Andrade, H., et al.: IBM Streams Processing Language: Analyzing big data in motion. IBM Journal of Research and Development (IBMRD) 57(3/4), 7:1–7:11 (2013)
13. Kantere, V., Kiringa, I., Zhou, Q., Mylopoulos, J., McArthur, G.: Distributed triggers for peer data management. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 17–35. Springer, Heidelberg (2006)
14. MongoDB NoSQL database, `http://www.mongodb.org/` (retrieved December 2013)
15. Node.js v0.10.24 manual & documentation (2013), `http://nodejs.org/`
16. Pascalau, E., Giurca, A.: JSON rules: The JavaScript rule engine. In: Knowledge Engineering and Software Engineering (KESE) (2008)
17. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Operating Systems Design and Implementation (OSDI), pp. 251–264 (2010)
18. Rivera, J., van der Meulen, R.: Gartner says in-memory computing is racing towards mainstream adoption. Press Release (April 2013), `http://www.gartner.com/newsroom/id/2405315`
19. Stolfo, S.J., Prodromidis, A.L., Tselepis, S., Lee, W., Fan, D.W., Chan, P.K.: JAM: Java agents for meta-learning over distributed databases. In: Conference on Knowledge Discovery and Data Mining (KDD), pp. 74–81 (1997)
20. Streambase, `http://www.streambase.com/` (retrieved December 2013)
21. WODM: IBM Operational Decision Manager, `http://www-03.ibm.com/software/products/en/odm/` (retrieved December 2013)
22. WXS: IBM WebSphere eXtreme Scale (2013), `http://www.ibm.com/software/products/en/websphere-extreme-scale/` (retrieved November)