
TYPHON
POLYGLOT AND HYBRID PERSISTENCE ARCHITECTURES FOR BIG DATA ANALYTICS

**Project Number 780251**

# D5.2 Data Event Organisation and Representation Report

**Version 1.0**
**18 December 2018**
**Final**

**Public Distribution**

## University of York

**Project Partners:** **Alpha Bank, ATB, Centrum Wiskunde & Informatica, CLMS, Edge Hill University, GMV, Nea Odos, The Open Group, University of L'Aquila, University of Namur, University of York, Volkswagen**

# PROJECT PARTNER CONTACT INFORMATION


| | |
|---|---|
| **Alpha Bank**<br>Vasilis Kapordelis<br>40 Stadiou Street<br>102 52 Athens<br>Greece<br>Tel: +30 210 517 5974<br>E-mail: vasileios.kapordelis@alpha.gr | **ATB**<br>Sebastian Scholze<br>Wiener Strasse 1<br>28359 Bremen<br>Germany<br>Tel: +49 421 22092 0<br>E-mail: scholze@atb-bremen.de |
| **Centrum Wiskunde & Informatica**<br>Tijs van der Storm<br>Science Park 123<br>1098 XG Amsterdam<br>Netherlands<br>Tel: +31 20 592 9333<br>E-mail: storm@cwi.nl | **CLMS**<br>Antonis Mygiakis<br>Mavrommataion 39<br>104 34 Athens<br>Greece<br>Tel: +30 210 619 9058<br>E-mail: a.mygiakis@clmsuk.com |
| **Edge Hill University**<br>Yannis Korkontzelos<br>St Helens Road<br>Ormskirk L39 4QP<br>United Kingdom<br>Tel: +44 1695 654393<br>E-mail: yannis.korkontzelos@edgehill.ac.uk | **GMV Aerospace and Defence**<br>Almudena Sánchez González<br>Calle Isaac Newton 11<br>28760 Tres Cantos<br>Spain<br>Tel: +34 91 807 2100<br>E-mail: asanchez@gmv.com |
| **Nea Odos**<br>Charalampos Daskalakis<br>Themistocleous 87<br>106 83 Athens<br>Greece<br>Tel: +30 210 344 7300<br>E-mail: cdaskalakis@neaodos.gr | **The Open Group**<br>Scott Hansen<br>Rond Point Schuman 6, 5th Floor<br>1040 Brussels<br>Belgium<br>Tel: +32 2 675 1136<br>E-mail: s.hansen@opengroup.org |
| **University of L'Aquila**<br>Davide Di Ruscio<br>Piazza Vincenzo Rivera 1<br>67100 L'Aquila<br>Italy<br>Tel: +39 0862 433735<br>E-mail: davide.diruscio@univaq.it | **University of Namur**<br>Anthony Cleve<br>Rue de Bruxelles 61<br>5000 Namur<br>Belgium<br>Tel: +32 8 172 4963<br>E-mail: anthony.cleve@unamur.be |
| **University of York**<br>Dimitris Kolovos<br>Deramore Lane<br>York YO10 5GH<br>United Kingdom<br>Tel: +44 1904 325167<br>E-mail: dimitris.kolovos@york.ac.uk | **Volkswagen**<br>Behrang Monajemi<br>Berliner Ring 2<br>38440 Wolfsburg<br>Germany<br>Tel: +49 5361 9-994313<br>E-mail: behrang.monajemi@volkswagen.de |

# DOCUMENT CONTROL

| Version | Status | Date |
|---|---|---|
| 0.1 | First draft | 02/10/2018 |
| 0.2 | Revised draft | 29/10/2018 |
| 0.3 | Event structure metamodel | 12/11/2018 |
| 0.4 | Code generation and analytics architecture discussion | 20/11/2018 |
| 0.5 | Changes based on feedback during the project meeting | 05/12/2018 |
| 1.0 | Final version after feedback from internal reviews | 18/12/2018 |

# TABLE OF CONTENTS



# TABLE OF FIGURES

# EXECUTIVE SUMMARY

This document extends the work described in deliverable D5.1 (Data Event Domain Analysis Report) by investigating in more depth the structure of messages for events triggered by stores and delivered through TyphonQL to be used for TYPHON analytics. The document also provides an overview of the code generator used to actualize the TyphonML and TyphonDL models into concrete implementation of the events to be used by analytics. Finally, a discussion of the proposed analytics architecture is presented.

# 1. INTRODUCTION

## 1.1 OVERVIEW

Polystore-generated events are the main enabler for data analytics to be applied concisely. The Data Event Organisation and Representation report (this document) investigates the possible events required for analytics across the different TYPHON stores and their structure. This report extends upon the Data Event Domain Analysis report (Task 5.1).
We highlight current widely adopted relational and non-relational stores and we discuss available implementation of instrumentation and tracing capabilities available for each of them. Emphasis is given on the definition of a common data event metamodel, which will define the structure of the generated events. In addition, a code generation approach that is based on Model-driven Engineering (MDE) principles is presented, while its implementation is discussed. The code generated by this automated approach will facilitate both the construction of the events triggered by the polystores and the development of analytics by analytics experts. Finally, a presentation of the proposed analytics architecture is also discussed in this document. An interim version of this architecture will be presented in the next deliverable (D5.3 – M18).

## 1.2 DOCUMENT STRUCTURE

The rest of document is organized as follows: Section 2 will address instrumentation and tracing in databases as a general consideration for TYPHON. A discussion will be further extended to performance considerations in Section 2. Section 3 will highlight instrumentation used in existing storage systems. In Section 0, we present the event publishing and monitoring architecture. In Section 5, we discuss the different events that the TYPHON analytics platform expects and their structure. In Section 6, the code generation approach is presented, while Section 7 concludes the work presented in this document and outlines the next steps.

# 2. STORAGE INSTRUMENTATION

Instrumentation is a general term used to cover different aspects of monitoring system behaviour. This can include code level tracing and debugging using logging messages to higher level profiling to observe the system response at runtime, e.g. monitoring memory leaks. We will use the term instrumentation unless otherwise required by the context of the discussion. For storage systems, there is even more need for instrumentation to fulfil several constraints in terms of data consistency, security and analytics. Instrumentation can help identify the overall database performance as the amount of data grows. Also, it can provide a mechanism by which system administrators can detect actions applied on the database including ones that breach security constraints.

TYPHON is interested in instrumenting different aspects of a polystore, which will help tackling:

- Security: instrumenting actions applied on a polystore will help identifying anomalous usage behaviour, which can help in halting malicious access to the stored data.

- Consistency: data consistency across different polystores can also be maintained through instrumentation, by insuring that events generated from a single storage node in a polystore are consistent with the rest of the system, e.g. generating a pre-delete event in a relational database to ensure that there are no data dependencies in other document or no-document databases present in the polystore.
- Profiling: generated events can help in monitoring the polystore usage and identify performance bottlenecks. Profiling the generated events can help also in decisions related to the evolution of the hardware/software requirements to sustain the sheer amount of data.
- Analytics: tracing the generated events can allow for custom detailed low-level analytics of raw data inserted by the user, which can provide business insights based on the nature of the data being processed.

# 3. INDUSTRY-WIDE INSTRUMENTATION

## 3.1 OVERVIEW

In the previous deliverable (D5.1), we presented an in-depth analysis of the different kinds of events (e.g. pre-access, post-access, pre-update, post-update) that a TYPHON polystore could publish in order to facilitate the development of orthogonal analytics and monitoring services. In that analysis, we assessed six databases to identify notifications that are triggered when data operations are executed. More specifically, we categorised the databases into two distinct categories, i.e., relational and non-relational databases (SQL and NoSQL). We then studied the most widely used databases for each category based on a public database ranking index [1]. In particular, for relational databases we assessed MySQL [1] and Oracle DBMS [2]. As different types of non-relational databases exist, we assessed one for each the following four types: document based (i.e., MongoDB [3]), key-value (i.e., Redis [4]), graph based (i.e., Noe4J [5]) and column wide stores (i.e., Cassandra [6]). We then presented examples, for each of the four industrial use-cases of the project. Through these examples we demonstrated how the events that databases produce could be useful in developing analytics, extracting information and responding accordingly.

In this section of the current report, we investigate how different databases publish instrumentation events. The approach in this section is not targeted to provide a thorough investigation of all databases considered in the Data Event Domain Analysis report (D5.1), but rather an overview of fields available in the most widely used of them. These fields could be directly incorporated in the events structure (discussed in the next section) or a custom implementation through TyphonQL will be required to provide them using database specific trigger events, as elaborated in the Data Event Domain Analysis report (D5.1). The custom implementation will provide any of the fields required in the event message, if the underlying store of a polystore does not provide direct access similar to the ones discussed in this section.

## 3.2 ORACLE

There is a range of methods to instrument actions applied through an Oracle database. Auditing actions on an Oracle store can be enabled in different modes depending on the value assigned to an audit flag named AUDIT_TRAIL. For example, "AUDIT_TRAIL = db, extended" will allow storing the SQL query executed by the user against the database. Several audit fields are also accessible through the SYS.AUD$ table or the DBA_COMMON_AUDIT_TRAIL view. Other methods provided by Oracle involve the use of analytics packages, such as DBMS_APPLICATION_INFO, available for the Oracle Enterprise. Table 1 lists few of the fields that get generated with audit events. The rest of the fields with a description can be found in [7].

**Table 1: Audit event for Oracle**

| Column | Example | Comments |
|---|---|---|
| DBA_AUDIT_TRAIL.OS_USER NAME | Admin | The username of the system |
| DBA_AUDIT_TRAIL.USERNAM E | HR | The database user |
| DBA_AUDIT_TRAIL.ACTION_ NAME | CREATE TABLE, ALTER TABLE, UPDATE, DELETE, SELECT, INSERT | Actions applied to the database |
| DBA_AUDIT_TRAIL.TIMESTA MP | 27-JUL-18 | The day of the action (this is not granular enough to show the second the action was executed in) |
| DBA_AUDIT_TRAIL.EXTENDE D_TIMESTAMP | 27-JUL-18 12.41.14.263000000 EUROPE/LONDON | Timestamp of the creation of the audit trail entry (timestamp of user log-in for entries created by AUDIT SESSION) in UTC (Coordinated Universal Time) time zone |
| DBA_AUDIT_TRAIL.SESSIONI D | 71277 | The session id for accessing the database |
| DBA_AUDIT_TRAIL.ENTRYID | 1108 | The id of the audit trail entry in a specific session id |
| DBA_AUDIT_TRAIL.STATEME NTID | 41 | An id of the statement executed in a session |
| DBA_AUDIT_TRAIL.RETURNC ODE | ● 0 - Action succeeded<br>● 2004 - Security violation | This field can be used to return the status of the action |
| DBA_AUDIT_TRAIL.SQL_TEXT | Select * from emp; | The actual statement |

## 3.3 MYSQL

MySQL provides similar fields to Oracle. These fields are accessible in the commercial distribution. The instrumentation details are populated through an audit log with the help of an audit plugin. An example, taken from [8], is shown in

```
<?xml version="1.0" encoding="utf-8"?>
<AUDIT>
  <AUDIT_RECORD
    TIMESTAMP="2017-10-16T14:25:00 UTC"
    RECORD_ID="1_2017-10-16T14:25:00"
    NAME="Audit"
    SERVER_ID="1"
    VERSION="1"
    STARTUP_OPTIONS="--port=3306"
    OS_VERSION="i686-Linux"
    VERSION="5.7.21-log"/>

<AUDIT>
```

Listing 1 that follows.

**Listing 1: An example of an audit record in MySQL**

Table 2 includes the fields populated in an audit message of MySQL through the audit plugin.

**Table 2: Audit message fields for MySQL**

| XML field | Example | Comments |
|---|---|---|
| NAME | The field can take one of the following values:<br>● Audit: when auditing starts, which may be server startup time<br>● Connect: when a client connects (log-ging-in).<br>● Query: a SQL statement (executed directly)<br>● Prepare: preparation of an SQL state-ment; usually followed by Execute<br>● Execute: Execution of an SQL state-ment; usually follows Prepare.<br>● Shutdown: Server shutdown.<br>● Quit: when a client disconnects.<br>● NoAudit: auditing has been turned off. | |
| RECORD_ID | “12_2017-10-16T14:25:00” | The sequence number at the begin-ning is initialized to the size of the audit log file, then incremented by 1 for each record logged |
| TIMESTAMP | "2017-10-16T14:25:32 UTC" | |
| COMMAND_CLAS | "drop_table" | A string that indicates the type of |

| | | |
|---|---|---|
| S | | action performed |
| CONNECTION_ID | 127 | An unsigned integer representing the client connection identifier of the session |
| CONNECTION_TYPE | "SSL/TLS" | |
| DB | “test” | A string representing the default database name |
| HOST | “localhost” | A string representing the client host name |
| IP | 127.0.0.1 | A string representing the client IP address |
| MYSQL_VERSION | "5.7.21-log" | A string representing the MySQL server version |
| OS_LOGIN | OS_LOGIN="jeffrey" | A string representing the external user name used during the authentication process |
| OS_VERSION | "x86_64-Linux" | A string representing the operating system on which the server was built or is running |
| SERVER_ID | SERVER_ID="1" | An unsigned integer representing the server ID. This is the same as the value of the server_id system variable |
| SQLTEXT | SQLTEXT="DELETE FROM t1" | A string representing the text of an SQL statement<br>Long values may be truncated. |
| STATUS | STATUS="1051" | A list of error messages are available here:<br>https://dev.mysql.com/doc/refman/8.0/en/error-messages-server.html |
| STATUS_CODE | 0 or 1 | STATUS_CODE is 0 for success and 1 for error |

## 3.4 MONGODB

The auditing facility can write audit events to the console, the syslog, a JSON file, or a BSON file [9]. Mongo provides several options for monitoring the database status and logging events. For example, *mongostat* [10] is a utility that can capture the counts of CRUD actions executed against the database. Another utility, *mongotop* [10] is used to check the read/write activity within the database. System-wide event monitoring can be enabled as well to provide a detailed audit of every action being executed against the database. The generated logging can be in JSON or the MongoDB-specific format BSON. Listing 2 shows a snippet of the audited events on a mongo database, while Table 3 provides a description for each field.

{ "atype" : "createDatabase", "ts" : { "$date" : "2018-10-09T15:36:22.986+0100" }, "local" : { "ip" : "127.0.0.1", "port" : 27017 }, "remote" : { "ip" : "127.0.0.1", "port" : 58139 }, "users" : [{"user":"antun","db":"admin"}], "roles" : [{"role":"read","db":"admin"}], "param" : { "ns" : "test" }, "result" : 0 }

{ "atype" : "createCollection", "ts" : { "$date" : "2018-10-09T15:36:22.986+0100" }, "local" : { "ip" : "127.0.0.1", "port" : 27017 }, "remote" : { "ip" : "127.0.0.1", "port" : 58139 }, "users" : [], "roles" : [], "param" : { "ns" : "test.inventory" }, "result" : 0 }

**Listing 2: Auditing events in MongoDB**

**Table 3: MongoDB auditing fields description**

| JSON Field | Values assigned | Comments |
|---|---|---|
| atype | addShard, applicationMessage, atype, authCheck, authenticate, createCollection, createDatabase, createIndex, createRole, createUser, dropAllRolesFromDatabase, dropAllUsersFromDatabase, dropCollection, dropDatabase, dropIndex, dropRole, dropUser, enableSharding, grantPrivilegesToRole, grantRolesToRole, grantRolesToUser, removeShard, renameCollection, revokePrivilegesFromRole, revokeRolesFromRole, revokeRolesFromUser, shardCollection, shutdown, updateRole, updateUser | Action type can take any value of possible actions that can get executed on MongoDB.<br>Each event comes with its own error code Appended to the "result" field,<br>e.g. "authenticate" actiontype will return either:<br>0 - Success<br>18 - Authentication Failed |
| ts | "2018-10-09T15:36:22.986+0100" | Timestamp of the event<br>UTC time of the event, in ISO 8601 |
| local | { "ip" : "127.0.0.1", "port" : 27017 } | Document that contains the local IP address and the port number of the running instance. |
| remote | { "ip" : "127.0.0.1", "port" : 58139 } | Document that contains the remote IP address and the port number of the incoming connection associated with the event. |
| users | [{"user":"antun","db":"admin"}] | Array of user identification documents. Because MongoDB allows a session to log in with different user per database, this array can have more than one user. |
| roles | [{"role":"read","db":"admin"}] | Array of documents that specify<br>the roles granted to the user. |
| param | e.g. for "authenticate" param holds : {<br>user: <user name>,<br>db: <database>,<br>mechanism: <mechanism><br>} | Specific details for the event. |
| result | e.g. "authenticate" will cause a result of:<br>0 - Success<br>18 - Authentication Failed | Holds the success/error code of an action |

## 3.5 NEO4J

Neo4J [5] is a leading graph database. The enterprise version of Neo4J provides security and query logging. Enabling query logging allows monitoring all query events that are executed against the database. A different logging facility allows monitoring security events such as creating/deleting a database user, permission and access rights modifications, password updates and similar events. Listing 3 is an example of a query logging [11] while Listing 4 is an example for the security logging [12].

```
2017-11-22 14:31 ... INFO  9 ms: bolt-session   bolt    johndoe neo4j-javascript/1.4.1      client/127.0.0.1:59167  ...
2017-11-22 14:31 ... INFO  0 ms: bolt-session   bolt    johndoe neo4j-javascript/1.4.1      client/127.0.0.1:59167  ...
2017-11-22 14:32 ... INFO  3 ms: server-session http    127.0.0.1   /db/data/cypher neo4j - CALL dbms.procedures() - {}
2017-11-22 14:32 ... INFO  1 ms: server-session http    127.0.0.1   /db/data/cypher neo4j - CALL dbms.showCurrentUs...
2017-11-22 14:32 ... INFO  0 ms: bolt-session   bolt    johndoe neo4j-javascript/1.4.1      client/127.0.0.1:59167  ...
2017-11-22 14:32 ... INFO  0 ms: bolt-session   bolt    johndoe neo4j-javascript/1.4.1      client/127.0.0.1:59167  ...
2017-11-22 14:32 ... INFO  2 ms: bolt-session   bolt    johndoe neo4j-javascript/1.4.1      client/127.0.0.1:59261  ...
```

**Listing 3: Neo4J query logging**

```
2016-10-27 13:45:00.796+0000 INFO     [AsyncLog @ 2016-10-27 ...]  [johnsmith]: logged in
2016-10-27 13:47:53.443+0000 ERROR [AsyncLog @ 2016-10-27 ...]  [johndoe]: failed to log in: invalid principal or
credentials
2016-10-27 13:48:28.566+0000 INFO     [AsyncLog @ 2016-10-27 ...]  [johnsmith]: created user `janedoe`
2016-10-27 13:48:32.753+0000 ERROR [AsyncLog @ 2016-10-27 ...]  [johnsmith]: tried to create user `janedoe`: The
specified user ...
2016-10-27 13:49:11.880+0000 INFO     [AsyncLog @ 2016-10-27 ...]  [johnsmith]: added role `admin` to user `janedoe`
2016-10-27 13:49:34.979+0000 INFO     [AsyncLog @ 2016-10-27 ...]  [johnsmith]: deleted user `janedoe`
2016-10-27 13:49:37.053+0000 ERROR [AsyncLog @ 2016-10-27 ...]  [johnsmith]: tried to delete user `janedoe`: User
'janedoe' does ...
2016-10-27 14:00:02.050+0000 INFO     [AsyncLog @ 2016-10-27 ...]  [johnsmith]: created role `operator`
```

**Listing 4 Neo4J security logging**

**Table 4: Neo4J auditing fields description**

| Field | Values assigned | Comments |
|---|---|---|
| Network Protocol | bolt, http | Neo4j provides several drivers for different languages implementing the Bolt protocol [13] for fast database connections.<br><br>Other network protocols are supported, such as http. |
| User | "johndoe" | The name of the database user |
| Driver type/version | neo4j-javascript/1.4.1 | Neo4j provides javascript, python and java drivers. It shows here the type of the driver and its version |
| Query logging | created user, deleted user, tried to delete user | This type of audit shows all possible actions executed against the database in system logging statements with the execution timestamp preceding each statement. |

## 3.6 REDIS

REDIS [4] is an in-memory NoSql data store that depends on key value pairs. Also, it has several storage persistence levels. Redis provides a built-in audit and logging functionality that allows monitoring actions applied on the Redis store at different levels of granularity. Listing 5 shows a snippet of the logging available through Redis GUI, which is also accessible through a REST API. Redis provides two other logging facilities, Slowlog [14] and Rsyslog [15] for low level CPU, memory usage and node assignments in clustered environments.

| Time | Originator | Source | Type | Description |
|---|---|---|---|---|
| 5/29/2018 8:36:02 AM | system | User | Notification | Login succeeded (username: Administrator) |
| 5/29/2018 8:24:55 AM | system | User | Notification | Login succeeded (username: Administrator) |
| 5/29/2018 8:24:52 AM | system | User | Notification | Login failed (username: Administrator) |
| 5/29/2018 8:24:50 AM | system | User | Notification | Login failed (username: Administrator) |
| 5/29/2018 8:24:46 AM | system | User | Notification | Login failed (username: Administrator) |
| 5/29/2018 8:24:37 AM | system | User | Notification | Login failed (username: Administrator) |
| 5/29/2018 7:41:49 AM | system | Cluster | Notification | Database activated.<br>Database: TT2 |
| 5/29/2018 7:41:49 AM | Administrator [test@redislabs.com] | Bdb | Notification | Database creation requested.<br>Database: TT2 |

**Listing 5: Redis events logging**

**Table 5: Redis auditing fields description**

| Field | Values assigned | Comments |
|---|---|---|
| Time | "5/29/2018 8:36:02 AM" | Event timestamp |
| User | "johndoe" | The name of the database user |
| Originator | System/ Administrator | The logged in user of the database |
| Source | e.g. User, Cluster | The Role of the logged in user |
| Type | "Notification" | Message type of the generated event |
| Description | e.g. Login failed | The type of the action in addition to its success/failure status. |

# 4. DATA ANALYTICS EVENTS SYSTEM ARCHITECTURE

Extending from the events generated from different industry-leading stores that could be integrated into a polystore, this section presents a high-level architecture for polystore events analytics in TYPHON, illustrated in Figure 1.

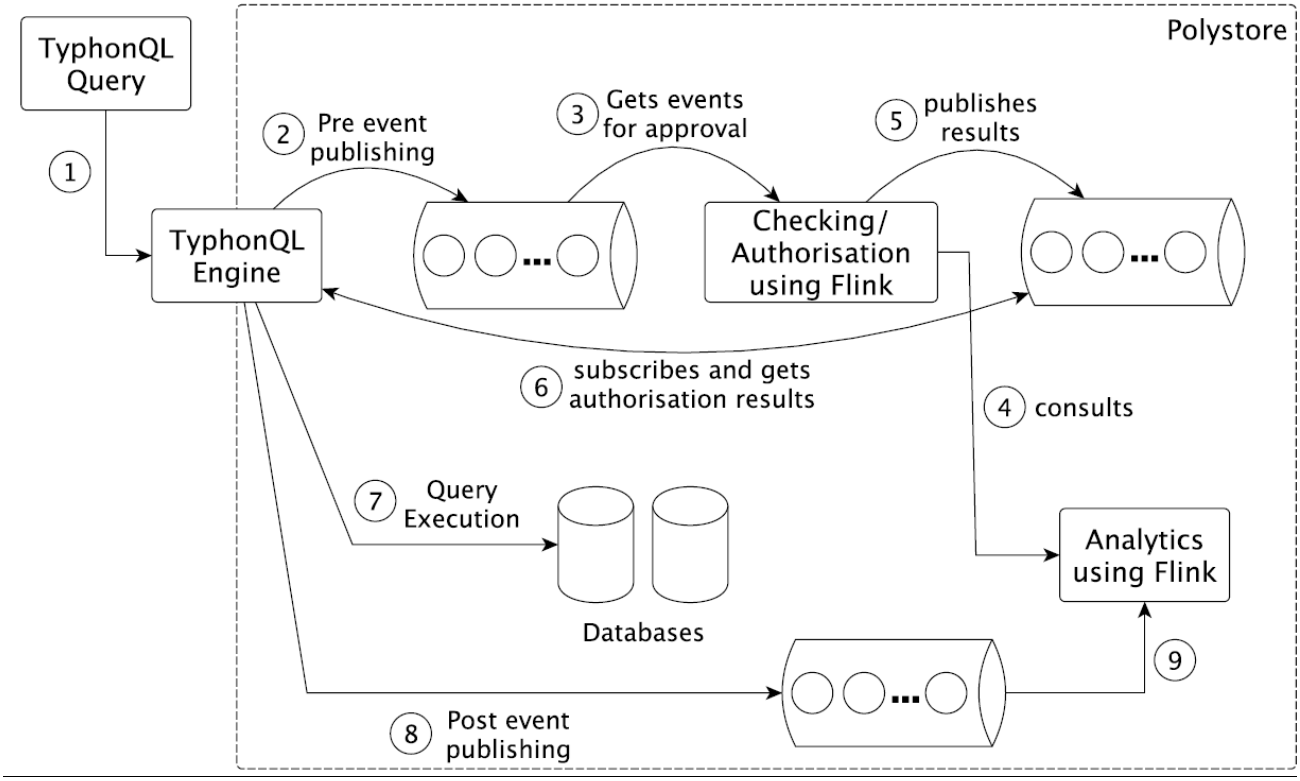


**Figure 1: Analytics infrastructure architecture**

The main subsystems involved in the proposed architecture are as follows:

- Queues: these operate as the intermediary communication channel between different subsystems to preserve the messages of events generated at each stage of the event flow to be consumed by the relevant subscriber. A preliminary implementation of the architecture is considering Apache Kafka [16] for message queuing.
- Stream processors: such processors will be responsible for applying direct analysis to events by providing a platform for the implementation of the necessary analytics. A preliminary implementation of the architecture is considering Apache Flink [17] for events analytics.

### 4.1.1 Events Flow

Following Figure 1, events undergo nine stages that are distributed among two main interleaved phases; Authorization and Analytics. The authorization phase involves validating if a new incoming event (request) will be allowed to be executed against the polystore, based on either hardcoded rules defined in the analytics suite or the history of previous events processed through the stream processor. This ensures that any malicious or unintended activity that does not follow the polystore's business rules will be detected and rejected. The second phase is the analytics, which involves the continuous consumption of event messages for analysis and for inquiries incoming from the authorization phase. Accordingly, the operation of the two phases is interleaved. Below are the stages an event will go through as it (the event) progresses within the proposed architecture.

1. In this stage, a query is passed by a user to the TyphonQL engine for parsing and execution.
2. TyphonQL will have to check the possibility of executing the parsed query. Accordingly, it will publish a pre-execution event, push it to an authorization queue and wait for the authorization decision of this event.
3. A stream processor dedicated for authorization, will consume messages from the authorization queue to apply the required validation checks before generating an authorization decision of an event.
4. In addition to any rules/checks the authorization stream processor has to apply, it can also consult the analytics stream processor, if there are any linked events that could indicate a malicious or an abnormal activity in relation to the new incoming event.
5. Following the application of the required validations/checks and the consultation with the analytics stream processor, the authorization stream processor publishes its authorization decision to the decision queue.
6. TyphonQL receives the authorization decision it was waiting for in Stage 2.
7. Based on the outcome of the authorization decision generated in Stage 6, TyphonQL will execute the query received at Stage 1 or reject it.
8. In case the query is executed by TyphonQL, a post-execution event will be generated and pushed to the analytics queue.
9. The analytics stream processor consumes the messages to which the relevant analytics could be applied. Examples of analytics are presented in Section 5.2.

The structure of the events that will be published in the queues (both pre and post execution) are described below.

# 5. DATA ANALYTICS EVENTS STRUCTURE

In this section we present the data event organisation metamodel. This will be used by TyphonQL to serialise events that take place in a polystore and transmit them through events queues. Data analytics platforms should then be able to monitor these queues and using the same metamodel, deserialize them in objects for manipulation. To facilitate this process, a tailored to the polystore Java code implementation will be generated retrieving information stored in the TyphonML models used to create the polystore. The code generation process will be discussed in Section 0.

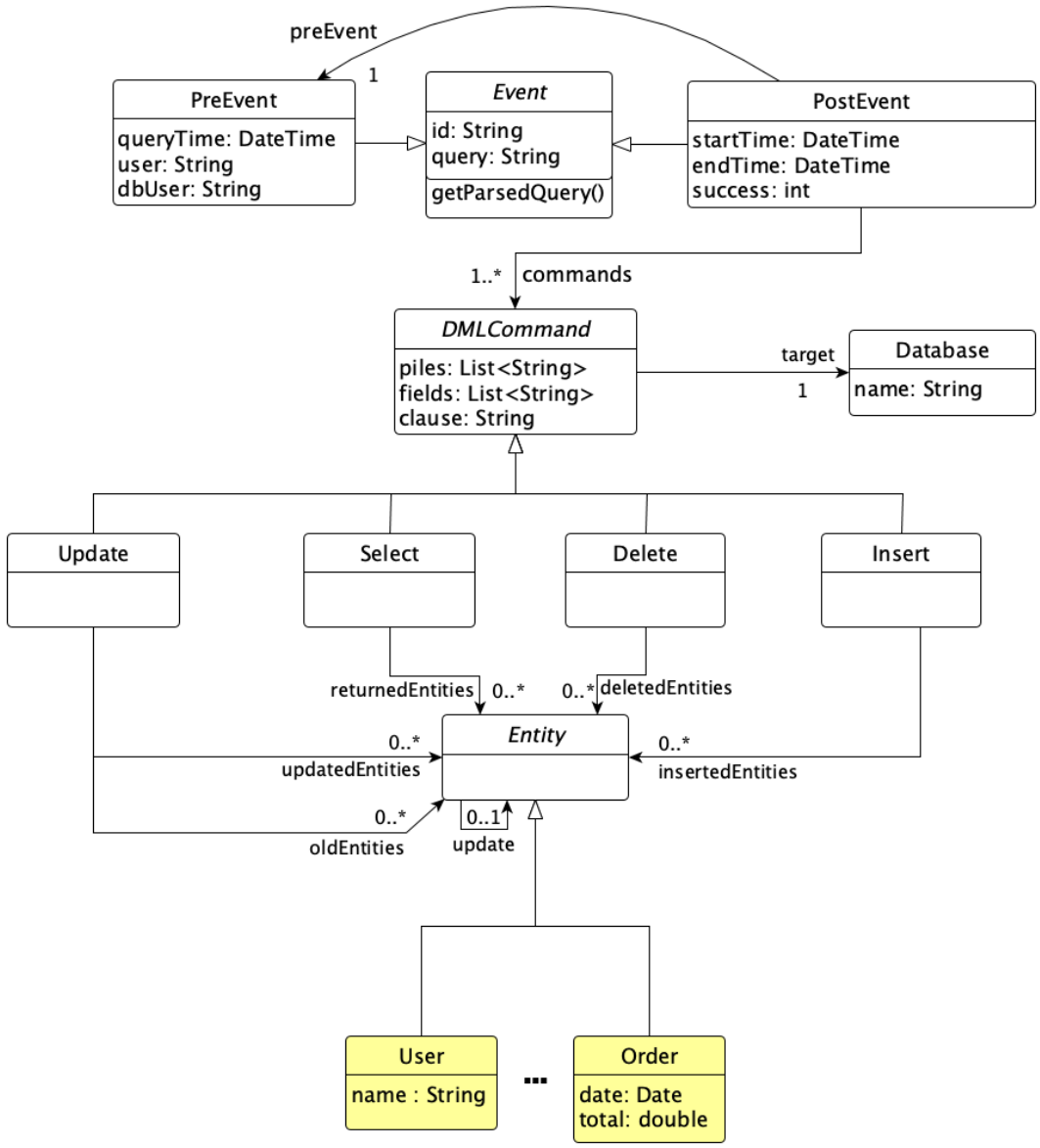


**Figure 2: Data event organization metamodel**

## 5.1 METAMODEL

Figure 2 presents the data event organisation metamodel. Each time a command arrives at the TyphonQL execution engine, the latter should generate one event before the execution of the command. This, labelled as *PreEvent* in the metamodel, will be used to block commands from execution based on conditions defined in the analytics engine. If the PreEvent gets approved, the command will be executed and a *PostEvent* will be generated by the TyphonQL engine. In any case, each event will be characterised by a unique *id* and will include the TyphonQL *query* that generated it. The abstract syntax tree (AST) of the query can be retrieved using the *getParsedQuery()* operation. PostEvent instances will point to their corresponding PreEvent instance (*preEvent* reference in the metamodel). In order to facilitate analytics related to performance and the evolution of the polystore (WP6) PreEvents will hold a timestamp of when the command arrived for execution (*queryTime*), while PostEvent when the execution started (*startTime*) and when it ended (*endTime*). In addition, PreEvents should store the *user* that requested the execution of the command along with the database user (*dbUser)* who executed it, while PostEvents will store a *success* code declaring if the execution was successful or not.

Each PostEvent will also point to the type of the data manipulation language commands (*DMLCommand*) of TyphonQL (e.g., Select, Delete, Insert and Update)[1] through the *commands* reference. As queries might consist of multiple DML commands, the multiplicity of the commands reference is set as 1 to many. Each DMLCommand will store a list of the "data structures" that were affected (e.g., the specific table in a relational database) in the field *piles.* In addition, some queries will only affect specific "fields" of the data structure (e.g., columns in a relational database). A list (i.e., *fields)* will hold this information. Finally, the *clause* property will hold the condition clause (if any) that was applied to this command.

In addition, each DMLCommand will point to the data entity that was manipulated (e.g., User, Order, etc.). The entities will be generated from information taken from the the TyphonML metamodel (highlighted in yellow in Figure 2). Depending on the type of the DMLCommand, the label of this reference changes (e.g., *InsertedEntities* for Insert DML commands). For commands of type Update, two references to the entities are needed. The first (i.e., *updatedEntities*) will hold a reference to the newly created entry while the *oldEntities* will hold a reference to the entity that has changed. In order to keep track of which entity is the updated version of an old entity, a reference called *update* needs to be used between them. This reference will start from the old version of the entity, pointing to the updated one.

Finally, each DMLCommand will also refer to the *target* Database in the polystore (e.g., relational, document, etc.) that this command was executed on. Database object will consist of the name of the database which we assume should be unique in a polystore. The type of the database (e.g., relational, document, etc.) will be received by querying the TyphonML model when analytics are developed. The type could be stored in the Database objects, instead. On the one hand, this would minimise the time needed to extract this information, but on the other hand it would increase the space needed in the

[1] TyphonQL syntax is not yet finalised but we expect it to support these DML commands as a minimum.

queue to host the event objects. This is a typical *space vs time* trade-off for which a decision can be taken based on specific analytics use cases each user of the polystore has. We discuss plans for future work on this in Section 7.

Table 6 summarises each type/field in the metamodel.

**Table 6: Data event message fields description**

| Tag Name | Description |
|---|---|
| Event | An event is published every time a command arrives at the TyphonQL engine |
| - id | A unique identifier for the generated event (Pre/Post) |
| - query | The TyphonQL query that triggered the generation of this event |
| - getParsedQuery() | An operation that returns back the AST of the TyphonQL query using the TyphonQL's parser |
| PreEvent | The PreEvent event is triggered before the execution of the command |
| - queryTime | The date/time the query arrived at TyphonQL engine |
| - user | The user in the application that generated the query |
| - dbUser | The database user (e.g., admin) who asked the execution of the query |
| PostEvent | The PostEvent event is triggered after the execution of the command |
| - startTime | The date/time TyphonQL started the execution of the query |
| - endTime | The date/time TyphonQL finished the execution of the query |
| - success | Code showing the success/failure of an execution of a command |
| - preEvent | Reference to the PreEvent object for a specific PostEvent object |
| - commands | Reference to the DML commands that were executed as part of the query that generated the event |
| DMLCommand | Each time a DML command is executed one subtype of this type is instantiated |
| - piles | The data structures (e.g., tables) the command accessed |
| - fields | The items in the data structures (e.g., columns) the command accessed |
| - clause | The TyphonQL sub-query for this command |
| - target | A reference to the database the command accessed |
| Database | A database in the polystore that is accessed by a command |
| - name | The name of the database |
| Update | This type is instantiated if the DML command was an UPDATE |
| - updatedEntities | A reference to the entity object that includes the updated fields |
| - oldEntities | A reference to the entity object that includes the fields before the update |
| Select | This type is instantiated if the DML command was a SELECT |
| - returnedEntities | A reference to the entity objects that were returned by the execution of the select statement |
| Delete | This type is instantiated if the DML command was a DELETE |
| - deletedEntities | A reference to the entity objects that were deleted after the execution of the delete statement |
| Insert | This type is instantiated if the DML command was an INSERT |
| - insertedEntities | A reference to the entity objects that were inserted by the execution of the insert statement |
| Entity | The object the holds the fields/getters/setters for each entity described in the TyphonML model. This class is abstract and is extended by the specific entities in the TyphonML model using a model-to-text transformation |
| - update | A reference from the old entity to the updated entity object |

The analytics architecture will be also able to capture DDL command events (not only DML commands as shown in the metamodel). However, we do not expect these types of events to be generated frequently. In addition, although DDL events might be used for the extraction of information, useful to the database evolution mechanism, we do not see these being used in any analytics scenarios. For this reason, in this document we focus on DML commands only.

In the following section, we present motivational scenarios for the types of analytics that can be performed based on such events. The examples also explain and justify the structure of the data event metamodel and are based on a simplified e-shop application.

## 5.2 ANALYTICS SCENARIOS

The following analytics scenarios are based on the assumption that the polystore hosts the simplified[2] schema (see Figure 3) for storing the users who have an account in the e-shop, the products available and the orders placed. As TyphonQL is under active

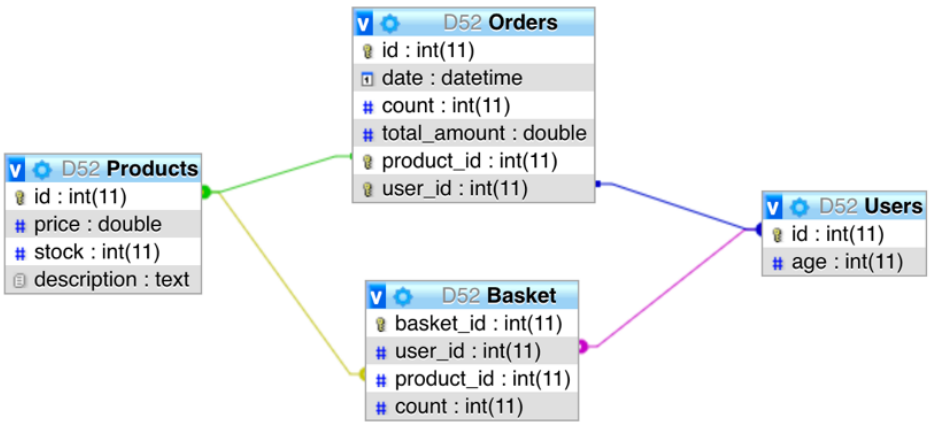


**Figure 3: Analytics scenarios database schema**

development at the time of writing this document, we have used a relational database and SQL instead of a polystore and TyphonQL for this concrete example, however, these scenarios are applicable to polystores/TyphonQL too.

| **Scenario 1** | |
|---|---|
| **Goal** | Find the top **viewed** but **not ordered** products in the last 24-hours to offer a discount on them. |
| **Description** | The analytics suite must be able to return back the ids of the top products which users looked for within a specific time window (e.g., 24 hours) but were not ordered. The top products in this category can be calculated as the ratio of the times a product was browsed divided by the times the product was purchased in this time window. |
| **Monitoring Queries**[3] | SELECT * FROM PRODUCTS WHERE id = 'id' |

[2] For reasons of brevity we omit details in the schema that are not used in the scenarios we present in this document.
[3] Using SQL-alike syntax as TyphonQL syntax is not yet finalised.

| | |
|---|---|
| | INSERT INTO ORDERS VALUES (‘id’, ‘date’, ‘count’, ‘total_amount’, ‘product_id’, ‘user_id’) |
| **Information from PreEvent** | • Pre-events are not relevant in this scenario |
| **Information from PostEvent** | • type of DMLCommand (i.e., Select) to count the times each product was browsed<br>• piles: the data item accessed (i.e., Products)<br>• returnedEntities: the Product object returned by each select command to extract information about it and offer the appropriate discount<br>• type of DMLCommand (i.e., Insert) to count the times each product was ordered<br>• piles: the data item accessed (i.e., Orders)<br>• query: to extract the product_id using the getParsedQuery() operation |

| **Scenario 2** | |
|---|---|
| **Goal** | Order more stock of prime items if users view them more than normally |
| **Description** | Identify the products that have higher views than normally to order more stock, before users place orders. We are only interested in those products labelled as prime, as next day delivery is guaranteed and we need to make sure that stock will last. |
| **Monitoring Queries** | SELECT * FROM PRODUCTS WHERE id = ‘id’ |
| **Information from PreEvent** | • Pre-events are not relevant in this scenario |
| **Information from PostEvent** | • type of DMLCommand (i.e., Select)<br>• piles: the data item accessed (i.e., Product)<br>• clause: get the id of the product and use TyphonQL to get if it is prime. We can also, as in scenario 1, get the Product object from the returnedEntities reference instead. We use this scenario to demonstrate the alternative solution. |

| **Scenario 3** | |
|---|---|
| **Goal** | Identify products that are running low in stock to order more. |
| **Description** | Find those products whose stock is getting below a specific threshold when orders are placed and order some more. The same action could be done by querying all the products from the database at a specific time (e.g., at midnight) and find those low in stock. However, with the latter approach we wouldn’t have real-time stock metrics and we also need to query all the products in the store. By monitoring the events, we just need to check those products for which an order was placed. In addition, this offers real-time stock monitoring. |
| **Monitoring Queries** | UPDATE PRODUCTS SET stock = stock - count WHERE id = ‘id’ |
| **Information from PreEvent** | • Pre-events are not relevant in this scenario |
| **Information from PostEvent** | • type of DMLCommand (i.e., Update)<br>• piles: the data item accessed (i.e., Products)<br>• updatedEntity: get the updated Product object which includes a field for the remaining |

| | |
|---|---|
| | stock. |

| **Scenario 4** | |
|---|---|
| **Goal** | Prevent ingenuine orders. |
| **Description** | Prevent user with multiple orders placed within a short time window (e.g. 5 minutes - number of orders above a specific threshold) placing another one. The analytics engine should provide the mechanism to block the execution of the order until the buyer confirms that it is genuine. |
| **Monitoring Queries** | INSERT INTO ORDERS VALUES (‘id’, ‘date’, ‘count’, ‘total_amount’, ‘product_id’, ‘user_id’) |
| **Information from PreEvent** | • queryTime: construct the time window<br>• clause: get the user_id to count the number of orders placed within a time-window<br>• query: get type of the command (i.e., Insert) using the getParsedQuery() operation<br>• query: the data item accessed (i.e., Orders) using the getParsedQuery() operation |
| **Information from PostEvent** | • No post event will be generated as we want to prevent the execution of the query |

| **Scenario 5** | |
|---|---|
| **Goal** | Monitor products removed from a basket |
| **Description** | A user can add a product to a basket, and later decide that she wants to remove it. This could indicate a problem with this specific product, especially if several users have removed it from their baskets, that the seller should be made aware of. |
| **Monitoring Queries** | DELETE FROM BASKET WHERE id = ‘id’ |
| **Information from PreEvent** | • Pre-events are not relevant in this scenario. |
| **Information from PostEvent** | • type of DMLCommand (i.e., Delete)<br>• piles: the data item accessed (i.e., Basket)<br>• clause: extract the id of the product which we can use to query the polystore to check the reason behind customers avoiding this product or<br>  ○ deletedEntities: get the Basket object from the deletedEntities reference |

| **Scenario 6** | |
|---|---|
| **Goal** | Monitor queried products before a purchase. |

| | |
|---|---|
| **Description** | A user searches for several items before making a final purchase. This can help in enhancing the collaborative filtering to enhance the recommendation for other users. |
| **Monitoring Queries** | SELECT * FROM PRODUCT WHERE id = 'id'<br><br>INSERT INTO ORDERS VALUES ('product_id', …) |
| **Information from PreEvent** | • user: get the user to group search queries for each individual<br>• queryTime: get the time the user searched for each product (to filter and check only queries happening within a specific time window, i.e., do not monitor searches that happened, for example, a week ago as they might be for a different "purchase session") |
| **Information from PostEvent** | • type of DMLCommand (i.e., Select)<br>• piles: the data item accessed (i.e., Product)<br>• type of DMLCommand (i.e., Insert)<br>• piles: the data item accessed (i.e., Orders)<br>• clause: extract the id of the product and use TyphonQL to extract details about the purchased product and the products searched before that |

| **Scenario 7** | |
|---|---|
| **Goal** | Predict a user's next search based on previous searches |
| **Description** | Some products are sold individually, or they are sold within a package. Analytics can recommend to a user, products that are related to her purchase and possibly provide her with the option of an offer to buy a package of two or more different products that are usually sold together (e.g. complement each other's functionality). Also, this information can be used to refine the queries and presentation of the results (e.g., list products that match the query and also have higher chances to match user's needs/preferences). |
| **Monitoring Queries** | SELECT * FROM PRODUCT WHERE id = 'id' |
| **Information from PreEvent** | • Pre-events are not relevant in this scenario. |
| **Information from PostEvent** | • type of DMLCommand (i.e., Select)<br>• piles: the data item accessed (i.e., Product)<br>• clause: extract the id of the products which we can use either to query the polystore for similar products to recommend to the user or use in a predictive/learning algorithm to identify user's needs/preferences. |

| **Scenario 8** | |
|---|---|
| **Goal** | Monitor query execution time. |
| **Description** | Use analytics to detect queries taking long to execute. This will help detect performance bottlenecks within the system. |
| **Monitoring Queries** | Any query |

| | |
|---|---|
| **Information from PreEvent** | • Pre-events are not relevant in this scenario |
| **Information from PostEvent** | • The unique event id and starting time<br>• The unique event id and ending time. |

| **Scenario 9** | |
|---|---|
| **Goal** | Identify if a customer is determined to buy a specific product. |
| **Description** | The analytics can provide access to the collection of queries performed by a user in browsing a specific product or similar ones (e.g. same product but different brands). This type of analysis can help in assessing how determined is the user to buy a specific product and give a hint to the e-commerce website owner that it worth investing on targeted advertisement for this user. |
| **Monitoring Queries** | SELECT * FROM PRODUCT WHERE id = 'id' |
| **Information from PreEvent** | • Pre-events are not relevant in this scenario. |
| **Information from PostEvent** | • type of DMLCommand (i.e., Select)<br>• piles: the data item accessed (i.e., Product)<br>• clause: extract the id of the product which we can use to query the polystore and find the similar ones |

| **Scenario 10** | |
|---|---|
| **Goal** | Tracking suspicious reviews (a) |
| **Description** | The events analytics should be able to track review submissions by the same user occurring within a short time window across several products or by different users across the same product. Such a behaviour will denote the possibility of fake reviews submitted by bots. |
| **Monitoring Queries** | INSERT INTO REVIEWS VALUES ('user_id', 'product_id', 'review_text') |
| **Information from PreEvent** | • queryTime: get the time the review was submitted to construct the time window<br>• query: get the type of command (i.e., Insert) using the getParsedQuery() operation<br>• query: the data item accessed (i.e., Reviews) using the getParsedQuery() operation<br>• query: extract the id of the user who is submitting a review to count the number of reviews this account has submitted within the time window |
| **Information from PostEvent** | • No post event will be generated as we want to prevent the execution of the query |

| **Scenario 11** |
|---|

| | |
|---|---|
| **Goal** | Tracking suspicious reviews (b) |
| **Description** | A user submitting reviews without actually having bought the product. This could be either a fake review from an individual or a bot to generate false review of a product causing it to lose customers. The events analytics should be able to detect such action to prevent it. |
| **Monitoring Queries** | INSERT INTO REVIEWS VALUES ('user_id', 'product_id', 'review_text')<br><br>INSERT INTO ORDERS VALUES ('product_id', …, 'user_id') |
| **Information from PreEvent** | • query (for both types of queries): get the type of command (i.e., Insert) using the getParsedQuery() operation<br>• query (for both types of queries): the data item accessed (i.e., Reviews and Orders) using the getParsedQuery() operation<br>• query (for the second type of query): extract the id of the users who placed an order for a specific product.<br>• query (for the first type of query): extract the id of the user who is submitting a review and the id of the product for which the review is submitted. Try to match if this user has bought the product (by checking the events collected from the previous bullet point) |
| **Information from PostEvent** | • No post event will be generated as we want to prevent the execution of the query |

## 5.3 USE-CASE SPECIFIC SCENARIOS

The architecture and the data event metamodel presented in Sections 0 and 5.1, respectively, can be used by the TYPHON project industrial partners to facilitate the implementation of analytics taking place in their business.

For example, GMV is interested in using the analytics engine to be able to filter and store in the database information about forest areas in which the percentage of healthy trees is high or low. Using the proposed analytics architecture and data metamodel, the images that are received from the satellite and are about to be inserted in the polystore can firstly be provided to the analytics infrastructure for further analysis. Every time an insert event is triggered the image can be passed to the pre-existing libraries developed by GMV to extract metadata and information regarding the number of healthy trees in the area. Images with healthy trees below (or above) a specific threshold will be stored in a database for further analysis while the rest will be archived.

Regarding Alpha Bank, exemplar analytics scenarios of interest that use the proposed architecture and metamodel are described below. Alpha bank would be interested to check real-time (debit/credit card) transactions and identify how their customers are behaving and what types of products they are buying during special events (e.g., Boxing Day, Black Friday, etc.). Using real-time analytics, they will be able to offer customised products immediately (e.g., loans or lower interest rates) to their customers that match a specific profile.

Another scenario would be that of fraud detection. Accounts that have no transactions for a specific period are marked as inactive (dormant). Alpha Bank would be interested to monitor activation of dormant accounts that belong to clients above a certain age and issue alerts as it might be the case that the account was activated by a third person who

knows that the account owner is deceased and tries to get hold of the savings. To further protect these types of accounts, any transaction from a recently activated dormant account should be blocked if it doesn't match the profile of the account owner (e.g., an elderly person who uses the account to buy video games). The proposed architecture and data event metamodel offers the facilities needed to accommodate such scenarios.

# 6. JAVA CODE GENERATION

In order to facilitate the development of analytics using the proposed architecture described in Section 0, we need to offer the code that will be used by TyphonQL to instantiate database events. In addition, the same codebase can be used by analytics experts to manipulate those events. As the proposed architecture relies on frameworks like Apache Kafka [16] and Apache Flink [17], which are developed in Java and Scala, we need to provide implementations that support them. In this project we opted for the Java programming language due to its broader audience. Figure 4 presents the code generation process which is described in detail in Section 6.1 that follows.

## 6.1 CODE GENERATION PROCESS

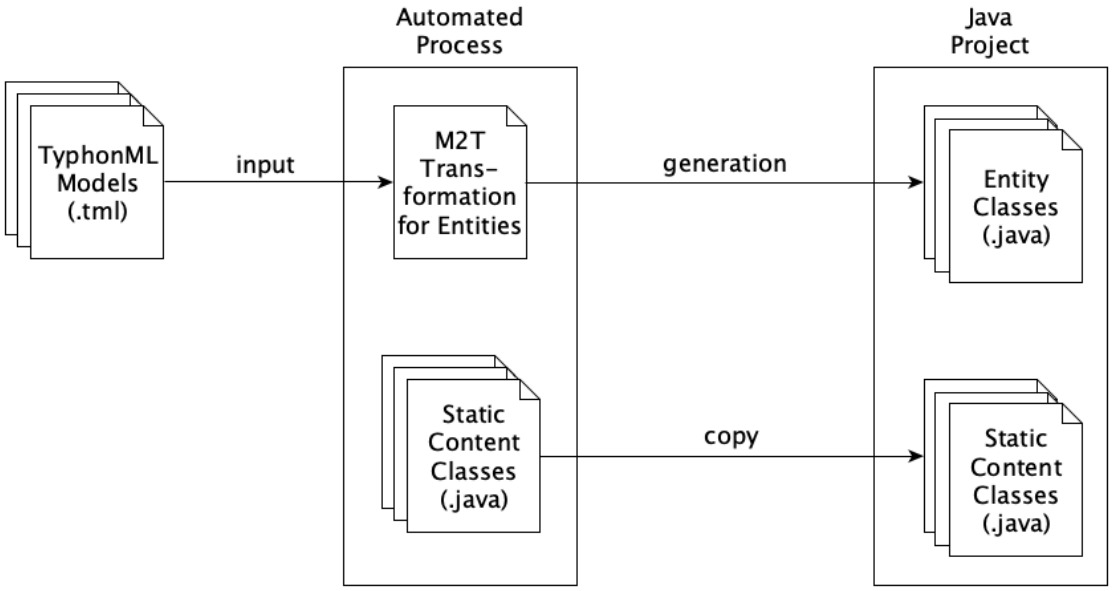


**Figure 4: Code generation process overview**

By looking at the data event metamodel in Figure 2 for the e-shop example, one can see that there are two categories of classes in the metamodel. The static content classes, i.e., those that will always be the same regardless of the content of the TyphonML models (appear in white in Figure 2) and the dynamic, i.e., those that need to be generated based on information taken from the TyphonML models (in color in Figure 2). Thus, the automated process of generating these artifacts consists of two phases. The first transfers the static content Java files in a new Java project[4]. The second uses model-to-text (M2T) transformations to generate one class for each element of type *Entity* in the TyphonML model. Each of these classes will extend the Entity class and will have a

[4] Another option would be to bundle these static classes in a library and reuse them instead of replicating them.

field and the appropriate getters/setters for each of its attributes provided in the TyphonML model.

We developed our solution as an Eclipse plugin. Users just need to select the TyphonML (.tml) file, right click and navigate to TYPHON -> Generate Analytics Code as shown in Figure 5. This will generate a new Java project in which all the static content classes are created. Then the plugin will automatically trigger the execution of the M2T transformation that will create the dynamic content files.

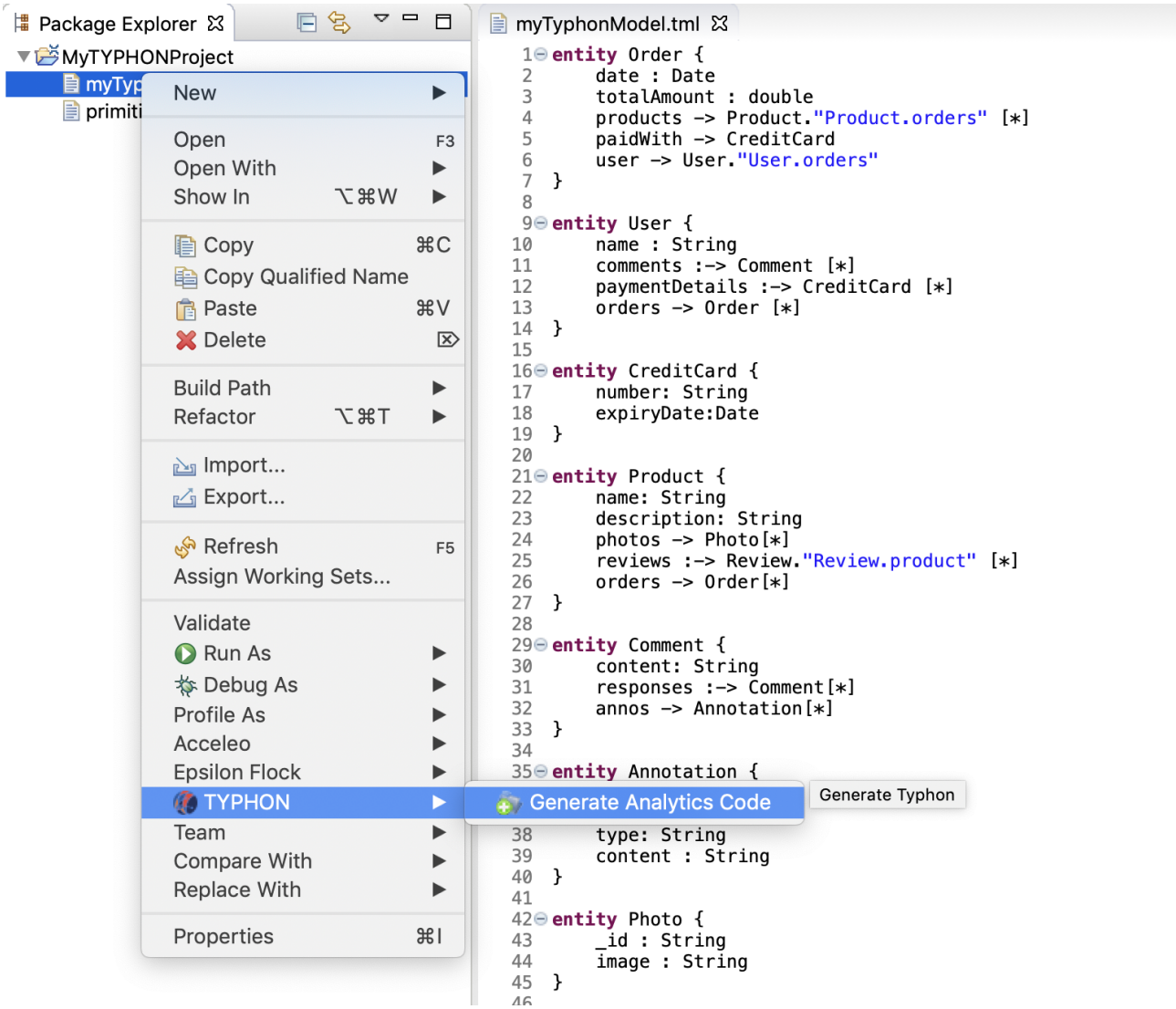


**Figure 5: Eclipse plugin that automates the generation of Java code from a TyphonML model**

We use the Epsilon Generation Language (EGX/EGL) [18] to implement the M2T transformation. The orchestrator EGX script is shown in Listing 6.

```
rule Entity2Class
  transform entity : TML!Entity {

  template : "entitiesGenerator.egl"

  target : "src/org/typhon/entities/" + entity.name + ".java"
}
```

**Listing 6: The EGX orchestrator**

For each of the elements of type Entity in the TyphonML model, we create a new file into the newly created Java project (inside src/org/typhon/entities package). The content of each of the files is created dynamically by the "entitiesGenerator.egl" template which is shown in Listing 7 and explained below.

```
1  package org.typhon.entities;
2
3  import org.typhon.entities.Entity;
4  import java.util.ArrayList;
5
6  public class [%=entity.name%] extends Entity {
7  [%
8  for (anAttribute in entity.attributes) {
9      %]
10
11     private [%=anAttribute.type.name%] [%=anAttribute.name%];
12
13     public [%=anAttribute.type.name%] get[%=anAttribute.name.firstToUpperCase()%]() {
14         return this.[%=anAttribute.name%];
15     }
16
17     public void set[%=anAttribute.name.firstToUpperCase()%]([%=anAttribute.type.name%] [%=anAttribute.name%]) {
18         this.[%=anAttribute.name%] = [%=anAttribute.name%];
19     }
20 [%
21 }
22 for (aRelation in entity.relations) {
23     if (aRelation.cardinality == Cardinality#zero_one or aRelation.cardinality == Cardinality#one) { %]
24
25     private [%=aRelation.type.name%] [%=aRelation.name%];
26
27     public [%=aRelation.type.name%] get[%=aRelation.name.firstToUpperCase()%]() {
28         return this.[%=aRelation.name%];
29     }
30
31     public void set[%=aRelation.name.firstToUpperCase()%]([%=aRelation.type.name%] [%=aRelation.name%]) {
32         this.[%=aRelation.name%] = [%=aRelation.name%];
33     }
34     [%
35     } else { %]
36
37     private ArrayList<[%=aRelation.type.name%]> [%=aRelation.name%];
38
39     public ArrayList<[%=aRelation.type.name%]> get[%=aRelation.name.firstToUpperCase()%]() {
40         return this.[%=aRelation.name%];
41     }
42
43     public void set[%=aRelation.name.firstToUpperCase()%](ArrayList<[%=aRelation.type.name%]> [%=aRelation.name%]) {
44         this.[%=aRelation.name%] = [%=aRelation.name%];
45     }
46     [%
47     }
48 }
49 %]
50 }
```

**Listing 7: The EGL template**

In EGL, all statements outside the [% %] and [%= %] markers are printed as static text (i.e., as shown in the file). In line 6, we create the class signature using as name the property name of the current entity. In lines 8 - 21 we iterate through all the attributes of this specific entity. Firstly, in line 11 we declare the attribute as a private field to the class followed by its type and its name (both taken from the TyphonML model). Then in lines 13 - 15 we create the public getter for this attribute and in lines 17 - 19 the public setter.

As some entities include references to other entities (e.g., foreign key relationships) we need to create fields and public getters/setters for these as well. If the cardinality of the relationship (as denoted by the TyphonML model) is 1 or 0..1, then the type of the field for this relationship is simply the type of the element this relationship points to. For example, the type of the *paidWith* relationship of the *Order* entity (see line 5 of Figure 5) will be *CreditCard*. In contrast, if the cardinality is 0..* or 1..* then the type should

be an ArrayList of items. For example, the type of the *products* relationship of the *Order* entity (see line 4 of Figure 5) will be *ArrayList<Product>*. The code generation for the relationship is done in lines 22 – 48 of Listing 7. More specifically, in line 23 we check the cardinality of the relationship and if that it is 0..1 or 1, we generate the field, the public getter and setter in lines 25, 27-29 and 31-33, respectively. Otherwise (i.e., the cardinality is 0..* or 1..*), we generate the appropriate field, getter and setter in lines 37, 39-41 and 43-45 respectively using the ArrayList datatype.

An example output for an entity "Order" of Figure 5 (lines 1 - 7) is shown in Listing 8

```
1  package org.typhon.entities;
2
3  import java.util.ArrayList;
4  import java.util.Date;
5
6  public class Order extends Entity {
7
8      private Date date;
9
10     public Date getDate() {
11         return this.date;
12     }
13
14     public void setDate(Date date) {
15         this.date = date;
16     }
17
18     private double totalAmount;
19
20     public double getTotalAmount() {
21         return this.totalAmount;
22     }
23
24     public void setTotalAmount(double totalAmount) {
25         this.totalAmount = totalAmount;
26     }
27
28     private ArrayList<Product> products;
29
30     public ArrayList<Product> getProducts() {
31         return this.products;
32     }
33
34     public void setProducts(ArrayList<Product> products) {
35         this.products = products;
36     }
37
38     private CreditCard paidWith;
39
40     public CreditCard getPaidWith() {
41         return this.paidWith;
42     }
43
44     public void setPaidWith(CreditCard paidWith) {
45         this.paidWith = paidWith;
46     }
47
48     private User user;
49
50     public User getUser() {
51         return this.user;
52     }
53
54     public void setUser(User user) {
55         this.user = user;
56     }
57 }
```

**Listing 8: The Java code generated for the "Order" entity of the TyphonML model**

# 7. CONCLUSIONS AND FUTURE WORK

In this document we presented the results of an in-depth domain analysis carried out in order to identify the types of events that a TYPHON polystore should be able to

publish. The consumption of these events, distributed through TYPHON channels, will help the development of orthogonal analytics and monitoring services.

We assessed two relational and four non-relational databases to identify notifications that are triggered when data operations are executed. We then presented examples, through which we demonstrated how the events that databases produce could be useful in developing analytics, extracting information and responding accordingly.

We finally presented the data event metamodel which was extracted by combining the knowledge acquired by studying the events existing databases publish and their potential applications to an e-commerce website scenario and the project's industrial use-cases.

In the future, though our collaboration with the industrial partners we plan to collect more examples that will help in the evolution of the data event metamodel. In addition to that, we expect the metamodel to evolve even more to adapt to design decisions taken as other artefacts of this project (e.g., TyphonQL) are taking shape.

Finally, to accommodate the *time vs space trade-off,* as this explained in Section 5.1, we plan to introduce a configuration mechanism in the analytics suite which will allow the polystore owners to define the level of granularity of the events stored in the analytics queues. For example, an organisation might not want to keep the actual objects inserted, deleted, updated, etc. into the event queues, but prefers to store their ids only and extract them using TyphonQL in order to reduce the space needed for storing events.

# 8. BIBLIOGRAPHY


[1] Oracle Corporation, “MySQL,” 2018. [Online]. Available: https://www.mysql.com/. [Accessed 04 December 2018].

[2] Oracle Corporation, “Database | Cloud Database | Oracle,” 2018. [Online]. Available: https://www.oracle.com/database/. [Accessed 04 December 2018].

[3] K. Chodorow, MongoDB: The Definitive Guide: Powerful and Scalable Data Storage, O'Reilly Media, Inc., 2013.

[4] J. L. Carlson, Redis in Action, Manning Publications Co., 2013.

[5] Neo4j, Inc., “The Neo4J Graph Platform,” 2018. [Online]. Available: https://neo4j.com/. [Accessed 04 December 2018].

[6] The Apache Software Foundation, “Apache Cassandra,” 2016. [Online]. Available: http://cassandra.apache.org/. [Accessed 04 December 2018].

[7] Oracle Corporation, “DBA_AUDIT_TRAIL,” 2018. [Online]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14237/statviews_3056.htm. [Accessed 04 December 2018].

[8] Oracle Corporation, “MySQL :: Security in MySQL :: 7.4.3 Audit Log File Formats,” 2018. [Online]. Available: https://dev.mysql.com/doc/mysql-security-excerpt/5.6/en/audit-log-file-formats.html. [Accessed 04 December 2018].

[9] MongoDB, Inc, “Auditing - MongoDB Manual,” [Online]. Available: https://docs.mongodb.com/manual/core/auditing/. [Accessed 04 December 2018].

[10] I. MongoDB, “Monitoring for MongoDB,” [Online]. Available: https://docs.mongodb.com/manual/administration/monitoring/. [Accessed 22 11 2018].

[11] Neo4j, Inc., “Query Logging,” [Online]. Available: https://neo4j.com/docs/operations-manual/current/monitoring/logging/query-logging/. [Accessed 04 December 2018].

[12] Neo4J, Inc., “Security events logging,” [Online]. Available: https://neo4j.com/docs/operations-manual/current/monitoring/logging/security-events-logging/. [Accessed 04 December 2018].

[13] I. Neo Technology, “Bolt network protocol,” 2018. [Online]. Available: https://boltprotocol.org/. [Accessed 6 December 2018].

[14] Redis, “Redis Slowlog,” 2018. [Online]. Available: https://docs.redislabs.com/latest/rs/administering/logging/redis-slow-log/. [Accessed 17 December 2018].

[15] Redis, “Redis Rsyslog,” 2018. [Online]. Available: https://docs.redislabs.com/latest/rs/administering/logging/rsyslog-logging/. [Accessed 17 December 2018].

[16] A. S. Foundation, “Apache Kafka: A distributed streaming platform,” 2017. [Online]. Available: https://kafka.apache.org/. [Accessed 21 November 2018].

[17] A. S. Foundation, “Apache Flink - Stateful Computations over Data Streams,” 2017. [Online]. Available: https://flink.apache.org/. [Accessed 21 November 2018].

[18] L. M. Rose, F. R. Paige, S. D. Kolovos and A. P. Polack, “The Epsilon Generation Language,” in *European Conference on Model Driven Architecture-Foundations and Applications*, Berlin, 2008.