Maciej PENAR[1] Rzeszów University of Technology, The Faculty of Electrical and Computer Engineering

STREAM PROCESSING SYSTEMS: PROTOTYPE OF ACTIVE DATA WAREHOUSE

S u m m a r y

Stream processing means that each incoming event triggers some response from system. Unfortunately, contemporary streaming systems are being bookshelved and used only for pipeline definition. Sometimes these pipelines require buffering which may lead to processing the events in scheduled (micro)batches. Nonetheless, there is a bigger potential for the streaming as it can be found in systems built using lambda or kappa architecture. This means that the opportunity to load the underlying Data Warehouse naturally emerges – which leads to decreasing the latency and brings us closer to the (near) Real Time Data Warehouse. In this article we present the basic theory concerning the stream processing, we divide the existing systems based on the method of output delivery and we present the prototype Stream Warehouse Query Language which is used to blend the concepts of Date Warehousing, like dimensions, hierarchies and measures, into the SQL-like language.

Keywords: Big Data, Data Streams, Active Databases, Continuous Queries, Stream ETL

[1] Corresponding author: Maciej Penar, Rzeszów University of Technology, The Faculty of Electrical and Computer Engineering, Aleja Powstańców Warszawy 12, 35-959 Rzeszów; mpenar@kia.prz.edu.pl, https://orcid.org/0000-0002-4481-807X

1. Introduction

Database systems (DBs for short) can be divided into two groups which corresponds to two orthogonal concepts of human interaction with these systems [1]. On the one hand, we have Human-Active Database-Passive (HADP) systems where human user is actively working with DB – sending various ad-hoc/prepared queries (DQLs) or commands (DML/DDL/DMLs). This approach forces solutions which supports batch processing. HADP DBs should not perform any computations when no user interacts. On the other hand, we have Database-Active Human-Passive (DAHP) concept which assumes that user awaits the data, which are being continuously sent by DBs in a push fashion [2].

Contemporary HADP systems offer their users so-called `*active components*` - the most common ones being constraints and triggers [3]. However, the overhead of these components is often unsatisfactory for the programmers. Also, a close lookup to the SQL-92 ANSI standard reveals that not all `*active components*` are available in commercial systems as their overhead is simply unacceptable – the most known example is Database Assertion [4].

Active Databases (ADB) are often HADP systems in which the triggers have far more powerful capabilities than those in ANSI standard [5], because ADBs are often event processing systems [6] or their advanced version Complex Event Processing Systems (CEP) [7] [8] which are used to discover patterns in analysed streams. Historically, these systems extends the Sequential Databases which were used to store and query the ordered data [9] [10] [11]. Effectively, they contributed to the OLAP windowed queries [12].

Stream Processing Systems (SPS) are generally non-persistent DAHP systems which aim to process data on-demand, record-after-record. Initially, they were deployed when (1) there was requirement for the low latency, (2) end user had accepted data loss in case of system failure (3) there were soft time requirements. Main difference between event processing systems which process the data based upon the „Best Effort" model and SPS is that the latter are reactive in nature and the underlying data is structural and can be processed using SQL-like language. Specifically, languages like Continuous Query Language (CQL) are used to define the data pipelines. Similarly to SQL, which is founded on the Relational Algebra, the CQL pipeline can be modelled using Stream Algebra [42].

Nowadays, typical use cases of SPS are:

- Complex Event Processing (i.e. Siddhi language)
- Analytical systems (ingestion of data from IoT systems)
- Publisher-Subscriber model
- Specialized Middleware between DB and web application

Over the years, many prototypes has been developed, notably: Aurora [2] [13] [14] [15], Borealis [14] [16] and STREAM [17] from the Stanford University, NiagaraCQ [18] [19], Tribeca [20] and StreamAPAS [21] from the Silesian

University. The importance of the SPS has increased due to the popularity of the Event Sourcing [22] and the Event Storming rapid prototyping method. It is worth noting that as the reactive programming has gained momentum (which is tied to publisher-subscriber model) – many languages incorporated the libraries which leverage the stream processing capability i.e. JDK 9 Reactive Stream in java.util.concurrent.Flow. For the Stream Processing, the defining moment came with the Big Data movement which has shaped the existing systems into the Shared Nothing architecture [23]. As a result of Big Data buzz, we can enjoy ourselves with systems like:

- Spark [24] [25], Storm [26] [27], Flink [28], Kylin [29], Samza [30] for the analytical purposes
- Kafka [31] [32] [33], Redis [34] [35], RabbitMQ [36] for the Publisher-Subscriber model
- PipelineDB [37] [38] as a middleware between DB and application layer
- KSQL as a CQL engine with Kafka as a primary storage [39].

Although there is plethora of SPS technologies, not all systems fulfil the requirements formulated in [40]. Especially, criterium no. 2 concerning SQL-like languages. Finally, the SPS has been used in conjunction with Command And Query Responsibility Segragation (CQRS), Lambda and Kappa [41] architectures, though not many articles exist which could scientifically establish gains of such architectures.

The still unexplored area-of-research concerning the SPS is the topic of Active Data Warehouses (or Real-Time Data Warehouses RTDW) and OLAP on data streams. The reasons for this are following:

1. The methodologies for building the "classical" DW are pretty solid (i.e. Inmon, Kimball and Lindstead methodologies)
2. Currently, the use case seems pretty narrow as in order to build the Active Data Warehouse the companies should have already built their application based on the stream-aware architectures (CQRS/Lambda/Kappa)

Furthermore, the "classical" Extract-Transform-Load (ETL) processes can be scheduled more frequently – to behave similarly to the RTWD (or near-Real-Time Data Warehouses nRTWD). Although such manoeuvres can be done, the engineers tend not to speak out loud about the costs that such solutions should require. StreamOLAP systems aim to deliver nRTDW at low cost. Prototype for such system will be presented furtherly in the article.

This article is organized as follows: section 2 contains the foundations of the SP and DW. Section 3 presents two cherry-picked contemporary SP solutions: Java Reactive Streams and LinkedIn Kafka. Section 4 is a showcase of our language: Stream Warehouse Query Language (SWQL) – we will present the syntax and the main differences in terms of data delivery with KSQL. Last section is the overview.

2. Streams and Data Warehouses

2.1. Properties of Stream Processing

SPS are founded on the concept of Data Stream – which is often defined as infinite sequence of tuples. It serves as an abstract of storage. Streams are described by the following properties:

- Volatile – means that streams are non-persistent
- Continuous – means that CQ are processes which are hosted in SPS
- Record-by-record processing – means that SPS should provide the response for every incoming tuple. That is why the tuples are often called Events in SPS.
- Homogenity – means that every tuple has similar scheme
- Ordering – means that tuples should be somehow ordered – preferably by their appearance timestamp
- Temporal – means that tuple have a lifespan in SPS. This requires every stream to define the policy of tuple eviction

These properties justify the need to develop new, efficient data structures which are dedicated for the stream processing. In batch processing, lack of assumption how the data will be queried (ad-hoc queries) forces users to tune the database i.e. create indexes and perform partitioning. In SPS the CQs are the data structures which are under heavy load of INSERT/UPDATE commands – the proper choice is of greater importance.

2.2. Temporality

DS are temporal, thus volatile. To model the various methods of tuple eviction we propose the concept of *temporal attribute* denoted TA. It is tied to the *time domain* denoted T. The tuples in the stream S contains their own schema $A(S)$ and the condition $TA \subset A(S)$ should always hold. TA can be divided into separate classes:

- Data-only: $TA(0)$, where $TA \equiv T$. Data in streams is described only by timestamp that is calculated in the moment of tuple arrival
- Uni-Temporal: $TA(1)$, where $TA \coloneqq \{(t_s, t_e) \in T \times T \mid t_s < t_e\}$. Lifespan of the data in the stream is described as the *validity period*. For such streams the output for the query in some point in time t is calculated for tuples for which the condition $t_s \leq t < t_e$ is true.
- Bi-Temporal: $TA(2)$, where $TA \coloneqq \{(t_{ts}, t_{te}, t_{vs}, t_{ve}) \in T^4 \mid t_{ts} < t_{te} \wedge t_{vs} < t_{ve}\}$, where data in stream is described by two validity periods. In general the first one is called *transaction time* and it describes the systems' perspective. The second lifespan has the similar semantics as $TA(1)$.

Very often in $TA(1)$ streams the validity period is considered transaction time. It is worth noting that uni/bi/tri-temporal data is borrowed from the theory of Temporal - exact semantics can be found in [44]. Additionally, two special TA models exist which has nothing to do with T. These are:

- Interpunction streams: $TA(*)$. These are the streams which break the *homogeneity* property as they introduce the concept of *interpunction* – special events which are internally recognized by SPS.
- Positive/negative streams: $TA(+)$, where $TA := \{+, -\}$. Such streams must interpret tuples with $+$ as INSERT-ed tuples and – as DELETE-ed tuples

At last, most of the contemporary SPS enforce the temporality on stream-level by using the notion of *windows*. In our opinion both concepts are very similar – the *temporal attributes* gives more flexibility. Nonetheless, to keep the following sections simple, we will call both concepts abstract *temporal constraint* and we will use ψ to denote it.

2.3. Stream and its buffer

Although the SPS process the data record-by-record, it is a common need to calculate aggregate or perform the query in a batch fashion on valid tuples (in extreme case we can use such TA that streams will resemble relation). This is why the SPS should allow users to query in push and pull fashion. Therefore, we propose two definitions for the SPS storages: *streams* which are used for push queries (definition 1) and *stream buffer* which is used for pull queries (definition 2).

Definition 1

Let the T describe discrete time domain. Stream $S(t)$ in any moment $t \in T$ is a relation of at most one element: $|S(t)| \leq 1$. In any moment the condition $A(S) \subset$ ATTR, where ATTR is set of all valid attribute names, holds.

Definition 1 is formulated to simplify things: we think of at most 1 tuple at a time. In [42] the following definition was presented as stream definition, but we believe that it describes the concept of *stream buffer*. We present it here as definition 2 in a slightly modified version:

Definition 2

Stream buffer $S_B^{\psi}(t)$ for stream $S(t)$ in any given point in time $t \in T$ is a relation with schema $A(S_B^{\psi}) \equiv A(S)$. Stream buffer in any given point in time $t \in T$ is always finite, that means that condition $\left|S_B^{\psi}(t)\right| < \infty$ holds. All tuples in buffer satisfy the temporal ψ $S_B^{\psi}(t) := \{s \in \bigcup_T^{t'} S(t') \,|\psi(s) \wedge t' < t\}$

The word buffer may be misleading as implementation-wise it can be any specialized structure which leverages the eviction policy.

2.4. Stream algebra

Stream algebra operators are analogous to their relational counterparts. However, not all relational operators make sense in SP – some of them can only be applied to the stream buffer. SPS should be flexible in terms of choosing if the operators should use relational or stream semantics. Therefore, to distinguish both types we will denote relational operators by op^{rel}.

The common one-argument operators which works directly on stream are:

- projection $\pi_{A_1,\dots,A_n,f_1(s)\dots,f_m(s)}(S,t) = S'(t)$, where $A(S'(t)) = \{A_1,\dots,A_n,f_1(s)\dots,f_m(s)\}, A_i \in A(S(t)), s \in S$
- selection (filtering) $\sigma_\theta(S,t) = S'(t), S'(t) \coloneqq \{s' \epsilon\, S(t) \mid \theta(s')\}$, where θ is any predicate formed on the schema $A(S)$
- rename $\rho(S)$ which is analogous to the relational $\rho^{rel}(S)$ operator

Other one-argument operators requires reference to stream buffer. These operators are:

- duplicate elimination $\delta(S,t) = S'(t), S'(t) \coloneqq \{s \in S(t) | \sim \exists_{s' \in S_B^\psi} s' = s\}$
- grouping ${}_{G_1,\dots,G_n}\gamma_{F_1(A_1),\dots,F_m(A_m)}(S,t) = S', S' \coloneqq \left\{ s \in \delta^{rel}\left(\gamma^{rel}{}_{G_1,\dots,G_n F_1(A_1),\dots,F_m(A_m)}\left(S(t) \cup S_B^\psi(t)\right)\right)\right\}$, where G_n is set of grouping attribues, $F_m(A_m)$ is an aggregate function F_m applied to A_m

In case of two-argument operators, there are three possible versions of them - both operands can be stream, both operands can be stream buffer or first operand is buffer whereas the second is stream:

- buffer-to-buffer operators – definitions of which do not differ from their relational counterparts i.e all join types($\bowtie$), set operators ($\cup$, $\cap$, $-$). Such operators cannot emit changes.
- stream-to-stream operators – which emits the output for every incoming tuple from any input stream. Currently, the only operand working in such fashion is multiset/concatenation ($\cup$) operator.
- stream-to-buffer operator – which emits the result for every incoming tuple in the stream i.e. stream versions of $\cup$, $\cap$, $-$, $\bowtie$ operators

2.5. Semantics of Continuous Queries

In the passive databases SELECT statements can be interpreted as a subset of table. In active databases few semantics of Continuous Queries exist:

- Periodic CQ, denoted CQ(p) – when CQ schedules the moment of emitting the record to the end-user. In fact CQ(p) do not perform the SP – as systems with such queries are often middleware i.e.. w Spark, Kylin
- On-Demand CQ, denoted CQ(d) – when user has to directly submit SELECT statement addressing the CQ and the query is evaluated on-line. Although this CQ requires batch processing, due to the tuple expiration policy these systems do utilize specialized streaming concepts i.e. PipelineDB
- Active CQ, denoted CQ(a) – when response from CQ is being pushed to the end-user i.e. Kafka, Storm

Table 1 addresses the concern of different perception of incoming/outgoing data on different system boundaries.

Currently, there is a lack of systems which provide the advanced CQ(a) – streams with dynamic tuple timespans (flexible TA), aggregating data on-line, and Stream OLAP (at least author did not found any). Very promising middle-ware which enables its users to perform CQ(a) and group data on-line (with some restrictions) is relatively new tool KSQL [39].

Table 1. Classification of Continuous Queries on the borders of the systems

Tabela 1. Podział zapytań ciągłych wg. rodzaju przetwarzania na granicach systemu

		Boundary	
		System	*Client*
CQ Type	*CQ(p)*	Batch	Stream (Period batch)
	CQ(d)	Stream	Batch
	CQ(a)	Stream	Stream

2.6. Data Warehouses

Data Warehousing (DW) is the process which aims to transform, load and present the data for end-user. Very often DW requires the databases as the storage engine. These databases feed the Business Intelligence solutions (which visualise the data) with data. Due to the complexity of warehousing processes and the often recurring architectural patterns, the three practical approaches have been widely acclaimed:

- Inmon Methodology– which bases on concept of DW databases as normalized relational model (Snowflake schema)
- Kimball Methodology – which bases on concept of DW database as denormalized relational model (Star schema)
- Linstead Methodology – where the heterogeneity of records and adaptiveness to changes is highlighted (Data Vault)

Kimballa/Inmona methodologies shares the naming of some of the warehousing concepts – author wants to mention that Linstead introduced new ideas, but they are very similar.

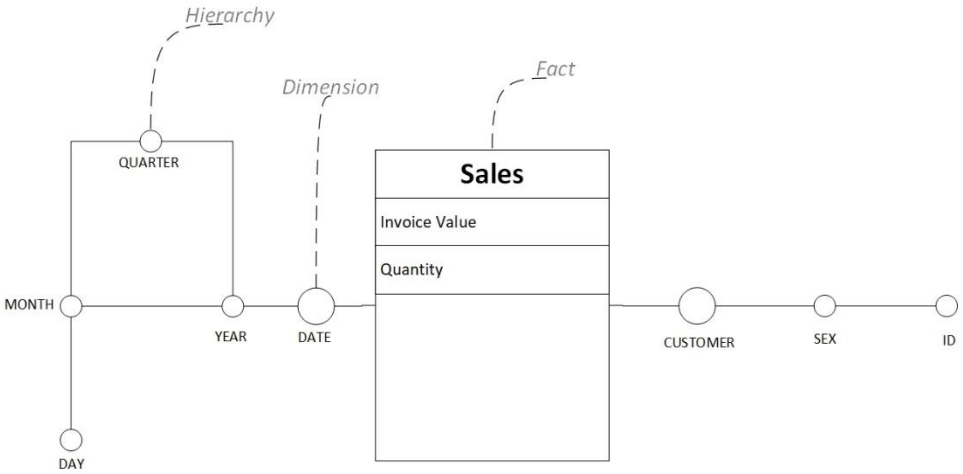

Fig. 1. Example of logical Data Warehouse in dot notation

Rys. 1. Przykład modelu logicznego Hurtowni Danych w notacji kropkowej

Fundamentally, DW is used to store so-called *facts* – which describe the processes or events in a measurable fashion. Instance of fact has measures – these are specialized attributes which describe the magnitude of event. Measures are often numeric values. Facts are analysed based on dimensions which represent the ways of grouping/generalizing of the data. Dimensions have attributes which relate to each other inclusively, forming hierarchy. Although mathematical models for DW exist, they are considered rather unusable for the practitioners – because they signify nothing. In DW the facts are often queried using OLAP standard: Roll-up, Drill-down, Slice&Dice, Pivot operations should be provided by the storage engine.

Example of DW is presented in the figure 1 where we can observe Sales Fact in dot notation. Its measures are invoice value and quantity of items on invoice. We describe this fact by time dimension (when the invoice was created, with hierarchy of Year->Month->Quarter->Day) and customer dimension (who has done purchase, with hierarchy Sex->Id used for client segmentation)

2.7. Real-Time Data Warehouses

As the DW are loaded from domain systems using Extract-Transform-Load processes (ETLs), the latency Δt occurs – the difference in time between

the time the event occurred in source and the moment it was persisted in DW. As a rule of thumb - Δt of 1 day is acceptable for most of the business. However, when the decision-making process relies on split-second decision – such time cannot be accepted. This notion had sprung the concept of Real-Time Data Warehouses (RTDW) or Near-Real-Time Data Warehouses (nRTDW). Such systems which often require Lambda/Kappa source systems aim to obtain $\Delta t \approx 0$s.

The biggest challenge for these system is on-line hierarchy updates. Given H independent hierarchies the number of required updates is equal $\prod_i^H h_i + 1$. nRTDW built on top of SPS is currently hypothetical – as the SPS impose the memory limitations, thus requiring the efficient eviction policy. Every evicted tuple would also trigger $\prod_i^H h_i + 1$. updates..

This hypothetical system could be built as a network of publisher-subscriber queues. The main stream (or its union) would emit the record to queues representing the lowest level of aggregation. Then each dimension would emit records towards its highest level – as presented in figure 2.

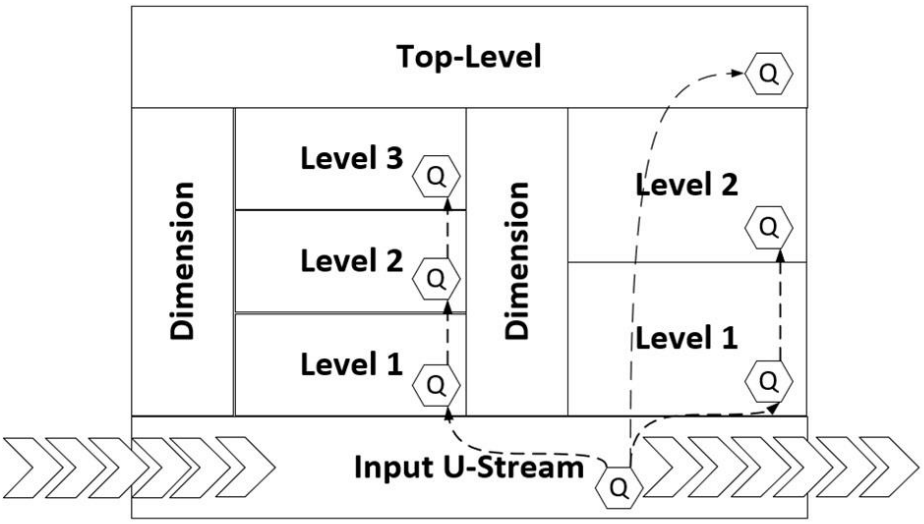

Fig. 2 Diagram for the hypothetical nRTDW representing flow of the events from raw input stream (U-Stream) to the underlying streams (levels of hierarchy / root)

Rys. 2. Diagram reprezentujący przepływ danych z surowego strumienia (U-Stream) do pozostałych strumieni w hipotetycznej nRTDW

In this model, the obvious disadvantage is that main stream has to emit record H +1 times. Also, this concept does not support the aggregates on mixed level. Finally the biggest problem is that such system should provide the DDL syntax for

mapping the DW concepts into the network of streams – currently we are not aware of any work describing such language. Given that, in section 4 we propose draft of the language for such system.

3. Accessing the streams – contemporary approaches

3.1. JDK 9 Reactive Streams

Java Reactive Streams (JRS) API available in JDK 9 are not SPS per-se. However, we decided to provide short description of this API to stress that modern languages incorporate the stream processing in order to keep up with the craze. In future, such foundations may be used to build general purpose SPS.

Main class of JRS is *Flow* which encapsulates 4 interfaces (Fig. 3) analogous to the concepts of Publisher-Subscriber system.

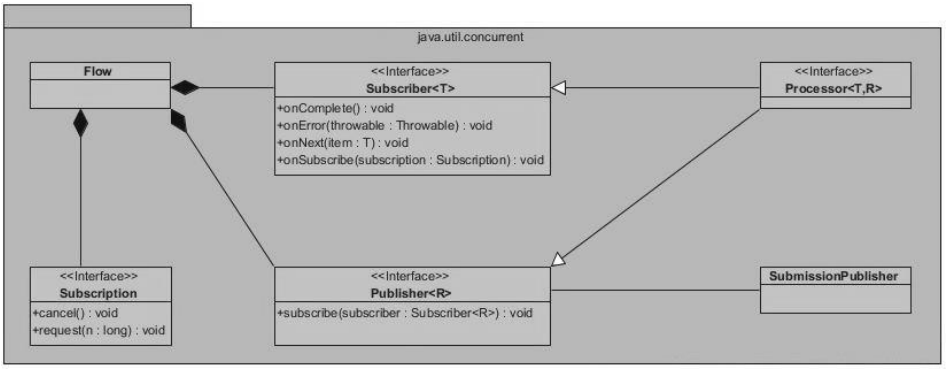

Fig. 3. UML diagram of Flow class in java.util.concurrent

Rys. 3. Diagram UML przedstawiający zależności w ramach klasy Flow w pakiecie java.util.concurrent

These three interfaces are:

1. Subscription – point of entry for SP. Classes which implements this interface represent the intention of the subscription – a concept very similar to stream.
2. Subscriber – this interface represents consumption behaviour – the processing is done in the onNext(T) method which is executed on every incoming event.
3. Publisher – this interface represents the process of feeding the stream with data. The publishers are often classes performing vast amount of IO work. In general Publishers interfaces in various languages tend not

to specify the terms of publishing the records into stream/subscripton, but rather serve as a binding class to Subscriber.

Additionally in *Flow* class there is default implementation of Publisher: *SubmissionPublisher*. It cover simple (and common) case of submitting a single event in a Subscription (submit(T) method).

There is an extra interface: Processor which is a mix of Publisher and Subscriber interface. It is used as a proxy between Publisher and Subscriber – it can be perceived as an abstract operator.

3.2. Kafka

Apache Kafka [31] [49] is a distributed log system. It was designed with a heavy disk IO in mind, thus it boasts being distributed and fault-tolerant. Additionally, as the logging process has similar requirements as publisher-subscriber model (as application "publish" logs) Kafka has also similar functionality to RabbitMQ. Kafka users have fine control over the data flow which is a pro and a con at the same time as the developers have to take care of it manually.

Kafka has only one way to exchange data – *topic exchange*: this is tied to the fact that topic is a storage abstraction. Internally topic is divided into *N* partitions. This is presented in figure 4.

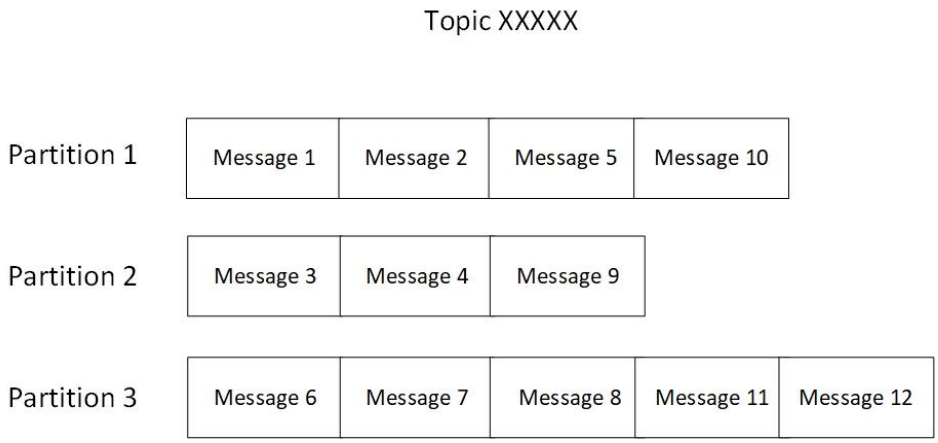

Fig. 4. Logical model of Apache Kafka topic

Rys. 4. Model logiczny wątku w Apache Kafka

Each of *N* partitions is replicated into *M* distinct (if possible) nodes. Administrators can choose the $N \geq 1$ and $M \geq 1$ values. Partitioning and replication of topics ensures the scalability of the system. However, Kafka is not free of architectural quirks.

Firstly, the actual subscriber of the topic is a Consumer Group abstraction $G(n)$ which contains n consumers. $G(n)$ can be perceived as identifier which is set by each client. When $n > 1$ client with same identifier compete over access to partitions. This extra layer $G(n)$ opens the possibility of scaling Subscribers. One should remember that $n > 1$ means each client received only a part of data. When multiple Consumer Groups of $G(1)$ exists then system behaves functionally identical to topic-exchanged RabbitMQ.

Secondly, when IO is concerned – Kafka can only perform append-to-file. This means that partitions of topic are locally ordered. To establish global order the client has to perform manual sort.

As an open source project, Kafka has several interesting extensions like Kafka Stream API and KSQL. Former grants user the possibility of defining the Topologies – DAG which nodes serve as input streams (Sources), stream processors and outputs (Sinks). Kafka Stream API define streams as infinite, ever-changing source of data [31]. Topology concept is common in many systems i.e. Spark [24] or Storm [26].

For this article, the most interesting extension is KSQL which is a middleware used to define Kafka Stream API topologies using SQL-like language. This extension can be used to create tables [50] which are stateful. By design KSQL delivers output in $CQ(p)$ fashion. Nonetheless, we are not aware if KSQL utilize specialized indexes to optimize groupings.

Two main storage abstraction in KSQL can be created using the following DDL syntax:

- CREATE STREAM [AS SELECT] – used to create streams from the topic or based upon the SELECT statement (w/o aggregations and windowed queries)
- CREATE TABLE [AS SELECT] – used to create tables based on topic or SELECT statement (aggregations and windowed queries are permitted)

General syntax for SELECT in KSQL is:

```
SELECT select_expr [, ...]
FROM from_item
[ LEFT JOIN join_table ON join_criteria ]
[ WINDOW window_expression ]
[ WHERE condition ]
[ GROUP BY grouping_expression ]
[ HAVING having_expression ]
[ LIMIT count ];
```

Apart from common and self-explanatory keywords like WHERE/GROUP BY/HAVING we have LEFT JOIN operator and window definition WINDOW.

4. Language prototype

In this section we present SWQL language created for the prototype of nRTDW. The DW engine was written in Java and the language was implemented using ANTLR framework. SWQL is used to help the user to create set of CQ which resembles the structure from figure 2. We assumed that the events were sent in JSON format.

4.1. CREATE CONTINUOUS WAREHOUSE

CREATE CONTINUOUS WAREHOUSE is main command in SWQL (analogous to the CREATE TABLE in SQL) used to register the „continuous data warehouse". Its syntax is presented in figure 5 as railroad diagram..

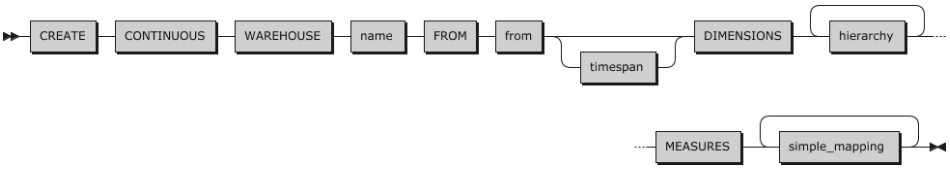

Fig. 5. CREATE CONTINUOUS WAREHOUSE syntax

Rys. 5. Składnia komendy CREATE CONTINUOUS WAREHOUSE

Definition of this abstraction requires specification of (nonterminal symbols):

- *name* – used internally to address the structure
- *from* – as input stream source in URI format. Can be socket or Kafka topic
- (optional) *timespan* – used to define the TA
- dimensions (section DIMENSIONS, nonterminal *hierarchy*)
- measures (section MEASURES, nonterminal *measures*)

Defined by this command DW is placed in our system and from now on it is being continually feed by the data from input stream. After the input stream is consumed the data is propagated to the dimensions.

4.2. Time constraints

Time constraints can replace the *timespan* nonterminal and its syntax is presented in figure 6.

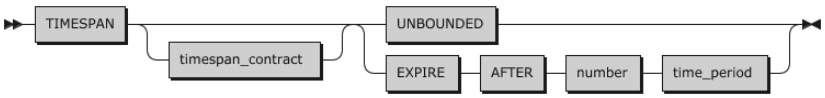

Fig. 6. TIMESPAN syntax for defining temporal attribute

Rys. 6. Składnia ograniczeń czasowych TIMESPAN

User can choose one of two options – called contracts (*timespan_contract*):

- SYSTEM TIME – default option which sets the beginning of the tuple lifespan as the time of arrival
- BUSINESS TIME *field* – this option enables the user to specify which field (nonterminal *field)* contains the information about the beginning of the lifespan

Expiration of the tuple is set with the following policies:

- UNBOUNDED – means that tuples will not expire
- EXPIRE AFTER *number* – means that tuples expire after some amount of milliseconds (MILLIS), seconds (SECONDS), minutes (MINUTES) or hours (HOURS)

4.3. Dimension and measures definition

The most crucial element for the designer of the DW is the DIMENSIONS section which is used to define hierarchies. Syntax is presented in figure 7.

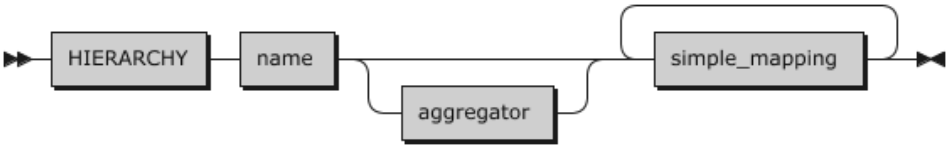

Fig. 7. Syntax for defining hierarchies

Rys. 7. Składnia definiowania hierarchii

User has the flexibility to define any number of hierarchies, but each one is required to have an unique name (nonterminal *name*). The *aggregator* symbol can be replaced to define the default aggregation function – used as a fallback later in SELECT statements. The format of aggregator is DEFAULT AGGREGATE *name*, where name can be replaced with SUM, AVG, MIN, MAX and COUNT. If this section is omitted the SUM function is assumed. At the end of

DIMENSIONS the user defines list of *simple mappings* (fig. 8) – these are the attribute names from JSON with optional AS clause used for renaming (similarly as in SQL).

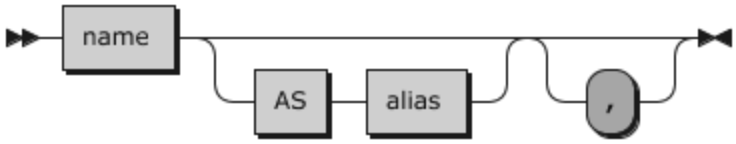

Fig. 8. Syntax for simple_mapping non-terminal

Rys. 8. Składnia prostego odwzorowania (simple_mapping)

Each consecutive position on the mapping list is a level of hierarchy. Last element in the list is the most fine grained level. Similarly the designer of the DW defines the list of measures in MEASURES section. The only additional requirement for measures is that values of the attributes should be numerical, but the check cannot be done in compile time.

4.4. Queries

The next area of our SWQL is the syntax for the CQ (figure 9). Although it resembles the SQL, it should be noted that semantics of the query is closer to the slice & dice OLAP operator. Users can access any of the aggregate on any level of the hierarchy.

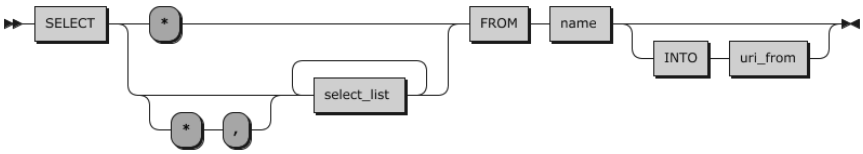

Fig. 9. Syntax for queries in SWQL

Rys. 9. Składnia zapytań w SWQL

As in SQL, FROM clause should contain internal name of the storage abstraction. Also, we borrowed INTO clause to force the system to push the data from it – the user can specify the destination using URI format i.e. WebSocket. The main section is the SELECT clause which is used to define mappings for the current CQ. We allow two types of mappings:

1. Simple SELECT mappings (Fig. 10)
2. Aggregate mappings (Fig. 11)

Thanks to the simple mappings user can obtain the attributes from triggering record (from raw stream) or from any valid DW attribute that maps from the triggering record. To reference the level of the hierarchy we propose the double-colon notation i.e: DATE_DIM::Year (dimension DATE_DIM, level Year). Additionally, user can change the alias of the output attribute using AS clause.

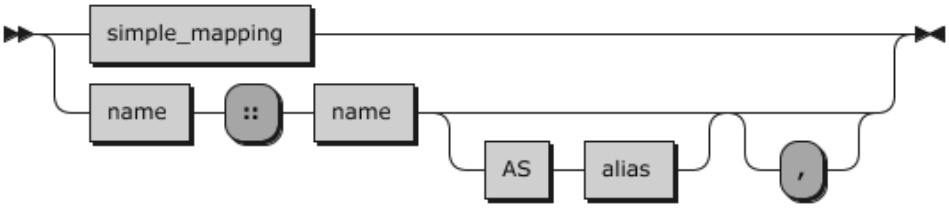

Fig. 10. Syntax for simple-select mapping in SWQL

Rys. 10. Składnia odwzorowania prostego-select w SWQL

The most crucial feature is the ability to reference the aggregates for a given level of the hierarchy. For the root aggregate we propose simplified syntax: AGGREGATE SUM(Quantity). In case when the user is interested in aggregate on the specific level of hierarchy – we use the AGGREGATE *name* ACROSS *dimension_path* clause. The *name* non-terminal is the name of the measure to be aggregated and *dimension_path* is aggregate function with level specification using double-colon notation i.e. AGGREGATE value ACROSS SUM(PLACE_DIM :: unit)

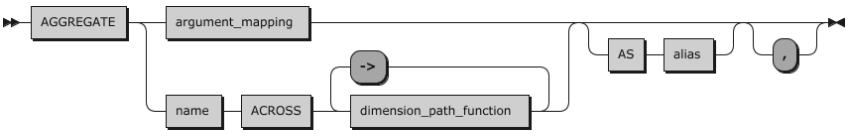

Fig. 11. Syntax for aggregate mapping in SWQL

Rys. 11. Składnia odwzorowania agregującego w SWQL

Currently, SWQL does not support WHERE clause. The filtering semantics are not clear – we did not yet decided if the filtering should operate on raw stream or if it should influence triggering events using the DW concept of dimension members. We believe that the latter approach is desired but it requires SWQL to be a mix of SQL and MDX.

4.5. Comparing the prototype output

We compare how our prototype delivers output with KSQL. The observations were made using the JSON formatted data as presented in figure 12. The data is organized to copycat the denormalized DW structure: the records contain dimensions like Date, Time and Unit, as well as some measures.

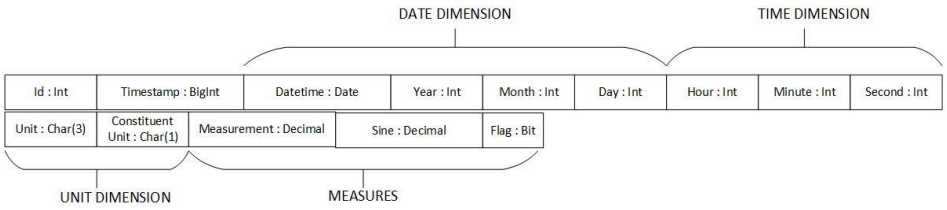

Fig. 12 Format of the record used in testing and its mapping to the DW model

Rys. 12. Format rekordu wykorzystywanego w testach oraz jego odwzorowanie na model DW

During the comparison we emitted 5 records to the KSQL CQ and to our system. The output stream should contain records grouped on *constituent_unit* and *constituent_unit+unit*. Also we required 15 seconds lifespan for the records. The output of KSQL is presented in table 2 and contains changes of every grouping, triggered by every incoming record. However, records which at some point fall out of the window scope (we used tumbling window) does not notify users of current value of aggregate. Also, we want to note that KSQL does not allow its users to define GROUPING SETS – so in case of KSQL we had to simplify the query and group only *constituent_unit*.

Table 2. CQ output from Kafka

Tabela 2. Wynik CQ z Kafki

Id	Rowkey	constituent_total
1	H-O : Window{start=1583594610000 end=-}	87.43
2	P-Z : Window{start=1583594610000 end=-}	92.11
3	A-G : Window{start=1583594610000 end=-}	88.67
4	P-Z : Window{start=1583594610000 end=-}	134.2
5	H-O : Window{start=1583594610000 end=-}	144.57

Observing the output from out prototype system is very similar in the beginning – the first 5 records is virtually the same as from KSQL – with the difference that our system can access multiple GROUPING SETS. The important distinction occurs when the tuples expires – our system notifies the user about the current state of aggregates (table 3).

Table 3. CQ output from our prototype system

Tabela 3. Wynik CQ z prototypowego system

Id	constituent_unit	Unit	constituent_total	unit_total	Count
1	H-O	M	87.43	87.43	1
2	P-Z	V	92.11	92.11	2
3	A-G	C	88.67	88.67	3
4	P-Z	Y	134.2	42.09	4
5	H-O	I	144.57	57.14	5
6	H-O	M	57.14	0.0	4
7	P-Z	V	42.09	0.0	3
8	A-G	C	0.0	0.0	2
9	P-Z	Y	0.0	0.0	1
10	H-O	I	0.0	0.0	0

Additionally, we requested COUNT(*) from our system – which is the number of records currently held by the buffer. At some point it had all 5 valid records and the last event has the value equal to 0 – because our DW does not have any valid record.

5. Summary

In this article we presented the brief overview of the Stream Processing concept and we tied it with the Data Warehousing. We pointed that existing streaming systems can be divided into three different categories which differ in the philosophy of the output delivery. We also noted that contemporary systems does not provide advanced OLAP on streams.

Against all odds, we proposed the prototype of language: Stream Warehouse Query Language, and we implemented the execution engine for it to deliver proof-of-concept for Real Time Data Warehouses on streams. Given the novel approach in terms of data delivery, we believe that it is hard to compare at this point the efficiency of our solution to others. After implementing the prototype we identified three major problems:

1. In SWQL WHERE clause semantics is drastically different than in SQL and may require incorporating parts of the MDX syntax
2. We have not found satisfying syntax for specifying the aggregates which depend on levels of few hierarchies
3. We have not developed data structure supporting efficient tuple expiration and collecting aggregates on multiple levels

As we mentioned, the first problem can be overcome by exploring MDX slice & dice syntax. To our knowledge – the solution for the second problem is non-existent in the current research. For the third problem, some work has been published.

Bibliography

[1] Babcock B., Babu S., Datar M., Motwani R., Widom J.: Models and Issues in Data Stream Systems, Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2002.

[2] Abadi D.J., Carney D., Cetintemel U., Cherniack M., Convey C., Lee S., Stonebraker M., Tatbul N., Zdonik S.: Aurora: A New Model and Architecture for Data Stream Management, 2003, s. 120-139.

[3] Cochrane R., Pirahesh H., Mattos N.M.: Integrating Triggers and Declarative Constraints in SQL Database Sytems, Proceedings of the 22th International Conference on Very Large Data Bases, 1996, s. 567-578.

[4] Stachour P., Thuraisingham B.: SQL Extensions for Security Assertions.

[5] McCarthy D.,Dayal U.: The Architecture of an Active Database Management System, Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, 1989, s. 215-224.

[6] Paton N.W., Diaz O.: Active Database Systems, ACM Comput. Surv., 1999.

[7] Flouris I., Giatrakos N., Garofalakis M., Deligiannakis A.: Issues in Complex Event Processing Systems, IEEE Trustcom/BigDataSE/ISPA, Helsinki, 2015.

[8] Robins D.B.: Complex Event Processing, 2010.

[9] Seshadri P., Livny M., Ramakrishnan R.: The design and implementation of a sequence database system, 1996.

[10] Seshadri P., Livny M., Ramakrishnan R.: SEQL A Model for Sequence Databases.

[11] Seshadri P., Livny M., Ramakrishnan R.: Sequence Query Processing, Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, 1994.

[12] Sadri R., Zaniolo C., Zarkesh A., Adibi J.: Optimization of Sequence Queries in Database Systems, Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Santa Barbara, California, USA, 2001.

[13] Abadi D.J., Carney D., Cetintemel U., Zdonik S.: Aurora: A Data Stream Management System, Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, 2003.

[14] Cetintemel U., Abadi D.J., Ahmad Y., Balakrishnan H., Balazinska M., Cherniack M., Hwang J.H., Madden S., Maskey A., Rasin A., Ryvkina E., Stonebraker M., Tatbul N., Xing Y., Zdonik S.: The Aurora and Borealis Stream Processing Engines, 2016.

[15] Zdonik S.B., Stonebraker M., Cherniack M., Çetintemel U., Balazinska M., Balakrishnan H.: The Aurora and Medusa Projects, IEEE Data Eng. Bull., 2003, s. 3-10.

[16] Ahmad Y., Balazinska M., Çetintemel U., Cherniack M., Hyon J.H., Lindner W., Rasin A., Ryvkina E., Tatbul N., Xing Y., Zdonik S.: The design of the Borealis stream processing engine, 2nd Biennial Conference on Innovative Data Systems Research, 2005, s. 277-289.

[17] Arasu A., Babcock B., Babu S., Cieslewicz J., Ito K., Motwani R., Srivastava U., Widom J.: Stream: The stanford data stream management system, 2004.

[18] Naughton J.F., DeWitt D.J., Maier D., Aboulnaga A., Chen J., Galanis L., Kang J., Krishnamurthy R.: The Niagara Internet Query System, IEEE Data Eng. Bull, 2001, s. 27-33.

[19] Chen J., DeWitt D.J., Tian F., Wang Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases, SIGMOD Rec, nr 6, 2000, s. 379-390.

[20] Sullivan M., Heybey A.: Tribeca: A System for Managing Large Databases of Network Traffic, In USENIX, 1998, s. 13-24.

[21] Gorawski M., Aleksander C.: StreamAPAS: Query Language and Data Model, CISIS '09. International Conference on Complex, Intelligent and Software Intensive Systems, Fukuoka, 2009.

[22] Fowler M.: Event Sourcing,12 12 2005, https://martinfowler.com/eaaDev/EventSourcing.html. (dostęp: 1 czerwca 2018 r.).

[23] Stonebraker M.: The Case for Shared Nothing, IEEE Database Eng. Bull, 1985, s. 4-9

[24] Apache Spark, Apache, https://spark.apache.org/. (dostęp: 1 czerwieca 2018 r.).

[25] Zaharia M., Xin R.S., Wendell P., Das T., Armbrust M., Dave A., Meng X., Rosen J., Venkataraman S., Franklin M.J., Ghodsi A., Gonzalez J., Shenker S., Stoica I.: Apache spark: A unified engine for big data processing, w Communications of the ACM, nr 59, 2016.

[26] Apache Storm, Apache, http://storm.apache.org/. (dostęp: 1 czerwieca 2018).

[27] Batyuk A., Voityshyn V.: Apache storm based on topology for real-time processing of streaming data from social networks, IEEE First International Conference on Data Stream Mining Processing (DSMP), 2016.

[28] Apache Flink, Apache, https://flink.apache.org/, (dostęp: 1 czerwieca 2018).

[29] Apache Kylin, Apache, http://kylin.apache.org/, (dostęp: 1 czerwieca 2018).

[30] Apache Samza, Apache, http://samza.apache.org/, (dostęp: 1 czerwieca 2018).

[31] Apache Kafka, Apache, https://kafka.apache.org/. (dostęp: 1 czerwieca 2018).

[32] Kreps J., Narkhede N., Rao J.: Kafka: A distributed messaging system for log processing, In Proceedings of 6th International Workshop on Networking Meets Databases, Athens.

[33] Zaharia M., Chowdhury M., Franklin M.J., Shenker S., Stoica I.: Spark: Cluster Computing with Working Sets, Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, Boston, 2010.

[34] Redis, https://redis.io/, (dostęp: 1 czerwieca 2018 r.).

[35] Macedo T., Oliveira F.: Redis Cookbook, O'Reilly Media, Inc., 2011, s. 26-30.

[36] RabbitMQ, Pivotal, https://www.rabbitmq.com/, (dostęp: 1 czerwieca 2018 r.).

[37] PipelineDB, https://www.pipelinedb.com/, (dostęp: 1 czerwieca 2018r.).

[38] Yemelianov A.: PipelineDB: Working with Data Streams, 2017, https://blog.selectel.com/pipelinedb-working-data-streams/, (dostęp: 1 czerwieca 2018r.).

[39] Narkhede N.: Introducing KSQL: Open Source Streaming SQL for Apache Kafka, 2017, https://www.confluent.io/blog/ksql-open-source-streaming sql-for-apache-kafka/, (dostęp: 1 czerwieca 2018r.).

[40] Stonebraker M., Çetintemel U., Zdonik S.: The 8 requirements of real-time stream processing, SIGMOD Rec., 2005, s. 42-47.

[41] Kappa Architecture, http://milinda.pathirage.org/kappa-architecture.com/, (dostęp: 1 czerwieca 2018 r.).

[42] Brodt S., Bry F.: Temporal Stream Algebra, 2012.

[43] Nascimento M.A., Eich M.H.: Decision Time in Temporal Databases, In Proceedings of the Second International Workshop on Temporal Representation and Reasoning, 1995.

[44] Kraft P., Sørensen J.O.: Semantics of Temporal Models.

[45] Soman A., Jacob S.: Lambda architecture: Working, advantages limitations, and its, International Journal Of Advanced Research, Ideas and Innovations In Technology, 2018.

[46] Suhothayan S., Narangoda I.L., Gajasinghe K., Nanayakkara V.: Siddhi: A Second Look at Complex Event Processing Architectures, GCE'11 - Proceedings of the 2011 ACM Workshop on Gateway Computing Environments, Co-located with SC'11, 2011.

[47] AMQP, OASIS, https://www.amqp.org/, (dostęp: 1 styczenia 2019 r.).

[48] ISO/IEC 19464:2014 Advanced Message Queuing Protocol (AMQP) v1.0 specification, https://www.iso.org/standard/64955.html.

[49] Narkhede N., Shapira G., Palino T.: Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale, O'Reilly Media, Inc., 2017.

[50] Sax M.J., Wang G., Weidlich M., Freytag J.C.: Streams and Tables: Two Sides of the Same Coin, Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, New York, NY, USA, 2018.

[51] Calcite A.: Apache Calcite, https://calcite.apache.org/, (dostęp: 7 kwietnia 2019 r.).

[52] Gorawski M., Gorawska A., Pasterak K.: A Survey of Data Stream Processing Tools, Information Sciences and Systems 2014, 2014, s. 295-303.

[53] I. 19464, Information technology — Advanced Message Queuing Protocol (AMQP) v1.0 Specification, 2014.

SYSTEMY PRZETWARZANIA STRUMIENI DANYCH: PROTOTYP AKTYWNEJ HURTOWNI DANYCH

Strumieniowy model przetwarzania strumieni danych zakłada, że każdy napływający rekord poddany jest uprzednio zdefiniowanemu potokowi przetwarzania. Niestety współczesne systemy ograniczają się do definiowania potoków przetwarzania, które w pewnych warunkach mogą wymagać przejścia w tryb wsadowy

– emitujący rekordy w pewnych cyklach. Niemniej jednak, model ten daje się znaleźć w systemach budowanych w oparciu o współczesne architektury aplikacji takie jak lambda czy kappa, a w takich sytuacjach, w naturalny sposób pojawia się okazja by dane trafiające do systemu dziedzinowego zasilały od razu systemy analityczne takie jak Hurtownie Danych. Tym samym zmniejszając czas latencji, prowadzący do uzyskania Hurtowni Danych Czasu Rzeczywistego. W tym artykule przedstawiamy podstawy przetwarzania strumieniowego, przedstawiamy klasyfikacje systemów oraz idziemy krok dalej i prezentujemy prototypowy język DDL/DQL służący do budowania grup strumieni danych zorganizowanych na podobieństwo modeli Hurtownianych: zawierających wymiary, hierarchie oraz miary.

Słowa kluczowe: Big Data, Systemy Przetwarzania Strumieni Danych, Aktywne Bazy Danych, Silniki Strumieniowe, Zapytania Ciągłe, Strumieniowy ETL