Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Emanuel Silva Ferreira

AIoTA
An IoT Platform On MonetDB

Braga, 2016-12-21

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Emanuel Silva Ferreira

AIoTA
An IoT Platform On MonetDB

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Dr. José Orlando Roque Nascimento Pereira
Dr. Ying Zhang

Braga, 2016-12-21

ACKNOWLEDGMENTS

This thesis is the final step on my masters graduation, and I would to thank several people who helped me during my academic career and the thesis itself. It also brought me the opportunity to do an Erasmus traineeship, which will be my first experience abroad.

I want to thank to the academic community of University of Minho for my formation during the past 5 years. For the most, I am thankful to Dr. José Orlando Pereira for his acceptance to be my supervisor during the thesis, his knowledge about database systems that helped me during this report, his availability and the opportunity he gave to me for the internship at CWI on the second semester. Also I want to thank the personnel from Distributed Systems at University of Minho and HASLab groups for their accommodation.

This thesis could not be done without the help of the community of Database Architectures group from CWI, who I want to thank deeply. First I want to thank Dr. Stefan Manegold for writing my acceptance letter to the Erasmus traineeship. Next I want to appraise the MonetDB Solutions personnel from the group for their help during AIoTA conception: Panagiotis Koutsourakis, Dimitar Nedev, Dr. Hannes Mühleisen, Dr. Sjoerd Mullender and Dr. Niels Nes. A special thanks to Abe Wits, Mrunal Gawade, Phạm Minh Dúc and Bo Tang for the great moments we shared together in our office. At the same time I want to thank Till Döhmen, Benno Kruit, Mark Raasveldt, Dean De Leo, Thibault Sellam, Eleni Petraki, Dr. Bart Scheers, Dr. Romulo Gonçalves and Dr. Eyal Rozenberg for amazing moments during the internship sharing knowledge and happiness.

Meanwhile, another special thanks from the service staff from CWI that help me on the internship. Irma van Lunenburg for her help and availability during my CWI registration. Bikkie Aldeias, Martine Gunzeln and Minnie Middelberg for their help during my first days at CWI. Arjen de Rijke for his help during IT-related installations.

I want to express greater gratitude to my supervisor at CWI Dr. Ying Zhang for her supervision, knowledge about database systems and hospitality. At the same time I want to give appreciation to Dr. Martin Kersten as the founder of the group and MonetDB, who helped building the back-end of AIoTA. Without his help, this thesis would not be able to finish on time.

In the end, I want to send cheers for my family, especially my parents, who raised and taught me during my whole life. Without their help, I could never accomplish this. Also it could not be done with the help of my local living community of Tadim and my friends, to whom I express gratitude.

ABSTRACT

The growth of the Internet and embedded systems have allowed physical devices to collect and exchange data in the Internet-of-Things (IoT). IoT allows objects to be monitored and controlled remotely across an existing network infrastructure, while creating opportunities to assimilate computer systems with the real world. The expansion of IoT's connectivity has lead devices to exchange large amounts of data, due to constantly required monitoring. The output of these devices can be seen as streams with data made available incrementally over time. This has created a new demand to collect, process and analyze IoT data in an efficient and scalable way.

In the meantime, databases have been organizing collections of data for several decades. At a low level, database management systems (DBMSs) to organize data efficiently. In particular, Data Stream Management Systems (DSMSs) have emerged to handle uninterrupted flows of streaming data and integrate them with relational databases [Aggarwal, 2007]. With this objective in mind, DSMSs have distinguished from traditional DBMSs with new architectures, data models, algorithms and specific query languages to deal with streams. As streams are uninterrupted, DSMSs aim to process them incrementally. This lead to the continuous queries concept, where streaming data is processed with small batches each time.

Meanwhile, other database management systems have explored alternate ways to organize data. MonetDB is a pioneer column-oriented relational database management system (RDBMS), storing relations column-wise opposed to rows as the majority of RDBMSs. Columnar-wise storage allows several benefits such as per-column query parallelization, data compression and late materialization. MonetDB is being developed at Centrum Wiskunde & Informatica (CWI) in Amsterdam since 1993, having achieved faster benchmark results than popular RDBMSs such as PostgreSQL [Muhleisen, 2014].

This master thesis has the objective to create a streaming engine over MonetDB while focusing on IoT processing. Amsterdam Internet-of-Things App (AIoTA) is a full-stack application aiming to be integrated easily with IoT devices to collect streaming data, while taking advantage of MonetDB's columnar-wise storage to process it and deliver results immediately.

RESUMO

O crescimento da Internet e dos sistemas embebidos tem permitido expor dispositivos físicos na Internet das coisas (IoT) para a troca de dados. A IoT permite a monitorização de objetos remotamente em infraestruturas de rede criando oportunidades para assimilar a computação com o mundo real. A expansão da conetividade da IoT tem levado esses dispositivos a promover o intercâmbio de grandes quantidades de dados, em maior parte devido a monitorização constante. Os dados resultantes desses dispositivos podem ser visto como *streams*, onde os dados são disponibilizados incrementalmente com o decorrer do tempo. Como consequência, as *streams* criaram uma nova forma de colecionar, processar e analisar dados provenientes da IoT de modo eficiente e escalável.

Ao mesmo tempo nas últimas décadas, as bases de dados tem organizado coleções de dados. A um nível mais baixo, os sistemas de gestão de bases de dados (DBMSs) tem procurado metodologias para organizar os dados de forma eficiente. Em particular, os sistemas de gestão de *streams* (DSMSs) tem emergido com novos métodos para lidar com fluxos de dados inenterruptos e integrá-los com as bases de dados convencionais [Aggarwal, 2007]. Com este objetivo em mente, as DSMSs tem-se distinguido dos DBMSs com novas arquiteturas, modelos de dados, algoritmos e linguagens de interrogação para lidar com *streams*. Como as *streams* são inenterruptas, as DSMSs tem a finalidade de as processar incrementalmente. Isto levou ao conceito de *continuous queries*, onde as *streams* so processadas com pequenas quantidades de cada vez e incrementalmente.

Entretanto outros sistemas de gestão de bases dados tem explorado metodologias alternativas para organizar os dados. O MonetDB é um sistema de gestão de bases dados relacional colunar pioneiro, onde as relações são armazenadas por colunas em vez de linhas como a maioria dos RDBMSs. O armazenamento colunar permite vários benefícios que não seriam possíveis com o armazenamento por linhas tais como paralelização de interrogações por colunas, compressão de dados e materilização mais tardia. O MonetDB tem sido desenvolvido pelo Centrum Wiskunde & Informatica (CWI) em Amsterdão desde 1993, tendo alcançado *benchmarks* com melhores resultados em comparação com populares sistema de gestão de bases dados como o PostgreSQL [Muhleisen, 2014].

Esta tese de mestrado tem o objetivo de criar uma extensão de streaming no MonetDB com focus na IoT. A Amsterdam Internet-of-Things App (AIoTA) é uma aplicação *full-stack* com o objetivo de ser integrada facilmente com dispositivos IoT para colecionar dados para *streams*, tendo ao mesmo tempo como vantagem o armazenamento colunar do MonetDB para processar os dados e disponibilizar resultados imediatamente.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 CONTEXTUALIZATION

The Internet-of-Things (IoT) allows objects to be sensed and controlled remotely across an existing network infrastructure, while creating opportunities to assimilate computer systems with the real world. With IoT, it is possible to change the way people interact with objects for a better quality of life. In 2011 there was about 25 billion devices connected to IoT, and by 2020 that number is expected to double [Evans, 2011]. The IoT's applicability is very wide, from health care and home monitoring, to agriculture and car vigilance. In this way the market of IoT is growing significantly as both consumers and businesses are getting benefits from connecting devices to the Internet [Greenough, 2014].

The expansion of IoT's connectivity has lead devices to exchange large amounts of data, due to constantly required monitoring. The output of these devices can be seen as streams with data made available incrementally over time. As a consequence, IoT accommodates the *Big Data* model which translates into data that needs to be processed efficiently from many sources simultaneously, while producing results promptly.

Big Data is often defined by the 4 V's: *Volume*, *Variety*, *Velocity* and *Veracity* [Han and Lu, 2014]. *Volume* represents the quantity of data, its size determines the value and potential of the data. *Variety* describes the possible types of data and the sources they can come from. *Velocity* indicates the speed that the data is being generated and the quickness of processing required to the demand. Finally *Veracity* represents the quality of the final results and how much efficiently we can process data without declining the quality. Due to these reasons, IoT and other stream-based applications such as social networks, lead to research focusing on large scale and high availability technologies for *Big Data*. This demand ranges from subjects such as machine learning and data mining, to scalability and data management.

Databases have been organizing collections of data for several decades. At a higher level, Database Management Systems (DBMSs) have researched how to organize data efficiently and have explored new ways to collect it simultaneously. In particular, Data Stream Management Systems (DSMSs) have emerged to research new methodologies on how to handle with uninterrupted flows of streaming data and integrate them with relational databases

[Aggarwal, 2007]. With this objective in mind, DSMSs have distinguished from traditional with new architectures, data models, algorithms and specific query languages to deal with streams. As streams are uninterrupted, DSMSs aim to process them incrementally. This lead to the continuous queries concept, where streaming data is processed with small batches each time.

Meanwhile, other DBMSs have explored alternate ways to organize data. MonetDB is a pioneer column-oriented relational database management system (RDBMS), storing relations column-wise opposed to rows as the majority of RDBMSs. Columnar-wise storage allows several benefits not capable row-wise such as column caching, data compression and late materialization of the results. MonetDB is being developed at Centrum Wiskunde & Informatica (CWI) in Amsterdam since 1993, takes full support of the ACID properties [CWI, 2015] and compiles the 2003 SQL standard. Through the years, MonetDB has researched new ways to build large scale databases in a more efficiently and scalable way, while taking advantage of column-wise storage [Idreos et al., 2012]. This research made MonetDB achieve faster TPC-H results than popular row-oriented RDBMSs such as PostgreSQL [Muhleisen, 2014].

1.2 PROBLEM STATEMENT

To accommodate the IoT interest raise in Amsterdam city in the last years, along with special attention to the recently proposed Amsterdam IoT network [de Vries, 2015], this report details the creation process of Amsterdam Internet-of-Things App (AIoTA), a new streaming application developed for MonetDB. This application has the objective to create a DSMS on MonetDB while focusing on IoT processing and fulfilling the *Big Data* 4 V's as much as possible. AIoTA is a full stack application aiming to be integrated easily with IoT devices to collect streaming data, while taking advantage of MonetDB's columnar-wise storage to process it and deliver results promptly.

1.3 OBJECTIVE

The main objective of this master thesis is to add a flexible streaming engine to AIoTA with focus on Internet-of-Things processing. For the IoT devices, two web servers will be created (one to input data and other for output data), hence the database layer will be completely transparent for them. As the DSMSs' implementation is quite divergent in the market, we will extend the MonetDB's kernel code with a simple but flexible streaming engine aiming to cover many of the existing solutions in the market.

1.4 STRUCTURE OF THE DOCUMENT

In Chapter 2, a research of related work in DSMSs is conducted. In this Chapter the evolution of DSMSs, their motivation and challenges are reported. At its end, some of the currently most popular DSMSs in the market are detailed, as they provided inspiration for this work. Chapter 3 gives an overview of MonetDB, with details of its architecture, query processing and its internal language, MAL.

AIoTA's architecture design and justification is documented in Chapter 4. Also in this Chapter, AIoTA's components are detailed, as well their communication process. The following three Chapters detail the implementation of the whole AIoTA platform. Chapter 5 reports the *IoT Web Server* development. These web servers aims to integrate IoT devices easily with the new streaming engine. Chapter 6 details the development of *Web API Server* server which outputs data produced by the streaming engine to be integrated easily with IoT monitoring devices. Chapter 7 details the development conducted over MonetDB. A streaming extension for MonetDB has been developed under the "iot" schema. This Chapter will detail the proposed streaming scheduler, the new SQL catalog under this schema, as well the explanation of execution plans of continuous queries.

AIoTA will be evaluated at Chapter 8. The implementation choices in AIoTA will be evaluated over the current state of the art as a functional evaluation. Later on the entire architecture will be tested on an Internet-of-Things scenario proposed during the internship with benchmarking against PipelineDB, one of the most popular DSMS in current the market as a performance evaluation.

2

RELATED WORK

2.1 MOTIVATION FOR STREAMING SYSTEMS

Sensors and monitoring devices demanding is increasing, mostly due to the Internet-of-Things. Some notable examples are weather observation, health, environmental monitoring, and object tracking. Most of the data is produced at high frequency and unbounded, resulting in data streams. A stream can be defined as an ordered sequence of immutable tuples (also called instances) to be processed only once or a small number of times using limited computing resources.

In many monitoring applications, data from multiple sources must be analyzed simultaneously, leading to scalability and load balancing problems. Other applications must relate freshly arrived data with historical data. Finally, the large amount of data might cause high spatial or temporal complexity, thus pushing these systems to handle smaller batches each time.

Data Stream Management Systems (DSMSs) have been developed to address these challenges. Unlike traditional database management systems, DSMSs have to deal with the constantly changing data and deliver the results immediately [Stonebraker et al., 2005]. Generally speaking, the data in DSMSs can't be processed and then forgotten. The system has to react to the changes with recalculation of the stored results, leading to a more severe query processing [Babu and Widom, 2001].

Traditional DBMSs has been developed since the 70's to answer some of these issues. However the DSMSs have a very distinct architecture as seen in the Figure 1. Just bellow that figure, Table 1 shows some of the most notable environmental differences that a DBMS and a DSMS have to face with.

In comparison, DBMSs will prevail on situations with persistent relations with one-time queries. There is no relevant importance of time and order, a large storage and possibly more complex queries. On other hand, DSMSs will be more suitable on the previous depicted situations: transient relations under bounded main memory with notion of time and order and unpredictable arrival of data.

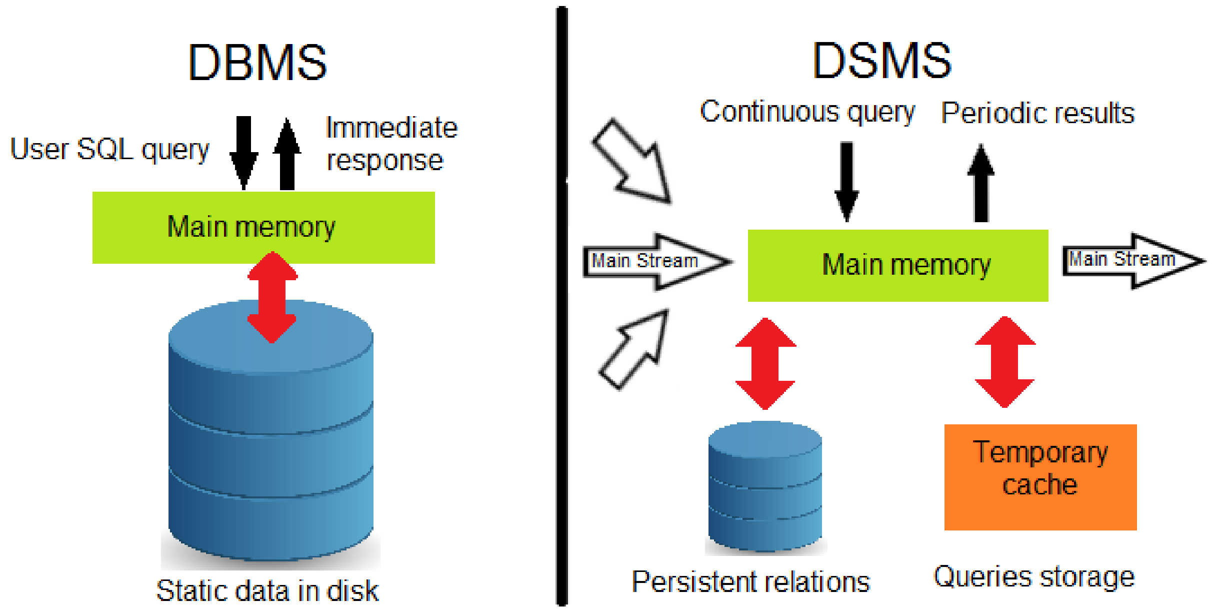


Figure 1.: Query processing comparison between DBMSs and DSMSs. In DBMSs, the queries are performed on static data and answered immediately. However in DSMSs, queries are made over streams with eventual access to static storage. As queries may run for a very long time, a cache is needed to store the temporary results.

DBMS	**DSMS**
Persistent relations	Transient relations
One-time queries	Continuous queries
Arbitrary access	Sequential access (notion of order)
Storage is ”unbounded”	Storage is ”bounded”
No necessity to store state	State is relevant
No real-time processing	Real-time requirements
Data is precise	Unpredictable arrival of data

Table 1.: Comparison between a DBMS and a DSMS. In the general scenario, DSMSs behave in a more stochastic environment than DBMSs.

2.2 REAL LIFE APPLICATIONS OF STREAMING

The most notable applications of data streams are in the Internet-of-Things field. The City Pulse project, for example, aims to build a framework capable of processing large scale streams of social data in real time in major European cities [Obaid et al., 2012].

The wot.io^{TM}data service exchange is a marketplace of web applications that operates on data from connected devices to enable data aggregation, analysis, and an expansive range of value-added services for enterprise customers [wot.io, 2015]. The service has been extended to the SQLstream DSMS to allow to streamline the design and development of IoT applications, and ultimately improve marked adoption.

Using Esper, a Java open source DSMS, a recent study at University of Oslo analyzed streaming data from different medical sensors in real-time to recognize myocardial ischemia with an ECG sensor.

The demand of DSMSs is increasing from sensor networks, so there are frameworks for integration of both technologies [Abadi et al., 2004]. The data streams are widely used on analysis and statistics across many fields. The Gigascope is an example of a stream database for network applications including traffic analysis, intrusion detection and performance monitoring [Cranor et al., 2003].

2.3 PROGRAMMING MODELS

The programming model in a streaming platform is one of its most important features, as it determines its possible operations and limitations. At the same time, the model defines the system capabilities and its possible use cases. DSMSs have approached two distinctive programming models to process streaming data: *Stream Processing* and *Batch Processing* with both models featuring advantages and disadvantages [Shahrivari, 2014].

In *Stream Processing* also called *Native Streaming*, the one-at-a-time tuple processing is applied. Data is processed immediately upon arrival, and allows more expressiveness. The reason for this is that the stream its taking control of itself and thus simulates a real continuous flow of data. As tuples are processed upon arrival, the latency in these systems are smaller than on *Batch Processing*. The existence of state is easier on these systems due to the one-at-a-time tuple processing. On the other hand, these systems are harder to implement, have lower throughput than *Batch Processing* and fault-tolerance is harder to achieve due to the fact the system has to store and replicate data for every single tuple.

In *Batch Processing* several tuples are processed at once for a better throughput at a cost of a higher latency. The system's expressiveness is more reduced compared to *Stream Processing*. State management and some operations such joins and aggregations can become harder to implement as the system has to deal with batches of operations. On the other hand fault-tolerance is easier, just by sending batches to every worker node. These systems are applied in scenarios where a huge collection of data is processed at once. There is still the case of *Micro-Batching* where stream data is processed with much smaller batch sizes [Shahrivari, 2014]. The latency is still low and the windowing and state-full computations are easier due to the batch size.

The *Stream Processing* model was mostly used on the first DSMSs, as it tries to mimic the smoothness of a stream. However the *Batch Processing* model obtained more importance in recent years, due to fact that requirements like scalability and fault-tolerance became predominant in IT. To achieve these requirements, others have to be sacrificed such as the smoothness seen in *Stream Processing*. Nonetheless is important to note that we can build a *Batch Processing* system on top of a *Stream Processing*. However the reverse is more difficult to accomplish when dealing with time.

On meantime, processing high velocity data has brought to two major processing use cases: *Distributed Stream Processing* (DSP) also called *Event Stream Processing* (ESP) and *Complex Event Processing* (CEP) [Luckham, 2006]. DSP/ESP is a stateless and straight way of processing incoming data using continuous queries. Data streams are transformed through query operators (joins, aggregations, filters) according to a topology or sequence of instructions. Only the final state is persisted for later analysis. ESP tends handle high volume in real time with a scalable, highly available and fault tolerant architecture. Typical use case scenarios are analysis on-the-fly like in IoT. On other hand CEP, is stateful and batched processing model where state maintenance is always present, hence is better suited for transactional environments. CEP engines try to optimize discrete events in streams while using defined topologies. The state management becomes complex in these systems due to higher requirements in transactional environments. The output can be either persisted or feed to another system. Some examples of CEP are stock exchanges for quick investments, detecting crucial clinic changes from vital monitoring, and accessing political vote intentions.

This report will focus on DSP/ESP for their lower requirements compared to CEP. Building transactional environments for CEP, brings extra requirements which is out of scope for this report. At the same time DSP/ESP have better applicability for IoT, but many CEP applications can be rebuild with DSP/ESP [Chakravarthy and Jiang, 2009].

It is also important to note that not all DSMSs are built over databases. As an example, messages queues are also often used to implement a streaming engine. Using different models, will result in different approaches to streaming with respective advantages and limitations compared to others.

2.4 STREAMING QUERY LANGUAGES

Initially defining a SQL query over a stream could be approached on a simple way, just by replacing relations with streams. However when queries get complex with joining, aggregation and mixing relations with streams, it becomes necessary to extend the language for separation of concerns. Unlike the SQL standard in DBMSs, there is no standard for the query languages used on the DSMSs. For this reason, query languages vary considerably from system to system [Jain et al., 2008].

To accomplish the task of creating a specific query language for streams, a new set of models and algorithms should be implemented as well. For this reason, most DSMSs expose these models and algorithms on their query languages.

To process data incrementally, the windows concept became an essential component of most DSMSs. Windows are buffers of data able to store streams' tuples in memory. Windows operate according to fixed parameters such as the size and bounds of the window, being these parameters updated after each query call [Ghanem et al., 2007]. Therefore in most DSMSs Query Languages, windows are used implicitly to process data incrementally. The windows concept will be explained in detail in Section 2.5.

The query operators must also be revised to accommodate both streams and persistent relations [Law et al., 2004]. Therefore different types of operators have been built to address several possible scenarios. The overview of query operators is given on Section 2.6.

SensorBee is an open source stream processing engine dedicated to IoT. SensorBee query language was derived from CQL, the language from STREAM, one of the first DSMSs declarative languages integrating streams with persistent relations [Arasu et al., 2006]. Now taking a sample query of a shopping website using streams:

```
SELECT P.price * (1 + P.tax) AS TotalPrice
FROM Orders[RANGE 5 TUPLES] AS O, Products AS P
WHERE O.ProductID = P.ProductID;
```

"Products" is a table listing the information of the products with respective prices and taxes. "Orders" is a stream of incoming orders in the shop. `[RANGE 5 TUPLES]` specifies a 5 element sliding window, which means the query will be executed whenever 5 orders are made in the website.

However the main question is how we should interpret this query: Can a product's price change between query calls? How we know if we are returning a relation or a stream? If the current transaction rollbacks, should we put the last window orders back on the stream? In general, the streaming concept introduces several implementation issues: The aggregations on streams should be based on the current window, or the whole stream? It's feasible to join streams? How to trigger a query which uses two streams with different windowing parameters? Can we reuse the same stream tuples on multiple queries? All these questions must be answered by the DSMSs themselves at their implementation.

2.5 WINDOWS

The desirable feature in DSMSs is to approximate stream processing likewise persistent relations. A common approach is to partition the stream in windows. Each window is comprised by a finite bag of tuples and processed sequentially. Also a window has an implicit notion of order, which means the tuples belonging to it are ordered according to a specific attribute. The number of valid elements in an window determines its type, and varies through the window dimension unit, edge shift and progression step.

The dimension of a window can be measured in either two ways: a *time-based window* through a τ units of time, or a *tuple-based window* meaning x first elements in the window are valid for the query at the time, likewise a FIFO queue. For both models, the inclusion of a timestamp in tuples is important for ordering in the queue. However some systems have introduced windows with other ordering criteria. Aurora added *value-based windows*, in which the windowing attribute is any other field [Abadi et al., 2003]. Other systems added *predicate-windows* where the window is filled until when a tuple fails a pre-defined condition [Ghanem, 2006]. At Figure 2, window C is a *sliding time-based window* with temporal extent (number of timestamps at the window) $\omega = 2$ and progression step $\delta = 1$. Also window A is a *count-based sliding window* (a variant where the window always take the last n tuples, in this case 5).

Meanwhile the *edge shift* specifies the movement from the bounds of the streams. The shift can be either fixed or moving along with the stream. In *sliding windows*, both bounds move along with the data arrival. Some languages use the `SLIDE` parameter for this specification [Kajic, 2010] (*e.g.* `[RANGE 10 SECONDS SLIDE 5 SECONDS]` Once 10 seconds have passed, the window moves forward by 5 seconds). In *landmark windows* only one of the bounds moves, meanwhile on the *tumbling windows* there is an arbitrary progression step of both bounds. At Figure 2, window B shows a *landmark window* with the lower bound fixed at τ, resetting after every 6 new tuples. Window D shows a *tumbling window* of temporal extent $\omega = 1$ and progression step $\delta = 2$.

Finally the *progression step* defines the periodicity of windows movements. Like the measurement unit, the step can either be time or tuple based (move every 10 seconds or 15 new tuples). In this specification, tuples overlap in more than one window. Nonetheless in *tumbling windows*, there is a moment where all tuples in a window become invalid at the same time. It is important to note that distinct window specifications will be applicable to different specifications. For an aggregation operator, a *tumbling window* will be more advantageous; while for a join, a *sliding window* is preferable [Patroumpas and Sellis, 2006].

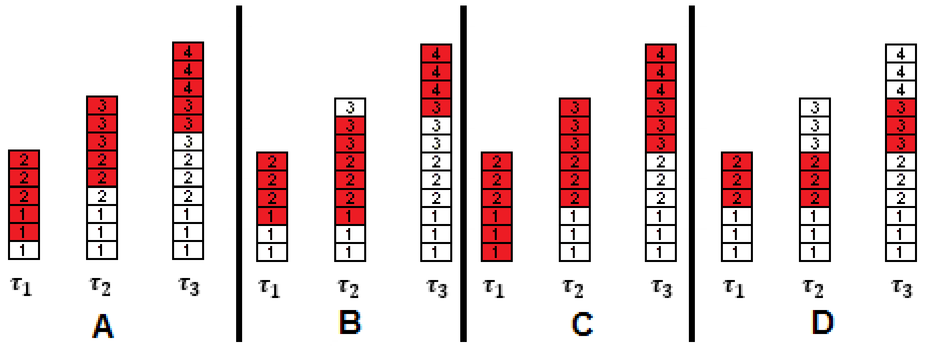


Figure 2.: Comparison of some windows implementations (A, B, C and D) with different edges shift and progression steps trough time instants τ_1, τ_2 and τ_3. New tuples arrivals are placed at the top. The number in the tuples represent their current timestamp. The window at instant τ_x is represented in red.

2.6 STREAMING OPERATORS AND CONTINUOUS QUERIES

Due to explicit constraints on streaming data, a revised set of operations for the query language is beneficial to DSMSs [Law et al., 2004]. In many applications, statically stored data is used with the streams to produce results. An example scenario is when historical data is updated with new incoming data. Therefore there is a necessity to join data from multiple streams and static data simultaneously. For this reason is necessary to re-implement the existing operators into *Stream-to-Relation*, *Relation-to-Stream* and *Stream-to-Stream* operators. Most DSMSs support *Stream-to-Relation* and *Relation-to-Stream* operators. *Stream-to-Stream* operators are more complicated to implement. For example in stream joining, the general approach for is to interpolate one of the streams as a persistent relation through sampling, turning to the situation of joining a stream with a persistent relation [Das et al., 2003].

In DBMSs *ad hoc queries* run through a set of operators which wait for all input before producing results. Due to continue flow of data in DSMSs, this implementation isn't feasible. To go over this, queries in DSMSs perform incrementally using *continuous queries* and reviewing a new set of operators.

For this reason, *continuous queries* have high importance in DSMSs, as it is not feasible to store the entire stream and then process it like in DBMSs [Babu and Widom, 2001]. Depending on external conditions, query statements might wait very long between arrivals of data in order to produce results. Varying from the operator's complexity the *continuous queries* might be harder to implement. For example, aggregation queries might require grouping changes with small computations, but join queries might produce an unbounded answer which is not feasible. To tackle this, *blocking query operators* were introduced so the queries are processed periodically instead at whole once.

At the same time, *non-blocking query operators* produce results sporadically, or on arrival of some tuples as it happens on some aggregation operators.[1] These operators are more suitable to continuous queries, however not all queries can be expressed with these type of operators [Law et al., 2004].

Another important distinction in DSMSs are *stateless operators* and *stateful operators*. The last ones (e.g. joins) require storing the intermediate state of their operations as the streams are unbounded. An immediate question is where and how to store the state between *continuous queries* calls while being fault-tolerant and scalable [Fernandez et al., 2013].

Now is important to note that this revision will vary considerably between *Stream processing* and *Batch processing* systems. The former ones prevail over *stateful* and *blocking query operators* to update each continuous query for each incoming tuple, while the latter ones prevail over *stateless* and *non-blocking query operators* for more efficiency.

Taking again SensorBee as an example, the following query outputs the join between a stream and a standard table in the database. It is expected to count the number of sells of all Amsterdam's merchants in the last 60 seconds. The `[RANGE 60 SECONDS]` is a *Stream-to-Relation* operator, responsible to convert the current window to a relation in each continuous query call. The `ISTREAM` is a *Relation-to-Stream* operator which creates a stream that only emits tuples present in the current window, but weren't so in the previous one. Therefore will be only outputted notifications of sells updates in the last 60 seconds.[2]

```
SELECT ISTREAM M.name, COUNT(*) AS total_count
FROM Sells [RANGE 60 SECONDS] AS S, Merchants AS M
WHERE M.merchantID = S.merchantID AND M.address LIKE '%Amsterdam%'
GROUP BY M.name;
```

2.7 TIME AND ORDER

2.7.1 *Timestamps*

In data streams, the notion of time is very important for ordering tuples in time based windows. The first question is how the values are timestamped. The general solution is to add an extra field to the tuples. The next question is to consider a logical or a physical way of timestamping. In the former each tuple is enumerated using a counter (e.g. a Lamport timestamp) but it serves just for ordering. In the later, the time information from the system is used (e.g. an UNIX timestamp) but systems differ in their timestamp interpretations. In many systems, *internal timestamps* are used whenever a new element arrives in the system

1 The aggregations can either be blocking or non-blocking depending if the data is sorted or not. More information on: http://sqlsunday.com/2014/06/15/blocking-aggregate-operators/

2 More information about these operators can be found at SensorBee's documentation: http://docs.sensorbee.io/en/latest/bql.html#relation-to-stream-operators

[Bai et al., 2006]. This guarantees that tuples are ordered by arrival time, and while they are pipelined through the system. In contrast, *external timestamps* are created by the external sources as an attribute, then ordered inside the system [Bai et al., 2006].

Some systems order the tuples for the same timestamp by arrival order. This is necessary to avoid semantic inconsistencies in some queries that can be provided from windows. At Figure 2, in *C* window, tuples of the same timestamp are present at same window. As an example, an unordered sequence could provide wrong results for the median value of a window.

Another relevant question is how to correctly assign timestamps after the results of *n*-ary operators. For aggregations, the result of a windowed minimum or maximum query could use the timestamp of the maximal or minimal tuple. In a count, sum or average the timestamp of the latest tuple could be kept. In a join a general solution is to timestamp the value of the tuple of the first table (in `FROM` clause), which can be used for *external* and *internal* timestamping models. A more generic attempt is to use the median value of the window.

2.7.2 *Order*

Generally speaking, many systems rely on the order of their elements for correctness. Also some operators become more efficient with the ordering of the input (e.g. usage of indexes). In DSMSs, ordering is easier due to sequential arrival of data. However this can be hard to achieve in systems with *external timestamps* from multiple sources. Since then, two solutions have been proposed for possible disordering problems.

The first solution is to tolerate disorder in stream's limits. In the Aurora system, tuples do not need to be ordered by timestamp [Abadi et al., 2003]. This imposition allows to split operators in *order-agnostic* and *order-sensitive* operators. The first group does not rely on order (filter, map and union) and therefore are executed efficiently. The second group (bsort, aggregate and join) have parameters on how unordered tuples should be handled. All other unordered tuples will be discarded through *load shedding* [Abadi et al., 2003].

The second solution is to indicate the order of tuples and reorder them whenever is necessary. Although the use of *internal timestamps* provides order, systems semantics with multiple sources require *external timestamps*. In some systems such as Gigascope, *Heartbeat* tuples are sent with the stream including a timestamp [Johnson et al., 2005]. These marks indicate that all following tuples require to have a greater timestamp than the mark itself. *Heartbeats* can be created by the sources or by the system itself.

2.8 QUERY OPTIMIZATIONS

The internal query execution is very similar to the one we find in DBMS, however the query optimization becomes very different in DSMS. In query optimization in DBMSs, we calculate several possible plans for a query, then we choose the least costly one using tables' cardinalities. This process differs in DSMSs because the cardinality calculation is problematic in a streaming environment. Also DBMSs storage statistics of data to help in optimization, but in DSMSs since the data of streams is unknown in advance, there are no such statistics. However it is possible to examine a data stream for a certain time to obtain a summary of the stream. Earlier DSMSs typically applied a plan migration strategy to replace a query plan with a new one at execution through time when the summary was obtained [Zhu et al., 2004].

The first common optimization technique is the *Rate-based optimization*, where rates of streams are taken in consideration in the query evaluation tree during the optimization process [Viglas and Naughton, 2002]. Instead of choosing the least costly plan, it is possible to decide for the plan with the highest tuple output rate. In advance it is required to derive expressions for the rate of each operator. As an example, we have a very fast selection operation on 100 tuples/second and a slower selection operation on 10 tuples/second. Both operations have the same selectivity of 0.1 tuples/s and can be commuted. Supposing we have a stream input of 200 tuples/s, if the slower operation happens first, by the first operator we have a throughput of 1 tuple/s by the first operator, then 0.1 tuples/s after the second. However if the faster operation occurs first we have a throughput of 10 tuples/s by the first operator, then 1 tuple/s after the second (about 10 times faster).

Often data streams obtain abnormal high inputs of data than usual producing longer queues of unprocessed elements. *Operator Scheduling* deals with tuple arrival rate and the operator path to answer against these situations [Babcock et al., 2004]. If the arrival rate of the tuples is uniform and lower than the system capacity, then there will be no problems in terms of scheduling. Whenever a tuples arrives at the system, we schedule it through all the operators in its operator path. In conclusion we refer this strategy as FIFO (*First In, First Out*), which is common through queueing systems. However we should note that uniformity in arrival is just a possible scenario, and hence we need more sophisticated scheduling strategies guaranteeing that the queue sizes do not exceed the memory threshold.

The *Chain Scheduling Algorithm* is used in DSMSs to manage queues and productivity of operators [Qian and Lu, 2010]. Suppose that each operator has a pair of selectivity and time to process (σ, τ). Later is possible to order them based on these two values. Operators with lesser time to process and more selectivity will have higher priority scheduling them first. A chart as seen on Figure 3 is built based on these pairs using three random operators with different times to process and selectiveness. The input is buffered between operators as shown in the *Greedy algorithm*.

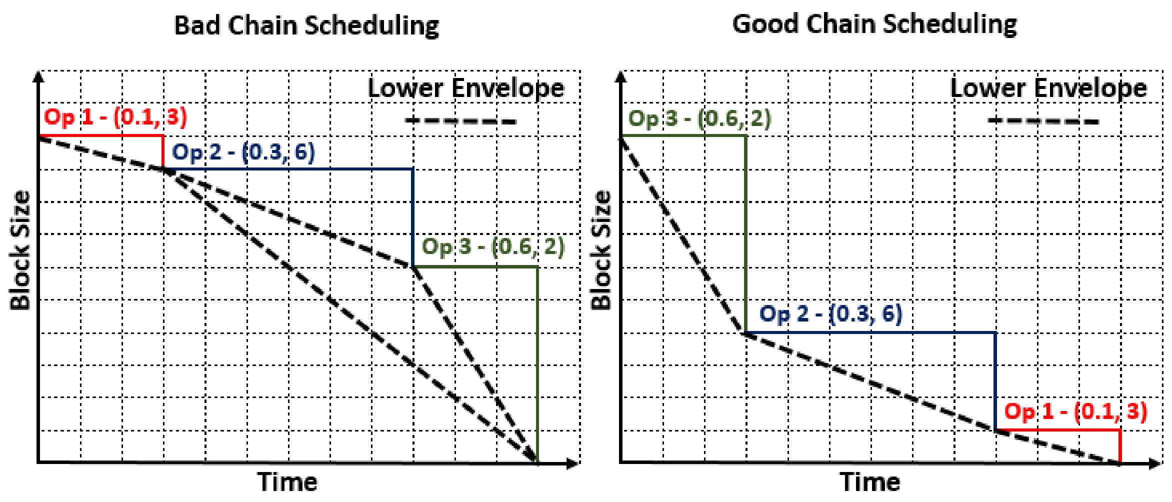


Figure 3.: Arrangements for operators with selectivity and time to process (σ, τ) in *Chain scheduling* resulting in distinct *lower envelopes*.

Chain Scheduling results in a *lower envelope* which represents the best case of the *Greedy algorithm*. A proper ordering of the operators might result in a smaller *lower envelope*, and therefore smaller queues and less memory requirements. An *lower envelope* will be better if it has a lower total area. *Chain Scheduling* helps to minimize memory usage, but CPU may be the bottleneck. Taking time in consideration, approximate answers are often more useful than delayed exact answers. So whenever an input stream rate exceeds the system's capacity, a stream manager can drop tuples. Now the goal is to minimize inaccuracy in answers while keeping up with the data.

The main concern is how often we should drop tuples, and thus becomes a challenging problem for the quality of results. The *load shedding* addresses this issue. A study calculated an optimal formula to estimate the error rate on *sliding window aggregate queries* with sub-queries on STREAM project [Mayur et al., 2003]. The algorithm has two steps: accumulate sampling rates for the queries (average and variance), then calculate a probability p for every tuple that will be used to discard it or not according to the equation. When there is only one operator in the query, it's straightforward, however, when some operators are shared by multiple queries, the situation becomes more complicated, since distinct queries require higher or lower sampling of tuples to produce the same maximum relative error estimate. Also there is a compromise of using *load shedders* earlier or late on a query plan (efficiency vs. accuracy) [Mayur et al., 2003].

Instead of optimizing the processing engine, another possible approach is to process less data for better latency and memory, thus achieving faster response in queries. *Synopsis structures* are commonly used to compress incoming data to achieve these requirements

[Gibbons and Matias, 1999]. These structures include making sampling on windows [Al-Kateb et al., 2007]; creating a compact synopsis of the data that has been observed through sketches [Matusevych et al., 2012]; creating a history of observations with histograms [Guha et al., 2006]; making hierarchical decomposition of incoming data with wavelets [Garofalakis, 2009].

2.9 STREAMING ENGINES OUTLINE

Having discussed a variety of challenges addressed and features offered by different DSMS, this section describes a set of representative systems and focuses on how they offer various design and implementation trade-offs.

The growth of demand for streaming data and distribute system behavior, brought more requirements for these systems. The *message delivery guarantees* is an imposing rule indicating that output and input tuples must be delivered eventually. At the same time, failures can happen anywhere: network down, disk failures, nodes going down for maintenance might compromise the system. *Fault tolerance* rules how well the system reacts to failures and recovers from them. As discussed on Section 2.6, *state management* has high importance of research in stream data. How the state is kept and updated is another important point of implementation in these systems.

Meanwhile the ”big” companies have introduced streaming engines in their products. MillWheel, developed by Google [Akidau et al., 2013], provides a fault tolerant directed computed graph in which the system manages persistent state and the continuous flow of records with low-latency. Trill, developed by Microsoft [Badrish Chandramouli, 2015], uses a tempo-relational model to handle arriving tuples with immediate results, while providing high performance. FlumeJava, also developed by Google [Chambers et al., 2010], uses the MapReduce programming model to handle high performing parallel pipelines that can be used on streaming data.

2.9.1 *Apache Storm*

Apache Storm is a distributed platform for Java focused on streaming data using the *Stream Processing* programming model with absence of windows.[3]. It has become an Apache Project in 2014 after being acquired by Twitter [Toshniwal et al., 2014].

Storm provides nodes to manipulate streaming tuples using three abstractions: *spouts*, *bolts*, and *topologies*. The *spouts* are the streams sources, where the user specifies how streams are generated. *Bolts* are responsible for the business logic of the streams. This logic includes filters, joins, aggregations and database interactions. *Spouts* and *bolts* are or-

3 Website: `https://storm.apache.org` GitHub repository: `https://github.com/apache/storm`

ganized into networks known as *topologies*. *Topologies* are Directed Acyclic Graphs (DAGs) with each node representing a *bolt* or a *spout*. Edges are represented by subscriptions to subsequent *bolts*, hence the topology is seen as the computation process of streams. After deployment, the topologies will run indefinitely until killed.

Storm's topologies are parallel by default, running in scalable and fault-tolerant clusters. The clusters are comprised of a master node daemon called "Nimbus", whose task is to distribute code instructions around the cluster, assign tasks and monitor failures. The workers run in a daemon called "Supervisor" who listens to tasks provided by Nimbus, for its own topology. A Zookeeper cluster is used to coordinate the communication between Nimbus and Supervisors [Toshniwal et al., 2014].

In recent years, Apache Storm suffered several scaling problems, and therefore the Twitter team developed a new streaming engine to face these issues: Apache Heron [Kulkarni et al., 2015].[4] Heron continuously examines the data in motion and computes analytics in real-time with better debugging and efficiency. It was open-sourced in May 2016.

2.9.2 *Apache Spark Streaming*

Apache Spark is a cluster computing framework influenced by Apache Hadoop's MapReduce programming model, having started in 2014.[5] Spark is notable for processing data in batches in its whole framework, answering today's scaling requirements. For this reason it has become one of the most active open source big data projects today [Harris, 2015].

Apache Spark contains several APIs within its framework for batched applications. These APIs include Spark SQL for data persistence; GraphX, a distributed graph processing library: MLlib (Machine Learning Library), a distributed machine learning library; and Spark Streaming, a library for stream data processing. Spark Streaming is a scalable, high throughput and fault-tolerant data stream processing system. While Hadoop's MapReduce is considered a case of *Batch Processing* [Shahrivari, 2014], Spark Streaming uses the *Micro-Batch Processing* model operating in smaller batch sizes compared to Hadoop.

The Spark's API is centered around Resilient Distributed Datasets (RDDs), a data structure of read-only data distributed over a cluster of machines on a DAG while fault-tolerant [Zaharia et al., 2010]. For this reason Spark internally slices data into batches to be distributed, hence its programming model. As a consequence, the results are also produced in batches.

Spark Streaming extends Spark Core's to perform streaming analytics. It takes data in mini-batches and operate RDD transformations on those batches at cost of higher latencies. The data can be provided through related open source libraries (e.g Apache Kafka) or barely

4 Website: `https://twitter.github.io/heron/` GitHub repository: `https://github.com/twitter/heron`
5 Website: `https://spark.apache.org/` GitHub repository: `https://github.com/apache/spark`

with a TCP connection. The streaming operators are provided through a functional interface with operations such as `map`, `reduce` and `join` [Zaharia et al., 2010]. These operations take RDDs as input and output RDDs as well for integration with other Spark APIs.

2.9.3 *PipelineDB*

PipelineDB is a stream extension of PostgreSQL open-sourced in 2015. This project will be more comparable to MonetDB's approach due to perform streaming with databases.[6]

The *continuous queries* concept is present through *continuous views*. The *continuous views* are materialized sets of results which are updated through time using time based windows. As soon as a tuple of a stream is read by the view, it will be processed and discarded, hence applying the *stream processing* model. This implementation gives predominance on the continuous views over streams. As a consequence if more than one stream is featured in a single continuous view, the windowing method applied will be the same for all the streams, hence this dominance.

It is also possible to integrate incoming streams with static data through continuous joins. Stream to table joins are performed whenever a tuple arrives, joining it with matching rows and updating the continuous view immediately. Note that if other matching rows in the persistent relation are inserted after the tuple arrival, the continuous view will not be updated as expected. The same happens whenever rows are updated or deleted.

To achieve fault-tolerance, PipelineDB supports streaming replication, using the passive replication approach. PipelineDB also has support for message queues through an integration with Apache Kafka.

However there are no tuple based windows, *stream-to-stream* joins are not supported, and due to usage of *continuous views* there is no native support to output stream data in physical devices. Nonetheless, this can be achieved by *continuous transforms*, which call user defined procedures whenever continuous views are updated, for producing output streams.

In the following example, the `page_views` stream collects URLs requests storing clients cookies and the latency of the request for benchmarking. Later the `page_stats` *continuous view* aggregates data from `page_views` in the last 10 minutes using a sliding window. For each URL it calculates the views count, unique visits using cookies and the 90^{th} value percentile of the latency.

```
CREATE STREAM page_views (url text, cookie text, latency integer);
INSERT INTO page_views (text, cookie, latency) VALUES ( ...... );

CREATE CONTINUOUS VIEW page_stats WITH (max_age = '10 minutes') AS
```

6 Website: https://www.pipelinedb.com GitHub repository: https://github.com/pipelinedb/pipelinedb

```
SELECT
  url::text,
  count(*) AS total_count,
  count(DISTINCT cookie::text) AS uniques,
  percentile_cont(0.9) WITHIN GROUP (ORDER BY latency::integer) AS p90_latency
FROM page_views GROUP BY url;
```

2.9.4 *DataCell*

DataCell was a DSMS extension of MonetDB, developed by Dr. Erietta Liarou between 2006 and 2012 [Liarou, 2013]. DataCell was positioned in the *Back-End* layer of MonetDB (Section 3.3), and was able to process continuous queries. After the creation of a continuous query, its optimizer generated the respective execution plan and handed it over to a scheduler who would be responsible to control the plan's life-cycle [Liarou et al., 2013]. To handle input and output, DataCell enclosed a set of *receptors* and *emitters*, which took advantage of MonetDB's column-based architecture to store data [Liarou et al., 2013].

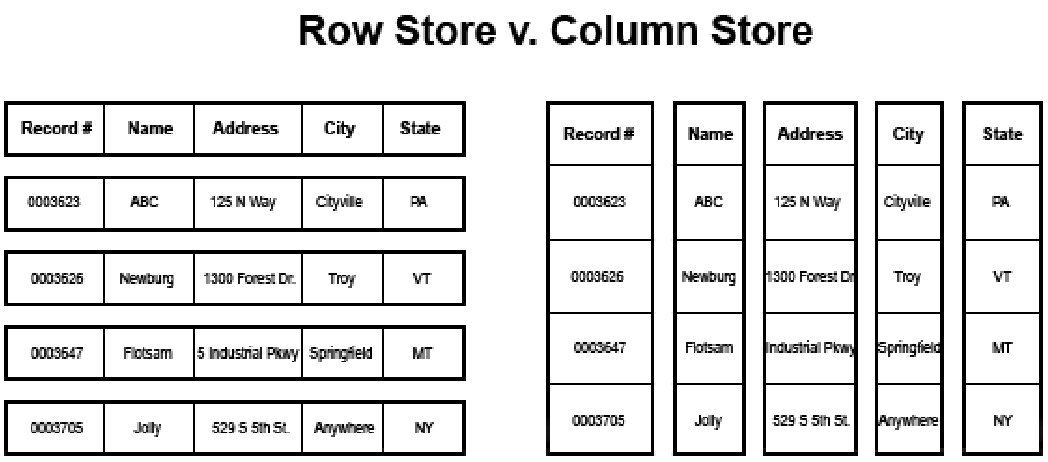
3

MONETDB OVERVIEW

3.1 COLUMN-WISE STORAGE

MonetDB is a RDBMS with a column-wise storage, opposed the traditional RDBMSs which use a row-wise storage. MonetDB was initially designed for Business Intelligence research often comprised by large data warehouses. These applications integrate large databases where an efficient access of data is required. The same scenario happens in e-science field where large bulks of data are inserted simultaneously for research purposes [Idreos et al., 2007].

Through the years, MonetDB has researched new ways to build these databases more efficiently and scalable, while taking advantage of column-wise storage.



Record #	Name	Address	City	State
0003623	ABC	125 N Way	Cityville	PA
0003626	Newburg	1300 Forest Dr.	Troy	VT
0003647	Flotsam	5 Industrial Pkwy	Springfield	MT
0003705	Jolly	529 S 5th St.	Anywhere	NY

Figure 4.: Internal representation of row and columnar-wise storage. The columnar storage provides better cache usage for a single column access, as well it creates more opportunities to compress data. Image taken from: http://arxtecture.com/wp-content/uploads/2014/01/row-store-v-column-store.gif

3.2 INTERNAL REPRESENTATION

The storage model in MonetDB is also different from traditional DBMS due to its columnar store. MonetDB handles relations tuples in *Binary Association Tables* (BATs) [Idreos et al., 2012], meaning for a relation with k attributes, there will be k distinct BATs for that relation. A single BAT contains data corresponding to the data type of a column. A table is then represented by a collection of BATs. After loading BATs into memory, they are accessed as an ordinary C-array using *Object Identifiers* OIDs.[1] It is important to note that OIDs are never serialized, just calculated on execution time depending on the data type of the column, allowing efficient operations like `COUNT` aggregations with $\mathcal{O}(1)$ cost. For variable length data types such as strings and BLOBs, MonetDB builds a heap of these values. BATs of the same value, are loaded into the same heap entry allowing further compression.

MonetDB uses the operating system's memory mapped files functionality to load persistent data, meaning that the binary representation in memory and the disk is the same, thus avoiding conversions. In addition to this, MonetDB uses late tuple reconstruction, meaning that all the intermediate query results are stored column-wise [Idreos et al., 2012]. Resulting relations are built just before sending them to the user. This technique allows to exploit CPU caches as well vector-wise operators for a more efficient processing.

On the other hand, column-wise storage might expose performance issues during relations updates. As the relations tuples are stored across several sources, a write operation across several columns simultaneously can be I/O expensive. To alleviate this, MonetDB uses *delta structures* kept in memory [Héman et al., 2010]. For each column there is a insertion and a removal *delta structure* alongside. During a transaction, instead of writing new changes to the sources immediately, they are written to these structures in memory. After the transaction is committed, the *delta structures* are merged with the respective columns in the sources.

The BAT representation is manipulated by MonetDB's kernel through the *MonetDB Assembly Language* (MAL). In its core, a relational algebra operator corresponds to a MAL instruction, meaning that each BAT algebra operator translates into a single MAL instruction. The conjunction of MAL statements for a SQL query is called a MAL plan. MAL instructions are evaluated with an *operator-at-time* schedule, meaning that each operation is evaluated completely before executing the next one. Complex operations are broken into a sequence of BAT operations that each perform on a entire column of values, known as *bulk processing* [Idreos et al., 2012]. The *bulk processing* mechanism allows loops without function calls, creating high temporal locality which reduces cache misses. Also these loops are

1 The BAT term was initially used to define an attribute with both a head (OID) and a tail (value). However as of June 2016, heads were removed completely from BATs as they are calculated on execution time. Therefore the term BAT may lead to confusion meaning that there is a binary relation but it is not.

more susceptible to compiler optimizations such as loop pipelining, and CPU out-of-order speculation.

The following example shows the MAL instruction for a selection on a integer column B for values equal to V into the result set R, while showing the respective C equivalent code.[2] The result set is a collection of OIDs of selected rows. The value of *n* in for loop is calculated based on the data type of the column and its size beforehand.

```
R:bat[:oid] := select(B:bat[:int], V:int);

for (i = j = 0; i < n; i++) {
    if (B[i] == V) {
        R[j++] = i;
    }
}
```

3.3 QUERY PROCESSING

MonetDB's query processing scheme is performed through three software layers: *Front-End*, *Back-End* and *Kernel* [Idreos et al., 2007].

The *Front-End* layer is responsible to parse SQL queries and convert them to MAL execution plans. The processing scheme begins with the *SQL Parser* translating an input query into a specific *Syntax Tree*. On the next phase, the *SQL Compiler* has responsibility to process the generated *Syntax Tree* and translate it into a logical plan, then to the respective MAL execution plan with some optimizations. These optimizations are performed at execution time and aim to reduce the size of input to be handled on the *Back-End*. The applied optimizations vary from the properties of the columns such as the data type, ordering or existence of an index.

Further optimizations are performed at the *Back-End* layer through the *MAL Generator*. The *MAL Optimizer* is a sequence of modules to manipulate a given *MAL Program* into a more efficient one. Unlike in the *Front-End*, the optimization in *Back-End* is inspired by programming language optimization instead of SQL optimization [Idreos et al., 2007]. For instance there is a set of optimizers for parallel query plan generation [Milena Ivanova and Groffen, 2012], which is made possible with column-wise storage. The *mitosis optimizer* splits the relation attributes into smaller chunks based on the number of available CPU cores. The *mergetable optimizer* merges the partial query results after all sub-queries have performed. It is important to note that some instructions cannot be executed in parallel, so they are not optimized further in order to warranty the correctness of the results. These

2 Adapted from MonetDB official website: https://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture/ExecutionModel

instructions are called *blocking instructions*.[3] The *mergetable* waits for all parallel instructions to finish, packing the result columns together, before calling a *blocking instruction*.

The *Kernel* layer also called *Goblin Database Kernel* (GDK), is responsible to perform CRUD operations on BATs, as well handing over a highly optimized library of the binary relational operators. As a consequence of *bulk processing*, each relational operator has access to the input's properties at execution time, therefore they judge the implementation algorithm also at execution time. As an example, a join operator decides at runtime to execute a merge-join if the correspondent attributes are sorted, or a hash-join if that condition does not happen.

Figure 5 shows all MonetDB's functional components and the generated data structures during the processing of a SQL query.

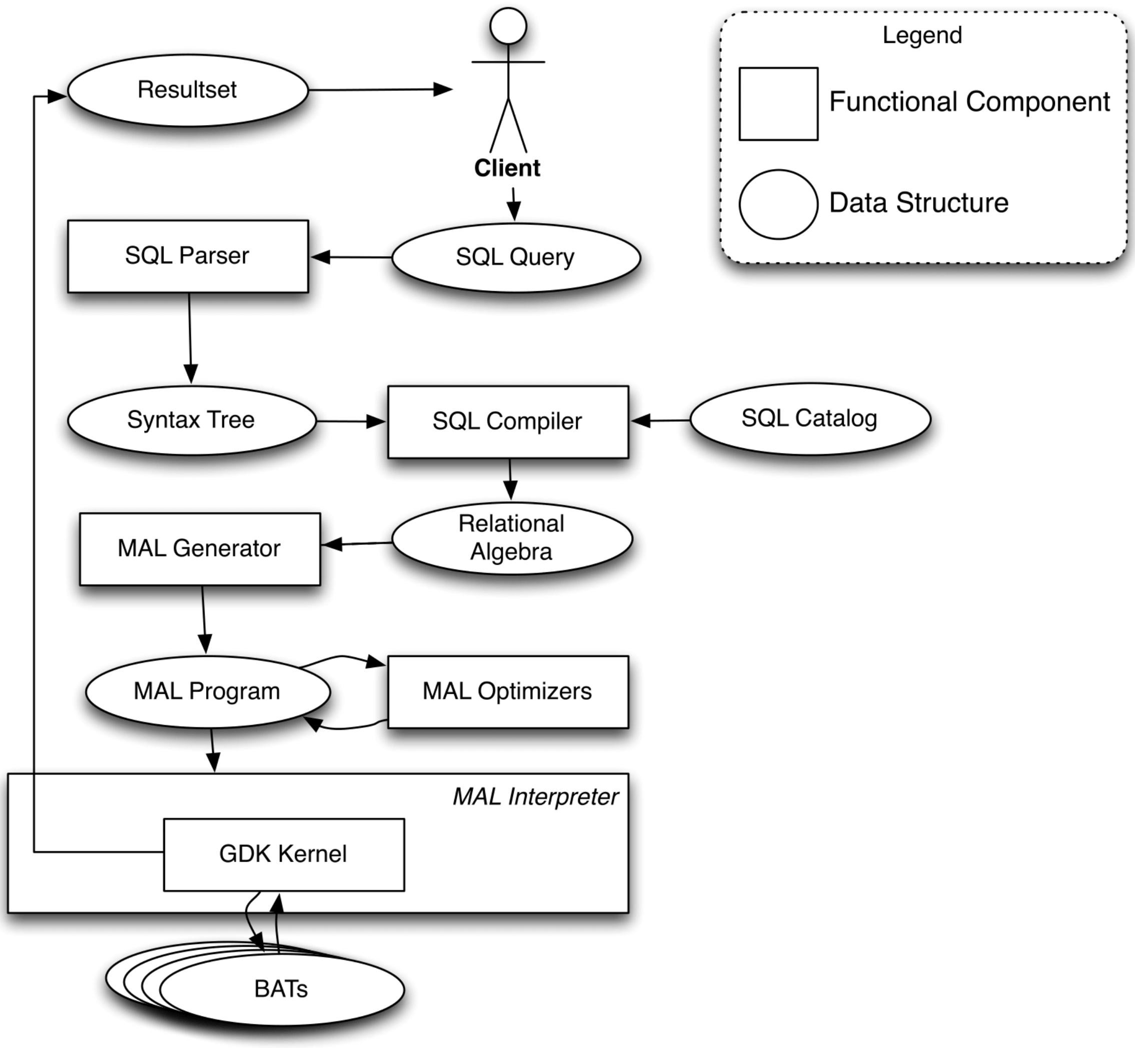


Figure 5.: MonetDB's components and relations during the execution of a SQL query.

3 Not to be confused with blocking operators in continuous queries (Section 2.4).

4 AIOTA PLATFORM

4.1 DESIGN CONSIDERATIONS

The Internet-of-Things growth, has led to increasing demands in streaming processing, hence building software solutions capable to answer the demand is notorious. At the same time, research in MonetDB focused on large databases through the years with focus on performance, which can be exploited to build a reliable streaming engine.

The proposed platform is called AIoTA (Amsterdam Internet-of-Things Application), to accommodate the IoT interest raise in Amsterdam city in the last years, with special attention to the recently proposed Amsterdam IoT network [de Vries, 2015]. The following list contains the main objectives for this platform:

1. Offer an interface for IoT devices, with low requirements and usability.
2. Add a flexible streaming extension to MonetDB's engine.
3. Assemble a topology vector-like to take advantage of MonetDB's columnar architecture.
4. Build an architecture easy to scale horizontally and vertically later on.
5. Provide an easy interface for monitoring and analyze the output, with a notification API.

Extending MonetDB with a continuous query processing capability is only one thing we need to do in the bigger picture. Next to that we need to build a full stack platform capable to collect, process and deliver streaming data in a transparent way to the IoT world. During this project a platform has been developed aiming to satisfy the previous stated requirements.

4.2 AIOTA ARCHITECTURE OVERVIEW

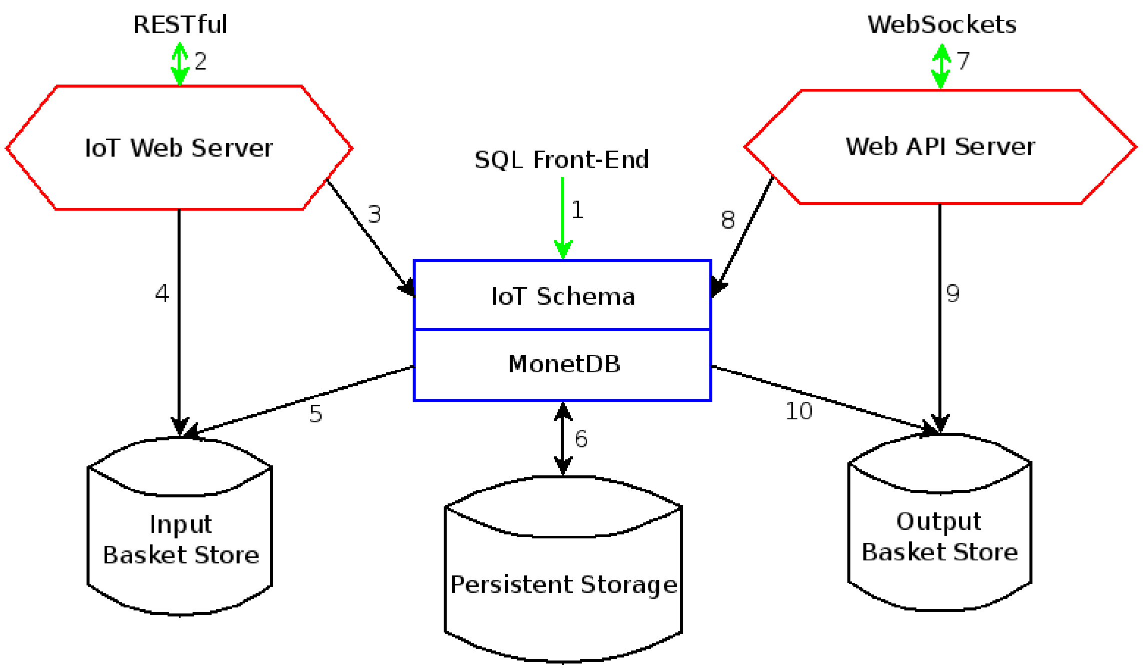


Figure 6.: AIoTA's proposed platform, components and relations.

The full AIoTA platform is depicted at Figure 6. AIoTA's topology exposes two web servers for IoT devices: *IoT Web Server* and *Web API Server* (red boxes), as well the regular SQL Front-End whenever is desirable to make a direct connection with MonetDB's engine. Meanwhile MonetDB's kernel code (blue box) is extended for AIoTA.

In the whole AIoTA's topology, is noticeable the usage of baskets as intermediate storage. Baskets are sets of binary representations of BATs. Since MonetDB's columns are stored using this representation, providing data immediately on the same representation allows a better performance while importing and exporting it using MonetDB's binary import feature.[1] Meanwhile, both web servers and the MonetDB instance run on distinct processes, hence the baskets act as the way of inter-process communication.

In the left side of Figure 6, (arrows 2, 3, 4 and 5) show the flow of *IoT Web Server*. This web server is responsible to collect and validate inputs from IoT sensor devices and translate them into streams' input using MonetDB's internal representation in the input baskets.

The generated baskets by the *IoT Web Server* are collected by the MonetDB's engine for processing. This engine includes a new streaming extension under the "iot" database schema (1). In this schema, a scheduler whichis responsible to manage the existing con-

1 More information about the binary import here: https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/BinaryBulkLoad

tinuous queries. The streaming data can also be assimilated with other database objects (e.g. tables, functions, procedures) on persistent storage (6).

The MonetDB's engine generates output data in the form of baskets for the *Web API Server*, with its flow shown in the right side of Figure 6 (arrows 7, 8, 9 and 10). This web server is responsible to translate the output the baskets to IoT monitoring devices.

4.3 AIOTA WORKFLOW

The *IoT Web Server* provides a RESTful interface, listening to HTTP JSON requests (2). The RESTful systems provide simple interfaces, approachable by IoT devices [Gruetter, 2012]. This server connects with the streaming extension (3) to create and delete streams and inserting data over its lifetime inside the *Input Basket Store* (4). The development of *IoT Web Server* is detailed in Chapter 5.

Incoming data stored inside the *Input Basket Store*, is processed by the streaming extension (5) inside MonetDB. It is important to note that the baskets contains volatile data, so once they have been consumed by the continuous query engine, they are deleted immediately as a recurrent behavior of DSMSs (Section 2.1). Continuous query results are stored on *Output Basket Store* by the new streaming engine on MonetDB. (10), later consumed by the *Web API Server* (9). Eventually the query results can be stored in regular MonetDB tables as persistent relations (6) instead in the output baskets. The development of the new streaming engine under the "iot" schema is detailed in Chapter 6.

The *Web API Server* looks up for existing streams in the streaming context (8). This server reads and converts data from *Output Basket Store*, while exposing it to a Web API using the WebSockets protocol (7). With WebSockets, it is possible to manage real time monitoring of events in IoT using a full-duplex connection. Therefore the web client can subscribe to be notified immediately whenever new output baskets are created. The development of *Web API Server* is detailed in Chapter 7.

This platform handles streaming data using a *Batch Processing* model (Section 2.3), taking advantage of MonetDB's vectorized architecture for processing in batches. At the same time, many IoT networks behave likewise Wireless sensor networks (WSNs) [Alcaraz et al., 2010]. WSNs are comprised by distributed autonomous sensors for monitoring, where data is sent cooperatively between them to a main collector for processing. This approach results in sending data in batches, hence is advantageous for AIoTA. AIoTA performs under a *Distributed Stream Processing* model (Section 2.3) as it provides more flexibility and lower requirements compared to CEP during the internship time.

5 IOT WEB SERVER IMPLEMENTATION

The *IoT Web Server* aims at an easy API for IoT, so it hides the underlying streaming and database layers for IoT sensors. The *IoT Web Server* is written in Python using the Flask-RESTful framework, a popular Python library for RESTful web servers.[1]

The *IoT Web Server* is capable of creating and deleting streams on AIoTA using a RESTful API. After a stream is created, it is possible to make batch inserts into it. Both stream creation and stream insertion are validated using an official JSON Schema.[2] In the current implementation, a batch of tuples will be inserted, if only all the tuples are valid. This approach will force further correction on a batch, meaning that a batch is correct if and only if all its tuples are also correct, as in transaction processing. As discussed in Section 2.7, the server adds an implicit timestamp column to perceive when the tuple was inserted. Inserted tuples are later imported to the corresponding streams in the streaming engine and consumed by the respective continuous queries.

5.1 RESTFUL SYSTEMS

The *REpresentational State Transfer* (REST) is an architectural style with a specific set of conventions and components to build web applications, where the focus is to specify the components roles, and interactions instead of concentrating on implementation details. The term was introduced in 2000 by Roy Fielding [Fielding, 2000].

Systems that employ the REST style are called RESTful. On RESTful systems such as the *IoT Web Server*, clients approach the web server using the HTTP protocol, and use the HTTP methods GET, POST, PUT and DELETE to discriminate their requests. RESTful systems expose their features using web resources identified by Uniform Resource Identifiers (URIs). For example, in *IoT Web Server*, `/stream/measures/temperature` is a URI to identify the stream ”temperature” in the schema ”measures”. To achieve the desired properties, REST defines several principles that should be followed by the RESTful systems:

1 Flask-RESTful at PyPI: `https://pypi.python.org/pypi/Flask-RESTful`
2 Over the JSON Schema: `https://spacetelescope.github.io/understanding-json-schema/` The version used is from the latest draft (4).

CLIENT-SERVER - Separation of concerns between clients and servers. Clients should not be concerned about how data is processed, thus achieving portability. At the same time, the server should not be concerned with the user state.

STATELESS - The client's state is never stored on the server. Each request by the client should contain the current state, so the client holds the state itself.

UNIFORM INTERFACE - Resources should be identified by URIs. The resources themselves are conceptually different from the representations that are returned to the client. As an example, a server might send response data in JSON, which is different from the database representation. Also each request should include all information required to process it due to the stateless property.

Due to its simplicity, REST is a popular architectural style to build web services serving as an API. As there is no standard for RESTful APIs, implementations may vary significantly, although most of them use the standards such as HTTP, JSON, and XML. The URIs are used to represent the name of the actions on the server, while the HTTP method represents the type (GET - read, POST and PUT - create/update, DELETE - delete).[3][4] The HTTP response code represents the result of an action (2xx - succeeded, 4xx - client's error, 5xx - server's error). Outputs are often formatted in JSON. JSON is an open-standard format using (key, value) pairs to represent data in a human readable way. This format has become popular in recent years due to its versatility for array-like structures, and the growth of the JavaScript programming language from which it was originated.

The *IoT Web Server* was built using the RESTful principles. Therefore producing JSON HTTP requests will be approachable from most IoT sensors [Gruetter, 2012]. A brief description of RESTful API can be found on Section 5.5, meanwhile the detailed implementation is listened on Appendix A.3.

5.2 IOT WEB SERVER BOOTSTRAP

For security reasons two web servers are deployed after start up, one for administration and the other for general usage labeled as the application server. The administration server is capable of creating and deleting streams, and hence is recommended to listen exclusively on *localhost*. The application server on the other hand, provides inserts for streams and should be listening to all network interfaces.

3 The GET method is a labeled as a *safe-method* because a GET request produces no side-effects, which means that it does not change data.

4 The difference between the usages of PUT and POST has to do with the fact that PUT is an idempotent method, i.e. that the same request will produce the same result on the server no matter how many times it is sent, unlike POST.

The communication with the database engine (step 3 in Figure 6) is done using a MAPI connection with the Python client *python-monetdb*.[5][6] Despite its slowness [Raasveldt, 2015], the *IoT Web Server* will use this connection to notify the streaming engine to import input baskets. Therefore the volume of data transferred through this connection will be minimal.

During the bootstrap of the *IoT Web Server*, the database credentials (name of the database, host, port and user) should be provided as arguments, while the user's password is requested during the server's bootstrap.

Later on if the server is scaled horizontally, it might be desirable to identify from which replica an inserted batch came from. For these situations, it is possible to add an extra column for every created stream to indicate the identification of the replica. The value of that column can be passed during the server's bootstrap. The full list of arguments is given on Appendix A.1.

5.3 IoT Web Server Life-Cycle

After a stream is created in the *IoT Web Server* and consequently on MonetDB, the corresponding baskets are also created. If there are existing baskets during the server reboot, they will be exported (flushed) to the streaming engine immediately and hence deleted. The following directory path is kept during *IoT Web Server*'s life-cycle:

`<web_server_root_directory>/baskets/<schema_name>/<stream_name>/<basket_id>`

Baskets are identified by an incremental sequence (e.g. 1, 2, 3...), with `<basket_id>` indicating the directory corresponding to the number in the sequence. Inside this directory, there is a binary file for each attribute of the stream alongside the implicit timestamp.

The *IoT Web Server* imports data to MonetDB using the `iot.import` procedure call (details on Section 7.2). Now the question is how to regulate the number of calls of that statement. For this reason, while creating a stream on the *IoT Web Server*, web client must specify the flushing method of the stream. The flushing can be time based, tuple based or automatic. If the automatic mode is selected, the basket will be flushed whenever a batch insert is made. If the stream was created via the SQL Front-End, it will use the automatic mode.

The created streams' information is stored in MonetDB database for later usage when the IoT Server restarts. Meanwhile the *IoT Web Server* also pools the database for new streams created in the SQL Front-End, updating its catalog right away. The details of streams creation in *IoT Web Server* can be found on the RESTful API examples on Appendix A.3.

5 MAPI (MonetDB API) is the MonetDB internal communication protocol.
6 Python-Monetdb at PyPI: `https://pypi.python.org/pypi/python-monetdb/`

5.4 DATA TYPES MAPPING

The IoT Web Server uses the binary import statement in MonetDB to efficiently add data from the streaming engine to the database. Therefore one of the main tasks of the *IoT Web Server* is to generate input baskets promptly, so it has to translate the incoming tuples to MonetDB's internal representation of the corresponding column's data type. This method saves an ASCII conversion and subsequent parsing, thus the performance improvement can be significant.

For every column it is possible to give a default value to be added by the *IoT Web Server*, whenever a new tuple misses it from the JSON input. The default value will be validated during the stream's creation. The columns can also be nullable likewise in SQL and thus missable from the insertions. However a column cannot be nullable and have a default value simultaneously.

The available data types in *IoT Web Server* include all the types that can be useful for IoT applications, which includes number, strings and timestamps. On other hand, it is unfeasible for IoT devices to send binary data (BLOBs), as it is typically large to fit in a single HTTP request without fragmentation.

For IoT usage, some extra types were added for further validations if necessary. These types are converted internally to an existing MonetDB data type, therefore they exist just for extra validation. Table 2 listens all supported data types by the *IoT Web Server*, with the correspondent MonetDB mapping for the added ones. More details of each data type can be found in Appendix A.2.

IoT Type	**MonetDB type**	**Description**
`text`, `string`, `clob`	`clob`	Unbounded String (Character Large OBject).
`char`, `varchar`	`char`	Bounded String.
`uuid`	`uuid`	Universally Unique Identifier (e.g.: 550e8400-e29b-41d4-a716-446655440000).
`mac`	`char(17)`	Media Access Control Address (e.g.: `12-34-56-78-9A-BC`).
`url`	`url`	Uniform Resource Locator (e.g.: `www.google.pt`).
`inet`	`inet`	IPv4 address (e.g.: `234.12.126.8`).
`inetsix`	`char(45)`	IPv6 address (e.g.: 2001:0db8:85a3:0000:0000:8a2e:0370:7334).
`regex`	`clob`	Strings validated against a provided regular expression.
`enum`	`char` (longest enum value length)	Strings validated over a defined array of values.
`tinyint`, `smallint`, `integer`, `bigint`	`tinyint`, `smallint`, `integer`, `bigint`	8, 16, 32 and 64 bit signed integers respectively.
`hugeint`	`hugeint`	128-bit integer (not available on some systems).
`real`	`real`	32-bit floating point number.
`float`, `double`	`double`	64-bit floating point number.
`decimal`, `numeric`	`decimal`	Floating point number with a specific precision and scale.
`boolean`	`boolean`	True or false value.
`date`	`date`	Regular date on format `YYYY-MM-DD` (e.g.: `2016-05-31`).
`time`	`time`	Regular time on format `HH:DD:SS.sss` with timezone or not (e.g.: `14:30:21.122`).
`timestamp`	`timestamp`	Regular timestamp on ISO 8601 format with timezone or not (e.g.: `2016-07-19T13:26:14+02:00`).
`interval`	`interval`	Interval of time.

Table 2.: *IoT Web Server* supported data types and correspondent MonetDB mappings.

5.5 RESTFUL API

All the features of *IoT Web Server* are delivered through a RESTful API. Table 3 shows the available REST resources, methods, in which of the servers is featured and a brief description.

In AIoTA's scenario, every created stream will be assigned with a unique table id. Also every new tuple will be created with a unique timestamp, therefore different requests will produce different results. For these reasons POST method was used instead of PUT. The details of the RESTful API, with examples can be found on Appendix A.3.

Resource	**Method**	**Server**	**Description**
`/context`	POST	Admin	Creates a new stream.
`/context`	DELETE	Admin	Deletes an existing stream.
`/streams`	GET	Both	Get a detailed listening of existing streams.
`/stream/<schema>/<stream>`	GET	Both	Get details of an existing stream.
`/stream/<schema>/<stream>`	POST	Application	Make a batch insert to an existing stream.

Table 3.: Available resources on *IoT Web Server*'s RESTful API.

6 WEB API SERVER IMPLEMENTATION

The *Web API Server* is responsible for delivering data from output binary baskets generated by the streaming engine to web clients through a specific Web API. To achieve this, the *Web API Server* employs a WebSocket server responsible to convert the binary data in output baskets and deliver it in JSON format. This server is written in Python using a lightweight WebSocket server library with full-duplex connections.[1][2]

The *Web API Server* creates a connection pool with the streaming engine, listening for every new output basket from it. The *Web API Server* stream's catalog will update right upon stream creation/deletion from the streaming engine likewise in *IoT Web Server*. A publisher/subscriber pattern has been added to the server, in which web clients can subscribe to be notified right away when an output basket is created. Output data can be read in batch sizes in order to allow pagination. "read" requests may include an offset and/or a limit of the number of tuples for this purpose.

6.1 WEBSOCKETS PROTOCOL

WebSockets is a protocol standardized by the IETF under RFC 6455 in 2011.[3][4] It allows a full-duplex communication channel over a single TCP connection, making possible real-time data transfer between a web server and its web clients. Note that WebSockets is an independent TCP-based protocol using port 80, and thus suitable for environments with non-web Internet connections using a firewall in the other ports.[5] The only relationship between WebSockets and HTTP is that the handshake phase is interpreted by HTTP servers as an upgrade request (101 code response).

1 Simple-websocket-server (not featured on PyPI repository) GitHub repository: https://github.com/dpallot/simple-websocket-server

2 In a full-duplex connection, the both sides of a connection are allowed to communicate, also simultaneously if desirable.

3 The Internet Engineering Task Force (IETF) is a community of network experts which delegates the evolution of the Internet including its protocols. These protocols are rectified under Request for Comments (RFC) publications.

4 RFC 6455: https://tools.ietf.org/html/rfc6455

5 Or port 443 when used with SSL/TLS.

This protocol was initially designed to be implemented in web browsers only, being supported by all the current major browsers, but it can be used by any client/server application. WebSockets begins with a handshake between the client and the server. Once the handshake is finished, the client and server can send data frames back and forth in full-duplex, binary or text wise.

6.2 WEB API SERVER BOOTSTRAP

The *Web API Server* starts in a similar way the *IoT Web Server* with analogous arguments. This server is listening on all network interfaces by default. The communication with the *Back-End* is carried out with a MAPI connection like the *IoT Web Server*. During the start up of the server the database credentials (name of the database, host, port and database user) should be provided as arguments, while the user's password is requested during the server's bootstrap. The full list of the possible arguments is detailed in Appendix B.1.

6.3 WEBSOCKETS API

A Websockets session is asynchronous, which means that a request/response message might be sent at any time during the session. For that reason, all messages are labeled with an identifier to detect a request to the server and its corresponding response to the client, while using a defined JSON schema likewise seen in the *IoT Web Server*.

All client requests must be handed over as text frames, with a JSON formatted string. A request must include a `request` field specifying the intended action to be performed in the server, followed by the other specific fields depending on the request. In the same way responses contain a `response` field to identify the type of the message. Table 4 lists the possible requests, while Table 5 lists the possible responses referencing the request they answer if so. All the details of each request and responses with examples is detailed on Appendix B.2. Note that to perform a `read` request, the web client does not have to be subscribed to the stream.

Identifier	Arguments	Description
`subscribe`	A stream's schema and name	Subscribe to new output baskets from the specified stream.
`unsubscribe`	A stream's schema and name	Unsubscribe to a previous subscribed stream.
`info`	A stream's schema and name (both optional)	Retrieve information about a stream. If the parameters are not provided, the response will contain data over all existing streams.
`read`	A stream's schema and name (required). Basket number, tuple limit and offset (opt)	Read output from a stream starting at the requested basket (by default its the first available).

Table 4.: Available *IoT Web API* WebSocket requests.

Identifier	Request	Description
`error`	none	Error message report.
`subscribed`	`subscribe`	Subscription confirmation.
`unsubscribed`	`unsubscribe`	Subscription removal confirmation.
`removed`	none	If a subscribed stream is removed on MonetDB, this message warns the subscribed web clients.
`notification`	none	Notification of a new output basket to subscribed web clients.
`read`	`read`	Response message to a "read" request, handing over the number of result tuples and data listening.
`info`	`info`	Response message to an "info" request for one stream only (Optional arguments provided).
`data`	`info`	Response message to an "info" request for all existing streams (No optional arguments provided).

Table 5.: Available *IoT Web API* WebSocket responses.

7 MONETDB STREAMING ENGINE IMPLEMENTATION

The MonetDB kernel code has been extended with a continuous query processing engine. For this task, a new set of MAL operators had to be created, along with a new database schema, a continuous query execution scheduler and several optimization policies. The streaming extension can be found on MonetDB's source repository on the *iot* directory inside the SQL backends on "iot" development branch: `https://dev.monetdb.org/hg/MonetDB/file/iot/sql/backends/monet5/iot`.

The streaming engine development includes adding an iot module into MonetDB to deal with streaming data. The implementation requires the creation of new MAL statements, the continuous queries concept and the scheduler to trigger them. The main objective during this project was to create a SQL catalog in order to implement a minimal streaming engine. At this time, a user has to make a combination of "iot" procedures and functions calls to create continuous queries and manage them. This approach allows more flexibility on the streaming engine for the users and it will be justified at the evaluation Chapter on Section 8.1.

Note that the integration with a transactional environment is very difficult for continuous queries. If a transaction rolls back, the read tuples have to be put back into the original streams with additional caution of possibly newly arrived tuples during the transaction. At this moment, the streaming engine has not been tested with a transactional environment or an OLTP related benchmark, therefore is not recommended for usage with transactions at the time of this report.

7.1 SCHEDULER

Before introducing the MonetDB continuous query scheduler, we first give a brief explanation on the mathematical model Petri-net, on which the scheduler is based.

7.1.1 *Petri-net model*

The Petri-net is a mathematical modeling language used to represent distributed systems [Murata, 1989]. A Petri-net is represented by a bipartite graph where its vertices can be decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. All acyclic graphs are bipartite. A cyclic graph is bipartite if all its cycles have even length [Asratian et al., 1998].

A Petri-net is composed by a set of *places*, *transitions* and *arcs*. *Places* (i.e. conditions) are represented by circles, while *transitions* (i.e. events) are represented by bars, forming the nodes of the graph. The *arcs* which establish the edges of the graph, illustrate the passage from a *transition* to a *place* or vice-versa, but never between nodes of the same type (hence the graph is bipartite). The *places* where *arcs* originate to a *transition* are called the *input places* of the *transition*, while the places where *arcs* end are called the *output places* of the *transition*.

Places may contain a concrete number of points called *tokens*. *Tokens* act as parts of a condition. A condition is fulfilled, if there are enough *tokens* on all *places*. When a *transition* is executed, all existent *tokens* are consumed in every *input place* and created in every *output place* in an atomic operation.

The execution of Petri-net is stochastic, which means that when several transitions are enabled simultaneously, any one of them may be triggered, unless an *execution policy* is implemented. With the stochastic approach, *tokens* might appear in any *place* of the net, and thus are suitable to characterize concurrent behavior of distributed systems. At a lower level, the Petri-net provides a clean representation of a finite state machine. For this reason, the Petri-net abstraction is implemented by the continuous queries scheduler on MonetDB.

Figure 7 shows a basic Petri-net representation. The *place* P1 currently has a token, and through *transition* T1, it can go to either P2 or P3. The total number of tokens between the incoming and outgoing *places* of a *transition* does not have to be equal on both sides. *Tokens* might be generated or consumed in *places* outside the *transitions*.

Note that the Petri-net model carries planning issues in some graphs. One such example is possible deadlocks, if data coming from one *input place* is required to be consumed by more than one *output place*. In this case, it is recommended to create an intermediary *place* that will multiply the incoming *tokens*, so they can be consumed by all of *output places* without locking the *transition* forever.

This nature is used to implement our streaming scheduler. A streaming table is a *place*, while a continuous query is a *transition*. The query triggering conditions (time or tuple based) are represented through *tokens*. Also the *transitions* can be used to implement *blocking operators*, as they require certain amounts of output to produce results.

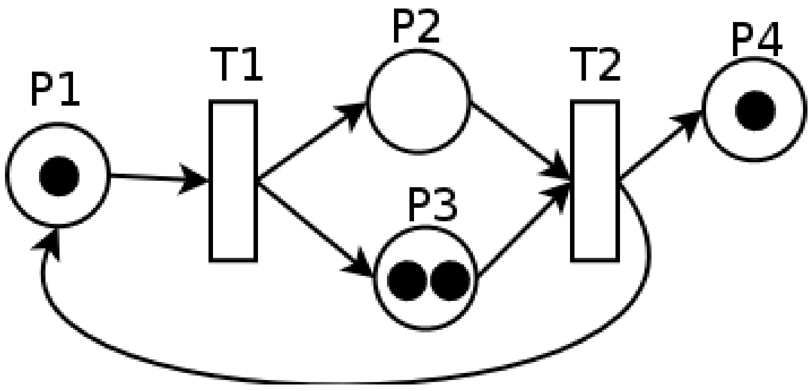


Figure 7.: A Petri-net graphical representation. Taken from URL: https://upload.wikimedia.org/wikipedia/commons/f/fe/Detailed_petri_net.png

7.1.2 *Continuous queries scheduler*

Unlike the conventional Petri-net, the MonetDB scheduler policy is deterministic. At the beginning of execution round, the scheduler determines all *transitions* eligible to trigger and activates one after another in the order they were registered. In a future implementation, this scheme may be replaced with a parallel implementation of the scheduler, in which each *transition* decides by itself when to fire. However, such parallel implementation may become expensive if there are enough complex continuous queries registered at the scheduler. Another possibility is to implement a priority queue, where higher priority continuous queries will be fired first, whenever it is possible.

The scheduler can be stopped or restarted at any time, the same happens for the registered continuous queries. The scheduler holds a table of the currently registered continuous queries, where looks it on every cycle. If the query meets the conditions to be fired then one thread will be created for its execution. After looking up the table, the scheduler will start all created threads and wait for all to finish. When all threads finish, the scheduler begins the next cycle.

7.1.3 *Concurrent continuous queries*

On every cycle, the scheduler looks up in the continuous queries table, processing them by the order they were registered. We allow more than one continuous query to be registered at the same time for one stream. However, to avoid complex implementation and potential performance problems, we have introduced a restriction into the current implementation. During the execution of each cycle. a stream can be consumed by maximally one continuous query. The first continuous query that can be fired gets exclusive locks on the stream it uses. So, what can happen is the following.

Taking an example of a stream that is queried by two continuous queries CQ1 and CQ2. Both are tuple based, and require 4 and 5 tuples to fire on the same streaming table (ST). CQ1 was registered first. In the next scheduler cycle, if there are 5 tuples, CQ1 will be fired and will lock the stream for the remainder of the cycle. Therefore CQ2 will not be able to execute because of the lock. Nonetheless, if CQ1 requires 5 tuples and CQ2 4, and there are 4 tuples available in the next cycle, CQ2 will be triggered because the available data do not meet CQ1's requirement. The same situation can happen in time based windows.

If a concurrent stream access is necessary, then currently the solution is to create a continuous query CQ3 that reads from input stream (ST3) into two new duplicated streams. In this way, each continuous query (CQ1 and CQ2) can process input from each of the new duplicated streams, avoiding any possible concurrency problem. Figure 8 shows the solution in a Petri-net representation.

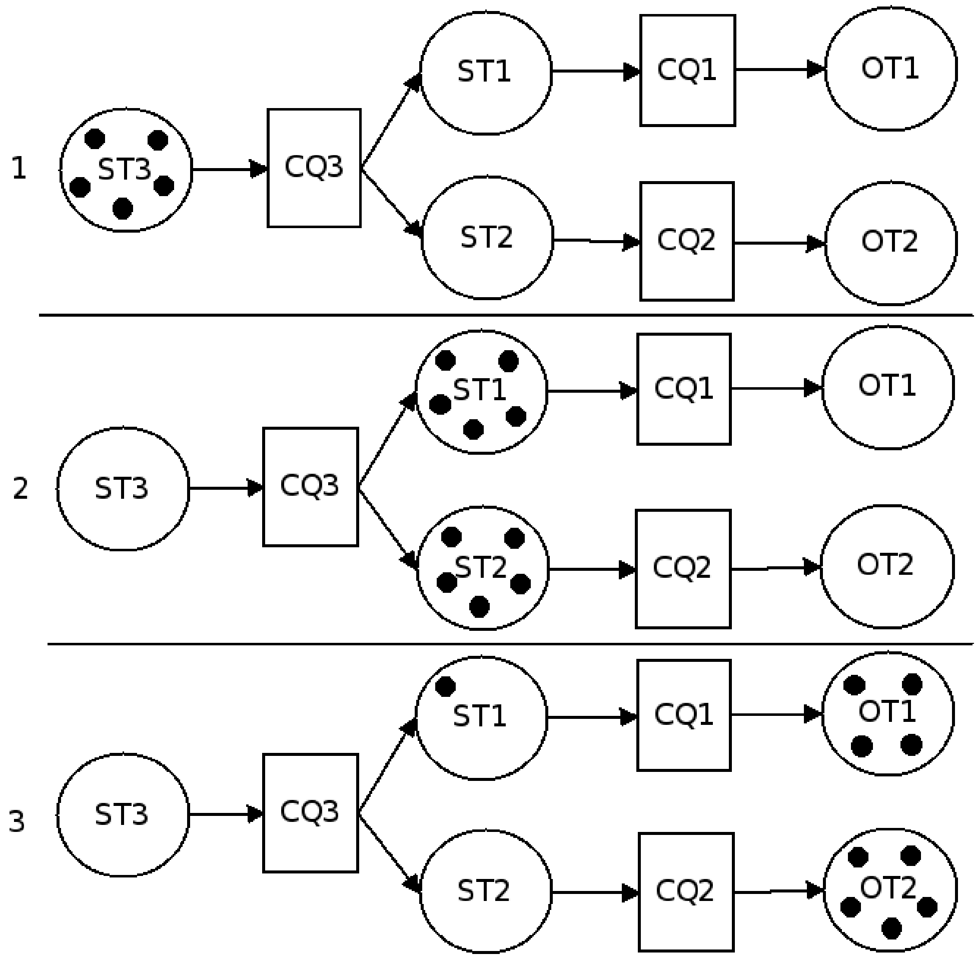


Figure 8.: The concurrency problem solution represented through a Petri-net. The state 1 is before the first scheduler cycle, 2 is after the first cycle and 3 after the second cycle. The black dots (*tokens*) represent inserted tuples in the stream tables. Acronyms: ST - Stream Table, CQ - Continuous Query and OT - Output Table.

7.2 SQL CATALOG

This section details all the functions and procedures created in the "iot" schema of MonetDB. The "iot" schema is divided into 4 categories: 1) scheduling to manage continuous queries by the scheduler, 2) windowing to create windows on stream tables, 3) baskets to import and export data from and to baskets, and 4) debugging procedures providing statistical information and debugging tools on the streaming objects. The following 4 subsections detail each of the categories. Finally, subsection 7.2.5 gives an example of creating a stream and registering a continuous query using the "iot" features.

7.2.1 *Scheduling procedures*

These procedures deal with registering and unregistering continuous queries at the scheduler, as well as pausing and resuming the scheduler itself. Table 6 lists the available scheduling procedures.

Name	**Arguments**	**Description**
`query`	schema: string, name: string	Registers a continuous query under a schema.
`query`	query: string	Registers a continuous query under the current schema.
`query`	schema: string, query: string, maxcalls: integer	Registers the named continuous query to be performed up to *maxcalls*.
`deregister`	schema: string, name: string	Unregisters the named continuous query from the given schema.
`pause`	schema: string, name: string	Pauses a registered continuous query.
`pause`	none	Pauses all currently registered continuous queries.
`resume`	schema: string, name: string	Resumes a paused continuous query.
`resume`	none	Resumes all paused continuous queries.
`cycles`	schema: string, query: string, n: integer	Registers the named continuous query only for the next *n* scheduler cycles.
`wait`	ms: integer	Pauses the scheduler for the next *ms* milliseconds after the end of current cycle.
`stop`	none	Stops the scheduler.

Table 6.: Available scheduling procedures in the SQL catalog.

7.2.2 *Windowing functions and procedures*

For windowing, there are 3 procedures which can be applied to the existing stream tables to indicate the triggering conditions for the corresponding continuous queries. The `iot.window` procedure is used to create a tuple based window. Its integer parameter, indicates how many tuples should be in the baskets in order to fire the continuous query again. The `iot.heartbeat` procedure is used to create a time based window instead, with an integer parameter specifying the number of milliseconds of the stride.[1] Finally the `iot.tumble` procedure takes an integer indicating how many tuples from the streaming table should be deleted, after they have been consumed by a continuous query. If a negative value is provided, then all the existing tuples in the window will be deleted after at each invocation, which is the default value. With combination of these procedures, it is possible to implement several types of windows, including sliding windows, landmark windows, tumbling windows, and both time and tuple based windows. Table 7 listens these just described procedures.

At the time of this report, the included timestamps by *IoT Web Server* are used just for indication of arrival. Therefore currently, there are no plans to implement windowing based on timestamps, as well timestamp updating during query processing as seen in Section 2.7.

Name	**Arguments**	**Description**
`tumble`	schema: string, stream: string, n: integer	Delete *n* tuples from the stream at the end of the continuous query.
`window`	schema: string, stream: string, n: integer	Read from the stream every *n* tuples insert.
`heartbeat`	schema: string, stream: string, ms: integer	Read from the stream every *ms* milliseconds.
`gettumble`	schema: string, stream: string	Returns the tumble value from a registered stream.
`getwindow`	schema: string, stream: string	Returns the tuple window value from a registered stream.
`getheartbeat`	schema: string, stream: string	Returns the time window value from a registered stream in milliseconds.

Table 7.: Available windowing procedures and functions in the SQL catalog.

1 This is not be confused with *hearbeats* used in some systems to keep order when receiving data from multiple sources in Section 2.7.

7.2.3 *Baskets procedures*

As shown in Figure 6, the stream inputs and outputs are temporarily stored in baskets. The following procedures are used to import/export data from/to baskets. Currently the *IoT Web Server* calls the `iot.import` procedure to export data into the streaming engine. Meanwhile the *Web API Server* only listens to file changes in the baskets directory. To use this procedure, the provided path as argument must have a file per column from the stream with the name of the column itself.

While the streaming engine is running, a receptor thread loops for changes in registered directories for automatic imports. New paths can be added using the `iot.receptor` procedure. At the same time, the emitter thread loops for changes in the streams exporting the results for provided directories. The `iot.emitter` procedure can be called to register new directories for output. Table 8 listens these procedures.

Name	**Arguments**	**Description**
`import`	schema: string, stream: string, path: string	Import data into a stream table from the provided path.
`export`	schema: string, stream: string, path: string	Export data from a stream table to the provided path.
`receptor`	schema: string, stream: string, path: string	Bind the provided path for file changes, importing to the specified stream table when files are created.
`emitter`	schema: string, stream: string, path: string	Whenever tuples are inserted on the specified stream table, results are exported to the provided path.

Table 8.: Available baskets procedures in the SQL catalog.

7.2.4 *Debugging functions and procedures*

For information and debugging purposes, several procedures and functions have been added. It is possible to get information about the registered continuous queries, inspect baskets and list the generated errors that have occurred during the execution of the CQs.

In general scenario, event processing through multiple layers of continuous queries is too fast to trace them one by one, thus the usage of these database objects should be used instead when debugging. Using these database objects it is possible to retrieve backlogged statistics about CQs such as the number of events handled per transition, average processing time of each continuous query, and the timestamp of the last invocation of a continuous query. This information can be used later on, e.g. when the scheduler policy needs to be re-designed. Table 9 listens these database objects.

Name	Arguments	Description
`show`	schema: string, query: string	Shows the MAL plan for a registered continuous query.
`baskets`	none	Returns a table inspecting data from the baskets.
`queries`	none	Returns a table inspecting data from the registered continuous queries including the timestamp of the last invocation.
`inputs`	none	Returns a table with the registered input places for the receptor thread.
`outputs`	none	Returns a table with the registered output places for the emitter thread.
`errors`	none	Returns a table with the generated MAL errors while performing continuous queries.

Table 9.: Available debugging procedures and functions in the SQL catalog.

7.2.5 *SQL catalog example*

In this example, a temperature streaming table is created and used in a continuous query to select the minimum, the count and the average of the temperature measurements collected on the last 5 seconds. At this moment, a user has to wrap the continuous query inside a procedure, as we have not extended MonetDB's SQL language support with features for continuous queries and streaming data. All streaming details must be handed over using the procedures in the "iot" schema, as shown in this example.

```
SET SCHEMA iot;
SET OPTIMIZER = 'iot_pipe';

CREATE STREAM TABLE temperature (t TIMESTAMP, sensor INT, val DECIMAL);
CREATE TABLE results (minimum DECIMAL, tuples INT, average DECIMAL);

CALL heartbeat('iot', 'temperature', 5000);
```

```
CREATE PROCEDURE examine_temperatures()
BEGIN
    INSERT INTO results
        SELECT MIN(val), COUNT(*), AVG(val) FROM temperature;
END;

CALL query('iot', 'examine_temperatures');
.................
SELECT * FROM results;
SELECT * FROM errors();
```

The `SET OPTIMIZER = 'iot_pipe';` statement will allow the "iot" optimizer to be used during the optimization process alongside other optimizers. The details of the optimization process as well the "iot" optimizer will be given on Section 7.4.

In this example, the streaming table "temperature" and table "results" are first created. Then we call the `iot.hearbeat` procedure indicating that a continuous query operating over the "temperature" table will be evaluated every 5 seconds. Then we create the "examine_temperatures" procedure where we insert the continuous query business logic, in this call calculating the aggregates into the "results" table. Then we call the `iot.query` procedure the register the continuous query into the scheduler. Later on we can inspect the results as well check for errors using the `iot.errors` function. The `iot.tumble` procedure is not called, hence all the tuples will be deleted from the window when accessed by the query.

If it is desirable to implement a specific type of window, then it is necessary to make a combination of calls of the windowing procedures. For instance, to implement a landmark window (see Figure 2 - B), resetting every N new tuples, the following procedure should be used:

```
--The stream table will be checked whenever there is a tuple in it
CALL window('iot', 'temperature', 1);

CREATE PROCEDURE examine_temperatures_landmark(N INTEGER)
BEGIN
    DECLARE current_landmark INT;
    SET current_landmark = (SELECT COUNT(*) FROM temperature);
    IF (current_landmark > N - 1) THEN
        CALL tumble('iot', 'temperature', N);
    ELSE
        CALL tumble('iot', 'temperature', 0);
    END IF;
    INSERT INTO results SELECT MIN(temp.val), COUNT(*), AVG(temp.val)
         FROM (SELECT val FROM temperature LIMIT N) AS temp;
END;
```

In the above procedure, we set the tumble value to *N* when the landmark value has been reached, otherwise we set it to 0 to delete none of the consumed data in ”temperature”.

The usage of `*STREAM` *Relation-to-Stream* operators depicted at the example in Section 2.6 cannot be performed using the current implementation as it is. Although we can create an auxiliary table to store the current result relation of a continuous query, and then update it incrementally, the results won’t be delivered immediately due to the scheduler policy. This behavior is characteristic from *Stream Processing* methodology which is not desirable for MonetDB. It requires each continuous query to run on a single thread during its lifetime, which would demand a complete re-implementation of the scheduler.

7.3 AGGREGATIONS ON CONTINUOUS QUERIES

As discussed in Section 2.6, due to their nature, continuous queries provide additional challenges to regular ones. Aggregations are a fundamental component of the SQL language for many use cases including statistics. In regular queries, they perform by analyzing all the fetched data and aggregating them into the final result. However, in continuous queries environment, the data is not available in its entirety, so it has to be processed incrementally. For this limitation there are two major possible solutions.

The first solution is to calculate the aggregation every time using only the tuples present in the current window. Many DSMSs, such as PipelineDB, use this approach, taking into consideration that this solution might not be desirable. The second solution is to store the intermediate results and update them in each call. Depending on the aggregation, different values must be stored and updated. For a `SUM` we must store the total sum of the arrived values, while for an `AVG` we store the total amount of tuples, for `MAX` the biggest value, etc. We must also take into consideration if we want to keep data in memory, or write it to the disks at the cost of a possibly slower query execution but with persistence.

At this moment, both solutions are possible in AIoTA due to the database usage. In the following example the continuous query updates a *tmp_aggr* table with data from the current window to calculate the average temperature. On this example we could avoid calculating the tuple count as we are using a tuple based window, therefore the count is implicit. However the example was made to be the most generic as possible.

```
SET SCHEMA iot;
SET OPTIMIZER = 'iot_pipe';
```

```
CREATE STREAM TABLE temperatures (t TIMESTAMP, sensor INT, val DECIMAL(8,2));
CALL window('iot', 'temperatures', 10); /* evaluate every new 10 tuples */

CREATE TABLE tmp_aggr (tmp_total DECIMAL(8,2), tmp_count DECIMAL(8,2));
INSERT INTO tmp_aggr VALUES (NULL, NULL); /* init the aggregation with NULL */

CREATE VIEW tmp_avg AS
    SELECT CASE WHEN tmp_count IS NULL THEN NULL /* avoid 0 division */
        ELSE tmp_total / tmp_count END AS average FROM tmp_aggr;

CREATE PROCEDURE temperature_aggregator()
BEGIN
    UPDATE tmp_aggr
    SET tmp_total = tmp_total + (SELECT SUM(val) FROM temperatures),
        tmp_count = tmp_count + (SELECT COUNT(*) FROM temperatures);
END;

CALL query('iot', 'temperature_aggregator');
....................
SELECT average FROM tmp_avg;
```

7.4 CONTINUOUS QUERY PLANS AND OPTIMIZATIONS

The query execution plan is a fundamental component in a DBMS, as it dictates the ordered sequence of steps to access and manipulate data in a SQL query. Note that when a query is submitted to the database, the optimizers look up on the query plan, possibly changing it, as well its order to find the best execution option. Also it is noticeable through consequent invocations, the plan execution order might be distinctive, due to runtime factors such as usage of cache.

For this report, it is proposed to make an examination over the query execution plans on streaming tables and compare them with regular tables to notify the optmizations induced on the streaming engine. As referred in Section 3.3, traditional DBMSs' query optimizers concern over the implementation algorithm of operators. However in MonetDB, query optimizations prevail over MAL statements, as the implementation algorithm is established by the query operators (e.g. join, scan, append) themselves at execution time. Therefore the optimizations discussed on this section will carry on replacement of MAL statements.

The resulting query plans can be shown through the `EXPLAIN SELECT ...` statement for queries on regular tables, and `iot.show` for their respective variant on streaming tables.

7.4.1 *New iot optimizer*

The optimization process in MonetDB is carried through several function calls to change a MAL plan. In this process, each function call belongs to a specific optimizer, thus modeling an optimization pipeline. The current execution optimization pipeline can be changed through the `SET OPTIMIZER = <pipeline_name>` statement.

For the new streaming extension a new pipeline, *iot_pipe* has been created, which includes the new *iot* optimizer. This optimizer is responsible for updating MAL statements on streaming tables. This optimizer will be always applied to the MAL plan whenever a streaming table is present.

7.4.2 *Query execution comparison*

For the comparison, the example from subsection 7.2.5 was recreated both in a regular and a streaming context. On each scenario, the respective optimizer pipeline (*default_pipe* on regular and *iot_pipe* on streaming) was applied.

Each of the optimization pipelines contains 21 optmizers. We will compare the final plan from the normal table against the one from the streaming table. The final MAL plans from both queries can be found in their entirety in Appendix C.1. In every comparison bellow, the regular table plan is listed first, followed by the streaming one. Some MAL statements only exist in one of the plans.

For a better understanding, a reading over the MAL reference on MonetDB website is recommended, as a full description of the MAL language is out of scope of this report.[2]

```
X_0 := sql.mvc();
C_1:bat[:oid] := sql.tid(X_0,"iot","temperature");
X_4:bat[:lng] := sql.bind(X_0,"iot","temperature","val",0);
---------------------------------------------------------------------------
X_0 := sql.mvc();
X_34 := basket.register(X_0,"iot","temperature",0);
X_38 := basket.lock(X_34,"iot","temperature");
C_1:bat[:oid] := basket.tid(X_0,"iot","temperature");
X_4:bat[:lng] := basket.bind(X_34,"iot","temperature","val");
```

2 MAL reference: https://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

The first major distinction in the plans is the data fetching process. For the streaming table, the `basket.register` call is responsible to assert that the basket is registered on the scheduler, being (the last argument declares the role as an input basket). Before applying any operation on the basket, the `basket.lock` call is responsible to lock it for the rest of the plan. This has to be done to avoid concurrency problems with the receptor thread and eventually concurrent SQL users. For the normal table, the `sql.tid` instruction calculates the tables' OIDs. The `sql.bind` statement is responsible to load the column. For streaming tables, the bind is retrieved from baskets instead of persistence storage.

The `sql.mvc` instruction is called beforehand to set up an isolated execution environment for the query.

```
X_19:bat[:timestamp] := sql.bind(X_16,"iot","temperature","t",0);
(C_21:bat[:oid],r1_22:bat[:timestamp]) :=
                        sql.bind(X_16,"iot","temperature","t",2);
X_23:bat[:timestamp] := sql.bind(X_16,"iot","temperature","t",1);
X_24 := sql.delta(X_19,C_21,r1_22,X_23);
X_25 := algebra.projection(C_1,X_24);
------------------------------------------------------------------------------
X_13:bat[:timestamp] := basket.bind(X_10,"iot","temperature","t");
```

Alongside the ″val″ column, ″t″ column must also be fetched for the plan. In this segment however, it is noticeable the most considerable divergence between the two plans, as multiple `sql.bind`, `sql.delta` and `algebra.projection` calls were removed in the streaming table's plan.

As pointed out in Section 3.2, changes to columns in MonetDB are not reflected to persistent storage immediately, but to *delta structures* alongside them in memory. As a consequence at plan execution, columns and the respective *delta structures* must first be merged together to form the correct data for each column. The `sql.bind` statement is called 3 times, with the last parameter indicating the structure to load: 0 - from the base table, 1 - from the inserts *delta structure* and 2 - from the deletes *delta structure*. After loading the structures, `sql.delta` merges them together. Finally the results are projected according to the previously loaded OIDs in `algebra.projection`. This process is repeated for all required columns in the plan. The MAL merge calls for the ″val″ column are not listed here due to the redundancy of the code, but can be found on Appendix C.1.

The streaming extension should be scalable and efficient, thus implementing *delta structures* on streaming tables would affect its performance. As depicted in Section 2.1, streams are labeled as a sequence of immutable tuples, hence applying updates on streaming tables is not feasible in practice. However in the current implementation of the streaming engine, it is possible to update (`basket.update`) and remove (`basket.delete`) tuples from streams.

```
X_39 := sql.append(X_29,"iot","results","average",X_36);
sql.affectedRows(X_39,1);
------------------------------------------------------------------------------
X_29 := sql.append(X_19,"iot","results","average",X_26);
X_39 := basket.tumble(X_29,"iot","temperature");
```

After appending the final result column to the "results" table, the `basket.tumble` statement removes tuples from the basket according to the table tumble value, reflecting the streaming behavior. For this reason, this instruction has to come at the very end of the plan. Meanwhile in the regular table plan, the `sql.affectedRows` instruction administrates the number of rows (one row in this example) to commit the transaction.

```
Not present
------------------------------------------------------------------------------
catch SQLexception:str;
    iot.error("user","examine_temperatures",SQLexception);
exit SQLexception:str;
catch MALexception:str;
    iot.error("user","examine_temperatures",MALexception);
exit MALexception:str;
basket.unlock(X_39,"iot","temperature");
```

As continuous queries are run by background threads, possible errors will not be visible to the users straightforward. Therefore whenever SQL or MAL exceptions occur at execution time, the `iot.error` instruction will persist the exception. The persisted errors can later be inspected using the `iot.errors` SQL function. At the very end, the `basket.unlock` call releases the basket lock, as it is no longer needed by the plan.

8

EVALUATION

8.1 FUNCTIONAL EVALUATION

In this Section AIoTA is rated according to the stream processing requirements outlined in a previous paper [Stonebraker et al., 2005]. In this same Section, our streaming engine's implementation is justified while making a reflection against the state-of-the-art.

8.1.1 *Stream processing requirements*

In [Stonebraker et al., 2005], the authors outlined eight requirements that a DSMS should accomplish, in order to achieve *stream processing* under real-time constraints. In this Section, AIoTA is evaluated for each of the eight requirements with a *yes*, *no* or *partially* rating.

1. *Keep the Data Moving - Partially*

The first requirement is to process tuples in a smooth way with low latency. This means that the system should not store incoming data before performing any operation. At the same time, the system should not approach an active processing model (e.g. polling) for more fluency.

As AIoTA uses a *batch processing* model, this requirement cannot be fulfilled in its entirety. Although the *IoT Web Server* can send batches of data automatically, and the *Web API Server* notifies new baskets immediately, the streaming engine on MonetDB introduces latency while processing data. In every execution cycle, the scheduler checks for updates in baskets to activate continuous queries, therefore the queries are not performed fluently. If a time window is defined at every 10 seconds, the scheduler will check if 10 seconds had passed, before activating a query again, instead of doing so at the exact moment. More importantly, tuples are processed in batches, and thus our implementation is not completely fluent for less latency. Nonetheless, AIoTA's implementation is fairly efficient as will be show from the performance evaluation at Section 8.2 and the data is processed fast, thus this requirement is partially fulfilled.

2. *Query using SQL on Streams - Yes*

The second requirement demands to provide a high-level query language to find output events and perform analytics. The SQL language is preferred to be used, because it has the concept of processing primitives such as filtering, projecting and aggregate. Furthermore, SQL is widely used and has a standard that make it easier to communicate between systems. The SQL language should be extended to accommodate queries on streams with windows and streaming operators.

AIoTA is built on top of a relational DBMS, hence the integration with SQL is implicit. The web servers were added just for IoT integration, therefore making a MAPI connection using SQL is also possible. Although the SQL language was not extended, it is still possible to inspect stream tables with standard SQL. The debugging functions and procedures in the ″iot″ schema can be used for detailed inspections on streams. Therefore this requirement is fulfilled.

3. *Handle Stream Imperfections (Delayed, Missing and Out-of-Order) - No*

Under real-time requirements, systems must react to possible real world imperfections, such as unordered, delayed and missing messages. Systems with *blocking operators* must additionally prevent possible permanent locks. In a real-time system, it is a bad practice to wait indefinitely, hence *time out* mechanisms should be applied in exceptional situations.

When dealing with streams, tuples might arrive out-of-order or not be delivered to the next operator in the pipeline. A DSMS should be able to overcome these issues. The *message delivery guarantees* property (Section 2.3) makes this possible, however depending on the underlying system, this property might be hard to implement. In DSMSs using message queues, this concept is implicit, but in DBMSs using transactions, it becomes difficult to implement with streams. Furthermore when dealing with more than one input source, the system has to order the tuples before processing them.

In AIoTA, this requirement must be fulfilled through the baskets, because they are the means of communication between components. In both the *IoT Web Server* and the streaming engine, locks are used to assure correct writing and reading of the baskets. However transactions are not implemented with streams, also there is not present a multiplexing functionality when dealing with *IoT Web Server* replicas simultaneously. Furthermore there are possible concurrency problems on the scheduler (Section 7.1.3) if the streams are not handled correctly. Finally the *batch processing* method provides additional latency, thus more delay of the results. This requirement is far from being accomplished.

4. *Generate Predictable Outcomes - No*

This requires that the system outputs expected outcomes during its life-cycle. The system should produce the same results no matter the tuples arriving in time or delayed. While dealing with regular tables, if a tuple arrives later and the table has changed, the result should change as well. This requirement is very important to achieve *fault tolerance*, in distributed systems when all the replicas should produce exactly the same output at the same time. However this requirement is hard to accomplish whenever operations are not idempotent.

In AIoTA this requirement is harder to fulfill due to batch processing. Hence this requirement is not fulfilled so far, due to the fact that the streaming is working outside a transactional environment.

5. *Integrate Stored and Streaming Data - Yes*

In many applications, streams need to be merged with persistent relations to produce results. It is a common task to compare the past with the present in monitoring applications, hence a DSMS should provide efficient ways to manage state information through time. In a DBMS this integration is easier as databases already deal efficiently with static data. The state can be stored persistently, or at execution time with *stateful operators*. In either way, the state management should be efficient to minimize latency. Another requirement is to be able to query stream data and persistent data in the same language, and this is achieved with SQL language in DBMSs.

As AIoTA is built on-top-of a relational DBMS, this requirement is fairly easy to fulfill.

6. *Guarantee Data Safety and Availability - No*

A high-availability (HA) solution is a critical requirement to preserve data integrity, requiring special attention in CEP systems. In real-time environments, the systems must be up at every moment, so the replacement of a replica must take minimal impact.

Although high-availability can be achieved in a clustered environment, AIoTA has no native high-availability implementation, thus this requirement is not fulfilled. MonetDB has the feature to create replica and merge tables which could be used to accomplish this, but it requires another overhaul of the database kernel for the streaming engine.

The input and output baskets need to be replicated as well. However as they are simply binary files, there are many possible solutions for this, e.g. `rsync` to copy a file to another computer system, RAID with redundancy disks, or using a distributed replicated storage system such as DRBD.[1][2]

1 RAID (Redundant Array of Independent Disks) is a storage technology that combines multiple physical disks into a single logical disk for data redundancy.

2 DRBD (Distributed Replicated Block Device) is a distributed replicated storage system for Linux.

7. Partition and Scale Applications Automatically - Partially

This requirement demands the DSMS to give the option to perform operations in a distributed fashion across a cluster of machines. The system should be load-balanced to avoid overload on a single machine. Also the distributed system should scale automatically and in a transparent way as reaction to abrupt arrival of tuples. The DSMSs should be multi-threaded as well to take advantage of multi-core CPUs.

In AIoTA, the *IoT Web Server* creates a thread per request, the *Web API Server* uses a WebSockets connection per client, while each continuous query on the *Back-End* runs on a separate thread in every cycle. However, AIoTA does not automatically scale out horizontally. This requirement is fulfilled partially.

8. Process and Respond Instantaneously - Partially

The last requirement, requires a DSMS to implement an optimized execution engine, to deliver results as soon as possible, even under unexpected increase of incoming data. This means that the system must still provide low latencies under these circumstances. To achieve this objective, the system's execution overhead must be minimal.

In the core of AIoTA, we rely on the highly efficient MonetDB engine to execute continuous queries. The "iot" scheduler, *IoT Web Server*, *Web API Server* only add a reasonably small amount of overhead. Altough our *Batch processing* model introduces some latency, our preliminary performance evaluation at Section 8.2 against a commercial DSMS, shows that AIoTA's current performance is promising. So this requirement is partially fulfilled.

8.1.2 *Comparison against the state of the art*

In Section 2.4 we have discussed the importance of extending the SQL language for streaming data processing. Some DSMSs attempted to create a standard [Jain et al., 2008], but it never came in practice due to highly divergent implementations. More importantly not all DSMSs currently use a DBMS as their base of implementation, therefore it makes standardization more difficult to accomplish. In our implementation, we added the `STREAM` keyword to the SQL language in order to identify streaming tables at their creation. Extending the SQL language for other streaming functionalities would require a review of the SQL Parser and Compiler of MonetDB (Figure 5). In this work, we decided to give higher priority to extend the MonetDB kernel with a CQ engine.

The distinction between streams and persistent relations has become relevant in DSMSs. For this reason, DSMSs generally implement different ways to accomplish this task by creating specific operators between streams and persistent relations (Section 2.6). The proposed standard [Jain et al., 2008], also seen in SensorBee, gives the perception of a stream with a temporary persistent relation, which is updated incrementally through time, while introducing the `RSTREAM`, `ISTREAM` and `DSTREAM` operators to update the output whenever the temporary persistent relation changes. Meanwhile, in AIoTA, we label streams as stream tables using *batch processing*. However the implementation of stream operators like `RSTREAM` wasn't possible due to our approach. We leave to the users to specify in the continuous query declaration and afterwards the desired behavior of streams and persistent relations. As continuous queries are defined in custom procedures, the degree of freedom is extensive. The depicted `*STREAM` operators can be approached using a table to store the temporary persistent relation. To produce either a stream or a relation, a continuous query should output its results into either a regular or a streaming table.

In Section 2.5, this report detailed several windows implementations for distinct use cases. Due to the divergence of implementations in DSMSs we leave to the user to decide what type of windows he wants to use (example in Section 7.2.5) on the continuous query specification, hence our implementation is simplistic.

Time has high importance in the implementation of a DSMS, with special attention when dealing with time-based windows. As mentioned on Section 2.7, several approaches have been taken into consideration about dealing with time. Due to large diversity of implementations, we only use *implicit timestamps* in *IoT Web Server* to indicate whenever a tuple has arrived at the MonetDB engine. All other timestamping methods have to be dealt with by the users themselves, even if timestamps should be updated through the MonetDB's engine. The same rule apply if order matters for a stream. Currently, time-based windows rely solely on the scheduler's time to be triggered, which can possibly leave to time gaps between timestamping on the *IoT Web Server* and the streaming engine.

In Section 2.8, this report detailed several possible optimizations related to streaming data and continuous query plans. In this work, none of these optimizations techniques have been considered, due to these optimizations requiring an overhaul of the current relational operators in MonetDB. However, it was possible to build a streaming engine over a relational database without the necessity to write the kernel code from scratch, as it can be seen from the MAL plan generation (Section 7.4.2).

Table 10 shows a functional comparison between AIoTA and the closest DSMS existing in the market: PipelineDB (introduced in Section 2.9.3), to evaluate what AIoTA has accomplished so far. Despite that, both systems are based on a relational DBMS, they choose different trade-offs.

Functionalities that are not available on both DSMSs, are not listed here, such as updated aggregate values (Section 7.3), and continuous Stream-to-Stream joins. The same happens for technical specifications such as client libraries, third-party integrations and possible configurations.

Feature	**AIoTA**	**PipelineDB**
Programming model	Pure *Batch processing*.	*Batch processing* built over *Stream processing*.
Available client interfaces	SQL and RESTfull using JSON through the *IoT Web Server*.	SQL.
Implementation basis for continuous queries	User-Defined Procedures, which are executed by a Petri-net model based scheduler.	Materialized views depicted as *continuous views* updated through time.
Window implementations	Windows are evaluated over streams. It is possible to have multiple streams with different windowing requirements in the same *continuous query*.	Windows are evaluated over *continuous views*. Hence all streams in the same *continuous view* will be evaluated at the same specified interval.
Available windows types	Both tuple and time based, allows sliding windows.	Only time based, allows sliding windows.
Continuous updates of query results	Not present, although is possible to perform continuous updates with auxiliary tables.	Present, *continuous views* are updated as tuples arrive (*Stream processing*).
Outputting Streams	Possible, write into a stream table in the continuous query definition.	Possible through *continuous transforms* although not as much flexible.
Continuous joins	All joins are possible although the results will reflect only in the current window.	Only Stream-Table joins. Cross and outer Stream-Table joins are not supported.
Timestamping	Implicit timestamps added by the *IoT Web Server*.	Implicit timestamps added at arrival by the engine.
Concurrent queries on streams	Duplicate or split one stream into multiple sub-streams, one for each continuous query.	Whenever a tuple arrives at the system, it is added to a bitmap representing all continuous views in which the tuple is used. After having been processed by a continuous view, the corresponding bit is flipped. After all bits are flipped, the tuple is discarded.
Synopsis structures	Not present.	Present - *Bloom Filters*, *Count-Min Sketches*, *HyperLogLogs*, between others available as data types.
High Availability	None by default, although MonetDB's replica tables functionality can be exploited.	None, although streams replication is present. Failures must be replaced manually.

Table 10.: Functional comparison between AIoTA and PipelineDB.

Table 10 shows that AIoTA has less features than PipelineDB. One reason is our chance to give users a simplistic approach to a streaming engine, due to time limit. Another significant distinction is the *batch processing* vs *stream processing* models on the two systems. In PipelineDB each batch of tuples is processed individually through the pipeline, giving the proposed smoothness of streams, however the windowing parameters are determined by the *continuous view* rather than the streams themselves, and only time windows are available. In AIoTA the windows are defined on individual streams. Nonetheless, due to *Batch processing*, it is not possible to provide the same smoothness as PipelineDB.

If it is intended to output the results in PipelineDB, *continuous transforms* must be used to detect changes in *continuous views* and output them. However *continuous transforms* do not store data, hence aggregations cannot be used on *continuous transforms*. On the other hand, in AIoTA, outputting results is easy to achieve in the continuous query definition. In the end, AIoTA's implementation gives more flexibility, while being easy to debug.

Meanwhile PipelineDB provides more streaming oriented features, such as synopsis structures and better concurrency control. Some of these features will be labeled as the future work for AIoTA in Section 9.2.

8.2 PERFORMANCE EVALUATION

In this section, we report our performance evaluation of AIoTA. In the first segment of this evaluation, the components of AIoTA are evaluated individually in profiling tests to check CPU usage during the most relevant operations. For the web servers, Flame Graphs will be used, while for the streaming engine, MonetDB's profiling tool Tomograph is used instead.

After the profiling tests, the full AIoTA stack will be tested performance-wise against PipelineDB using a real life IoT scenario related to ship tracking (AIS tests).

All tests in this section (Flame Graphs, Tomograph and AIS tests) were performed on a machine with an Intel Core i7-2600 CPU maximum clocked at 3.40GHz, 16 GB DDR3 RAM, 32GB SATA SSD disk, while running the Fedora 22 x86_x64 operating system (Linux kernel version 4.4.14). The MonetDB version used is the experimental "iot" branch.

8.2.1 *Flame Graphs on the web servers*

To study the performance of the web servers components we use Flame Graphs. Flame Graphs are charts resulting from analyzing programs' function stack-frames during their execution. These charts reflect all the function calls performed during a program's execution, as well as the duration of each stack-frame. Within these charts it is possible to detect the most frequent code-paths and identify the program's performance issues [Gregg, 2016]. There are several possible Flame Graphs based on the physical attributes to evaluate (e.g.

CPU, memory, Off-CPU (calculating time in CPU operations vs IO operations)). For this evaluation, only CPU Flame Graphs are studied as the performance is the main objective of AIoTA in this project.

These charts are represented through a two dimensional Cartesian coordinate system. The x-axis shows the present stack-frames (from the available threads) during the monitoring, hence the most important aspect is that the stack-frames are sorted alphabetically instead of chronologically as any reader will perceive on first glance. The y-axis depicts the stack depth. Each rectangle illustrates a stack-frame. As it is wider, the more time the CPU spent there. As it is higher, the more function calls were made until it. The rectangles bellow represent its ancestry (the function calls made until that rectangle). In these charts, colors are irrelevant as their only propose is to distinguish neighbor frames.

To generate Flame Graphs for the web servers of AIoTA, both servers code had to be changed briefly to accommodate a profiling thread to generate the charts, as well as new function calls. Flame Graphs were generated for the most relevant operations in the servers: creating a stream and making 1000 tuples batch insert on the *IoT Web Server* and performing 1000 tuples read from the *Web API Server*.

The charts shown here (Figures 9, 10 and 11) will only depict the requests related code, thus hiding the introduced Flame Graphs profiling code. The reason for this is that the full charts show all the created threads on the servers performing other actions (e.g. waiting for requests). More importantly, the requests functions stack-frames are very small compared to the whole chart, and is not possible to generate Flame Graphs in logarithmic scales with the current version.

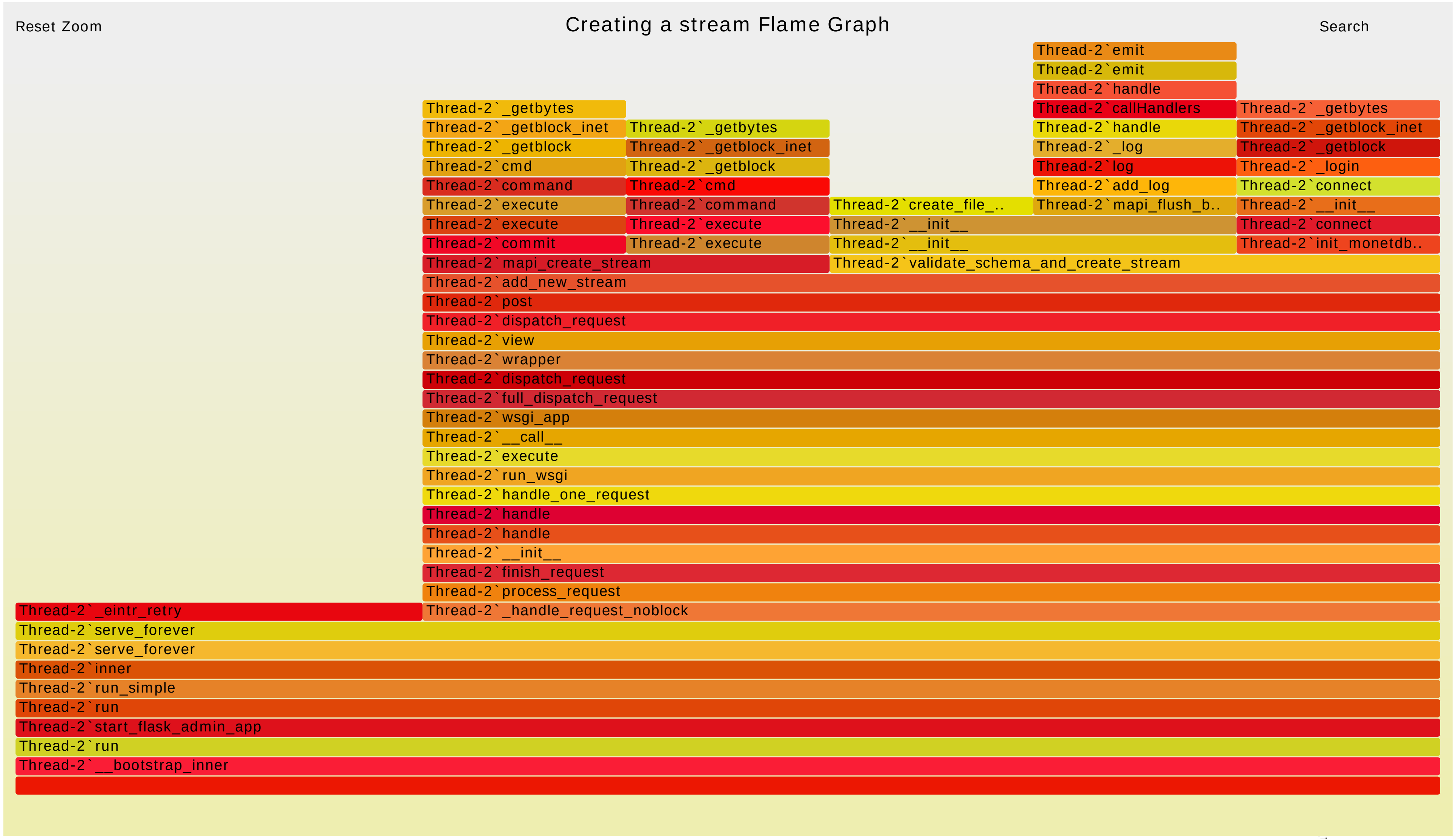


Figure 9.: Flame Graph corresponding to creating a stream at *IoT Web Server*.

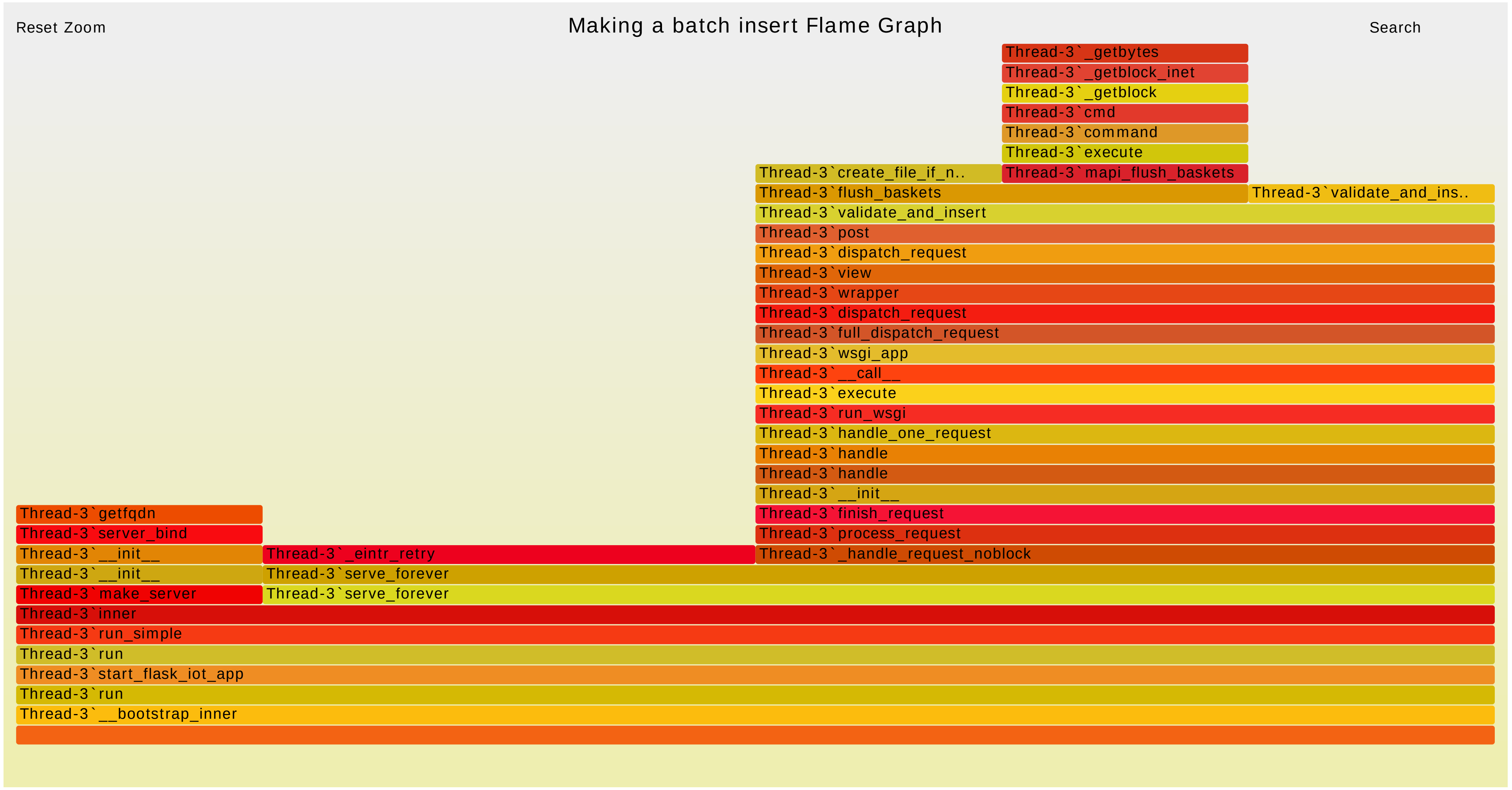


Figure 10.: Flame Graph corresponding to make a 1000 tuple batch insert at *IoT Web Server*.

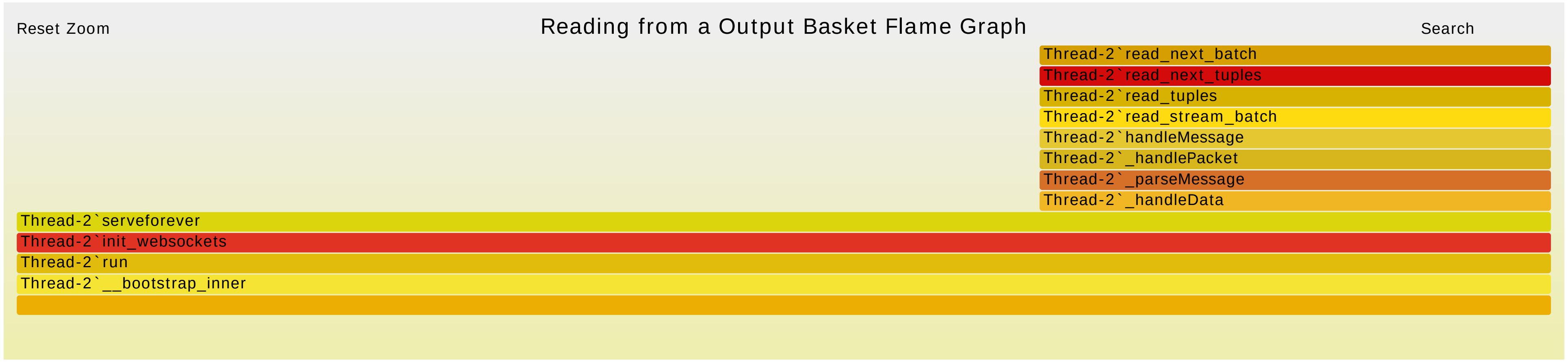


Figure 11.: Flame Graph corresponding to read 1000 tuples from a Output Basket at *IoT Web API*.

Normally, the *IoT Web Server* creates a thread per request, but for profiling, this was changed into one single thread per web server. This change was necessary to make the discrimination between the waiting and processing phases of the web servers more transparent on the charts.

Figure 9 depicts the Flame Graph of creating a stream RESTful request in the *IoT Web Server*. This request produces several MAPI requests to store the newly created stream in the database, as well as additional information for the *IoT Web Server*, such as the flushing parameters of the stream. The stream creation is maintained by the `mapi_create_stream` and `validate_schema_and_create_stream` function calls. The chart shows that only the MAPI connection calls and files access are present in the charts. Other instructions such as validating the request's JSON schema are efficient enough to not be caught by the profiling thread. The `validate_schema_and_create_stream` function call takes longer, because it had to flush existing baskets, and hence it requires to call the streaming engine to retrieve them and create the next set of baskets. This call also requires creating a MAPI connection for the stream itself to flush, whenever the flushing parameter is fulfilled. As an immediate conclusion, the performance of *IoT Web Server* largely depends on the efficiency of the MAPI connection and the storage of the baskets.

However the batch insert is the main point of examination for the *IoT Web Server*, as it will become the predominant action in the long term. To profile this request, a 1000 tuple batch insert was done on the stream as created on Figure 9. The Flame Graph of this batch insert is shown in Figure 10. The stream has 4 columns (including the implicit timestamp): a `varchar` column with maximal 32 characters, an `integer` column with both a minimum and maximum value, a `real` column with a default value and another nullable `integer` column. On the batch all the tuples were missing, with two fifths of them missing the nullable column, and other two fifths missing the column with a default value. The `validate_and_insert` call provides extra calls for the validation of the tuples and transpose them into batches for each column. Despite these calls, the writing segment took much longer, and only the `validate_and_insert` was depicted on the chart. During the `flush_baskets`, the MAPI call and the creation of new set of baskets took about the same time. This confirms again the same issues about stream creation, and these two issues should be considered the most important for performance improvements on the *IoT Web Server*.

Finally, Figure 11 shows the process of reading the inserted 1000 tuples by the *Web API Server*. The implementation of this server is very straightforward, and hence the resulting Flame Graph is very simple. In this chart we observe that the `read_next_batch` is the most expensive call in the reading process. In this function, there is an extra call to convert the binary data into textual data which is very small to be shown on the chart. Also in the `read_tuples` call, there is an extra call to transpose the columns, thus re-creating the original tuples. However processing files is the main bottleneck of the *Web API Server* likewise seen in the *IoT Web Server*. In the end, the implementation of both web servers in the web servers is still very efficient in the most parts.

8.2.2 *Tomograph on the streaming engine*

To profile the streaming engine of AIoTA, the Tomograph tool of MonetDB is used. Tomograph is a tool to chart parallel query plan executions and detect performance issues in MonetDB [Gawade and Kersten, 2013] in a similar way to Flame Graphs.

For the profiling we used the continuous query, whose detailed on the MAL plan was discussed in Section 7.4.2. As continuous queries run in the background, it is difficult for Tomograph to detect them, as this tool profiles regular SQL queries. For this reason, the profiling was executing normally the continuous query explicitly, outside the iot scheduler. To make it comparable to previous tests, 1000 distinct tuples were inserted on the mentioned `temperature` stream table. Figure 12 shows the generated Tomograph (the image blank-spaces were cropped to save space). The original MAL plan can be found on Appendix C.1.2.

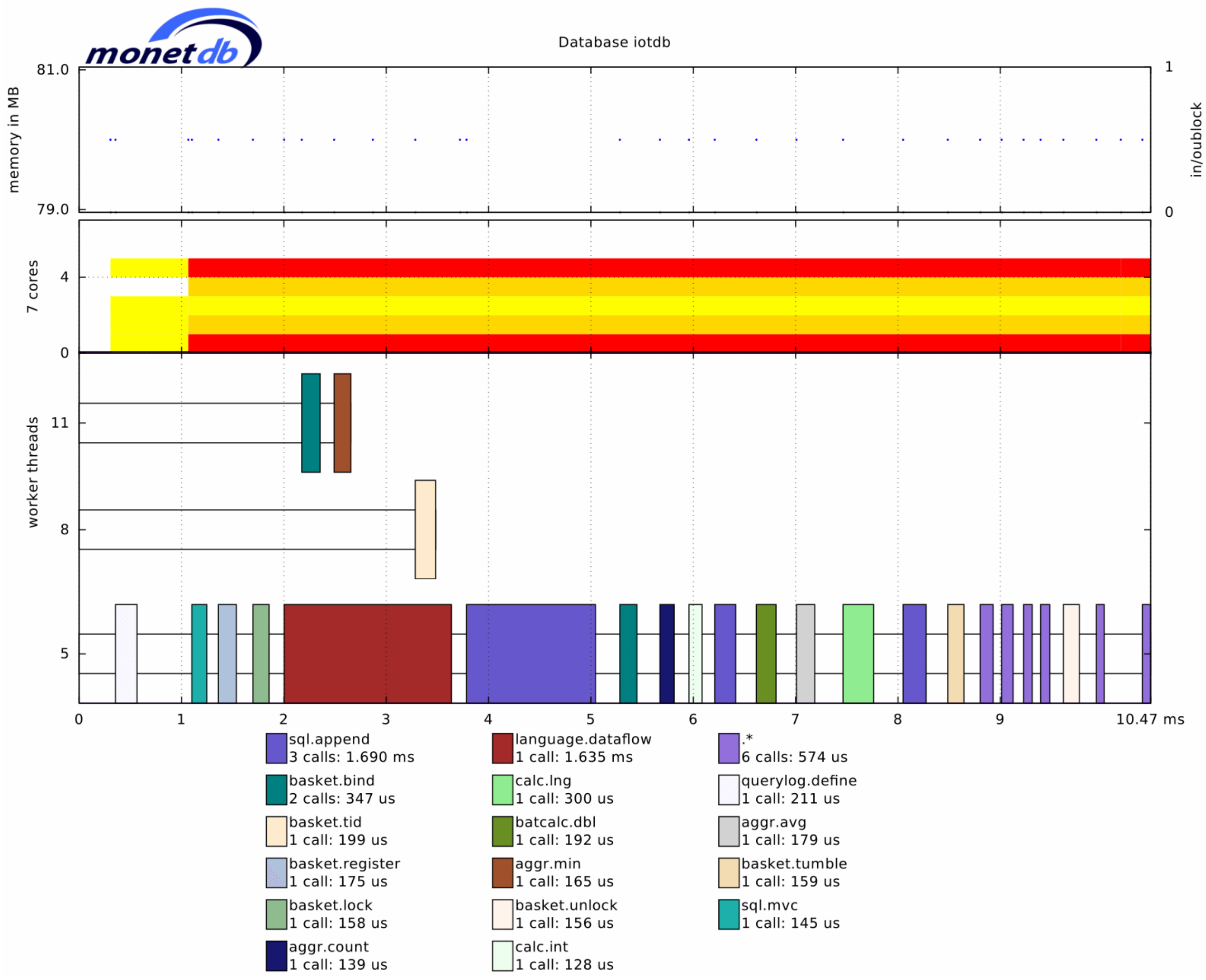


Figure 12.: Result tomograph from temperature examination continuous query.

In the upper part of Figure 12 shows the memory usage during the continuous query execution. The following chart shows the percentage of utilization of all 8 CPU cores (starting the count from 0). The darker the orange tone is, the higher the CPU usage is. Note that these two charts plot the memory and CPU usage for the whole system, instead of only the MonetDB activities. The memory usage kept roughly constant, showing the streaming engine's space efficiency. The CPU usage chart shows that only two cores were intensively used, due to the small parallelization opportunities in the query plan.

The query performance and bottlenecks can be found in the last chart, where it shows the MAL execution of each working thread. Note that not all activities here shown, are precisely related to the CPU usage chart, as tomograph does not provide the information of which thread is being executed by which core. The x-axis shows the time lapse, with the different MAL instructions depicted in different colors (the choice of colors is irrelevant).

The narrower white bars reveal the moments, when the threads were waiting for the next MAL executable instruction to become available.

In the last chart, the `querylog.define` MAL instruction defines the call to procedure explicitly, as it was not present in the original plan. The execution of this query has low parallelism. The reason is that the `sql.append` instruction writes results in shared memory, meanwhile parallelization creates race conditions and hence it cannot be applied. The first `sql.append` call is the longest instruction to execute, because it has to find the `results` table in the SQL catalog and prepare the writing cursor at the end of it to write the aggregations results. The `language.dataflow` also takes a long time, because it has to wait for the parallel executions to finish. In this plan it is evident that more parallelism could be applied (e.g. parallel the `aggr.count` and `aggr.avg` calls), hence the MonetDB team is working to enhance the *dataflow* optimizer.

As it possible to observe in the last chart, the overhead of the `basket` calls is not significant in the plan execution, despite existing 1000 tuples on them which would reflect on the `basket.bind` call. This execution took about 6.5 milliseconds, very comparable to the same execution with 100 tuples which took about 6 milliseconds. The same test was repeated replacing the streaming table with a regular one with 1000 tuples and it took about 14 milliseconds. The `language.dataflow` operator was called more often in the regular table's plan, also a small overhead was induced by the `sql.delta` call, therefore the execution time is more than doubled. The full MAL for the regular table is in Appendix C.1.1.

Note that the Tomograph code is still experimental, and likely still needs improvements. At the end of the main work thread are shown 6 unknown MAL executions. It is not known if it's on fact a bug on Tomograph or calls performed by other running routines on MonetDB.

8.2.3 *AIS benchmarking tests*

Automatic Identification System (AIS)

The Automatic Identification System (AIS) [Guard, 2014] is a ship monitoring and tracking system used by ship and vessel traffic services (VTS) to exchange data electronically between these entities also including AIS base stations and satellites. AIS data supplements marine radars providing information for collision avoidance and other in a comparable way to airplane control in airports.

AIS equipment include a VHF transceiver with a positioning system such as a GPS and other electronic navigation sensors. The information provided include the vessel's coordinates, speed, course, etc. The AIS equipment can then be tracked by nearby AIS base stations, or AIS satellites when out of range. The possible applications are collision avoidance, accidents investigation, marine search and rescue (SAR) operations and Aids to Navigation

(AtoN). The last one is a standard developed to broadcast the positions and names of possible obstacles in the sea, such as navigation aid objects and dynamic data like weather and currents in order to aid navigation.

AIS transceivers work in one of two channel frequencies: A and B. Class A transceivers are targeted at large commercial vessels, such as passenger ship. Due to their high importance, these transceivers transmit more frequently (every 3-5 seconds) and with more data. Class B transceivers are targeted at ship or boats with lower importance such as recreational usage. These transceivers transmit less often (around every 30 seconds), with less potency and data.

For information exchange, AIS uses a set of 27 message types for distinct purposes, including position report, binary data transmission, interrogation, acknowledgment, etc.[3] To view these messages in a computer, the signal must be demodulated in radiotelephone first under the AIS frequencies and then converted into a digital format.

Testing scenario

CWI holds an AIS transceiver, tracking the AIS data around Amsterdam. For this project, a dataset from June 22^{nd} of 2016 was used to create streaming data. There was a total of 1370 distinct vessels and 12 stations with a total of 1824370 AIS messages from the recordings. About 94% of these messages were of 1, 2, 3, and 4 message types, which were about vessels and stations position reports. Therefore the benchmark queries were designed based on these message types.

For the testing scenario, 11 AIS related continuous queries were created to test AIoTA against PipelineDB. The queries are of different levels of complexity, so as to evaluate the performance of several main operations: reading from the input stream, join stream data with a persistent relation and join streams in the current window. Some queries output their results to a stream, while other queries persist their results in regular tables.

In the benchmark it was recreated a situation in which several vessels send AIS data simultaneously to the server, while trying to find the server's saturation point. With this in mind, we measured the evolution of latency against the throughput of inputs on the server for all 11 continuous queries. The comparison will comprise the streaming engine by itself with a MAPI connection, the streaming engine and the *IoT Web Server* together to accommodate the real world scenario and PipelineDB using its native client.

In many benchmarks such as TPC-H, the overall latency of a SQL query is measured. In our case, this type of test is not feasible, because continuous queries run infinitely. At the same time, if we prefer to evaluate the latency by taking a snapshot of a result of continuous query, the evaluation would prevail over the performance of a regular SQL query, which is

3 All available message types and description: `http://www.e-navigation.nl/system-messages?order=field_msg_id&sort=desc`

not related to streaming at all. Instead, our tests will calculate the overall values of latency and throughput of the database system while performing inserts on a single stream, while running benchmark queries in the background and increasing the number of clients. The tests start with 16 clients, doubling in every consecutive test until 1024 clients. The window used was a 10 second time window.

In every test, each vessel represents a client, who sends its position data in an interval of 3, 4 or 5 milliseconds (the interval was downed from seconds to milliseconds to accomplish the tests in a shorter time). The interval given to each vessel will be deterministic for every test, which means that if a vessel gets to send data every 4 milliseconds during the first test, the same interval will be used for the remaining tests for that vessel.

The objective of each test is to study the behavior of the system under increasing workload and to detect the saturation point of the system. To determine the saturation point, it is pretended to calculate the ideal number of clients, so it satisfies the best combination of latency (time to execute a single request) and the throughput (number of requests processed within an unit of time). Remember that is expected to obtain the lowest latency and the highest throughput possible, thus this point becoming the saturation point. After that the latency is expected to grow indefinitely and the throughput will decrease. Therefore the best number of clients will be represented on the lowest right corner point of each line in every test.

Table 11 briefly describes all 11 AIS queries and the addressed features (e.g. stream joins, aggregations, usage of geomodule).

As stated in the functional evaluation (Section 8.1.2), AIoTA and PipelineDB architectures are divergent, thus creating the same continuous queries while under the conditions of both systems becomes difficult. To make this possible, the AIS benchmark queries were implemented for both AIoTA and PipelineDB in the most similar way possible. Despite this, some queries could not be implemented on PipelineDB.

The following three setups were benchmarked: *IoT Web Server* and the streaming engine of AIoTA together, just the streaming engine of AIoTA using a MAPI connection and PipelineDB using its native client. The tests executed in PipelineDB use *continuous transforms* to output query results. However, since *continuous transforms* do not support aggregates on queries, AIS query 2 cannot be evaluated on PipelineDB. In addition because Stream-to-Stream joins are not supported by PipelineDB, AIS queries 5 and 6 can be executed only on AIoTA.

	Description	Features	Output
1	Get the speed of vessels (in knots).	A simple Selection from a stream.	Stream
2	Compute the number of distinct vessels seen in the last 10 seconds.	Aggregation over a stream.	Persistent relation
3	Find currently anchored ship.	Selection from a stream.	Persistent relation
4	Compute the number of ship turning a degree over 180 degrees.	Selection from a stream.	Stream
5	For every ship, find the closest neighbor ship.	Join between the same stream.	Persistent relation
6	For each station, find the ship within a radios of 3 km.	Join between two streams and selection.	Stream
7	Which vessels are currently anchored at the harbors.	Join a stream with a persistent relation and selection. Usage of geospatial information.	Stream
8	Track the movements of a ship S.	A projection from a stream.	Persistent relation
9	Notify when a ship S arrived at any harbor.	Join a stream with a persistent relation and selection. Usage of geospatial information.	Stream
10	Estimated time of arrival of ship S at harbor H.	Join a stream with a persistent relation. Usage of geospatial information. The output table must be continuously updated.	Persistent relation
11	Calculate average speed observed per ship.	Continuous update over a stream (Section 7.3).	Persistent relation

Table 11.: AIS benchmark queries.

The versions of the tested libraries were: PostgreSQL 9.5.3, PipelineDB 0.9.5 and ZeroMQ 4.1.5 (library used by PipelineDB for inter-process communication), as well the stated version of MonetDB in the functionality evaluation. In the *IoT Web Server* tests Python *requests* library 2.11.1 was used for HTTP requests, while on the streaming engine tests, the *pymonetdb* library 1.0.2 was used for the MAPI client.

Due to the potential verbosity caused by displaying the results of all AIS queries, only some of them are displayed here. The MonetDB and PipelineDB implementation of the AIS benchmark queries are included in Appendix D. The latency and throughput of the three test setups of queries 1, 3 and 11 are shown in Figures 13, 14 and 15, respectively. A general observation we can make is that AIoTA performs better.

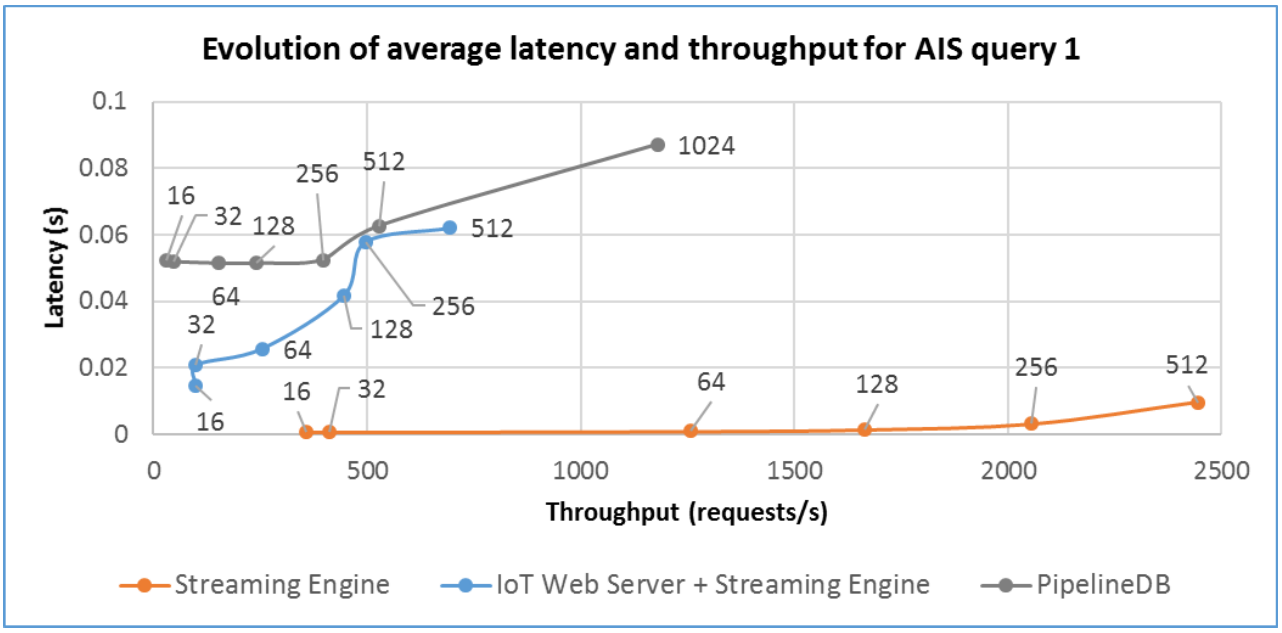


Figure 13.: Evolution of average latency and throughput for AIS query 1 with the number of clients.

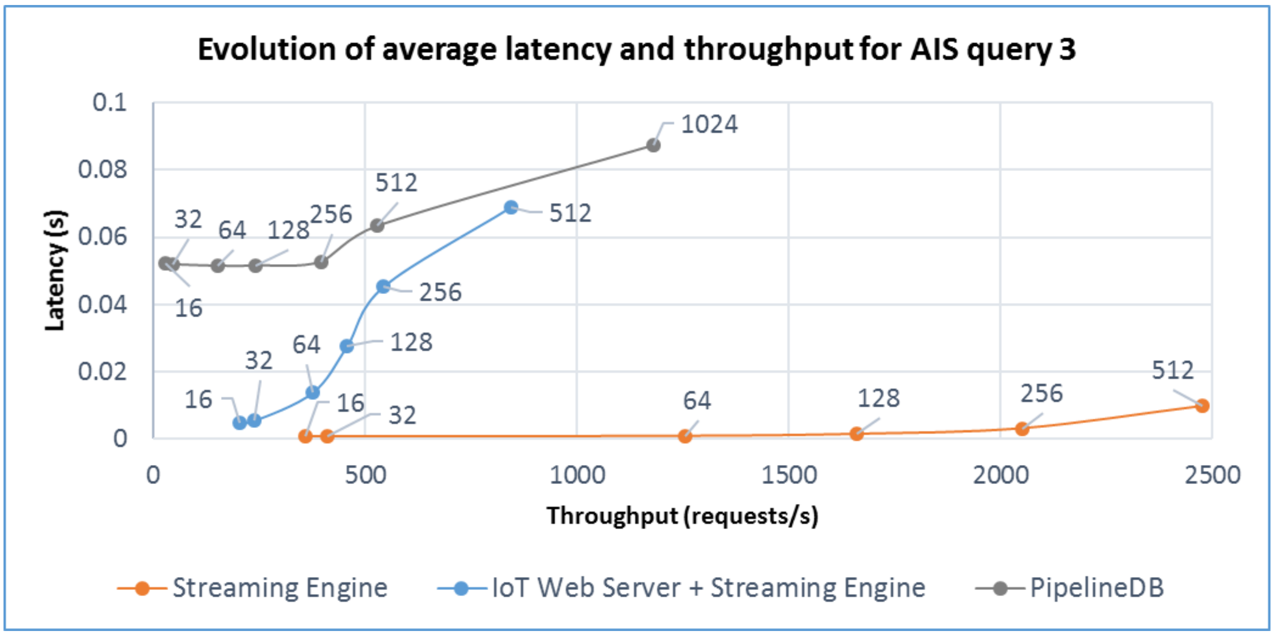


Figure 14.: Evolution of average latency and throughput for AIS query 3 with the number of clients.

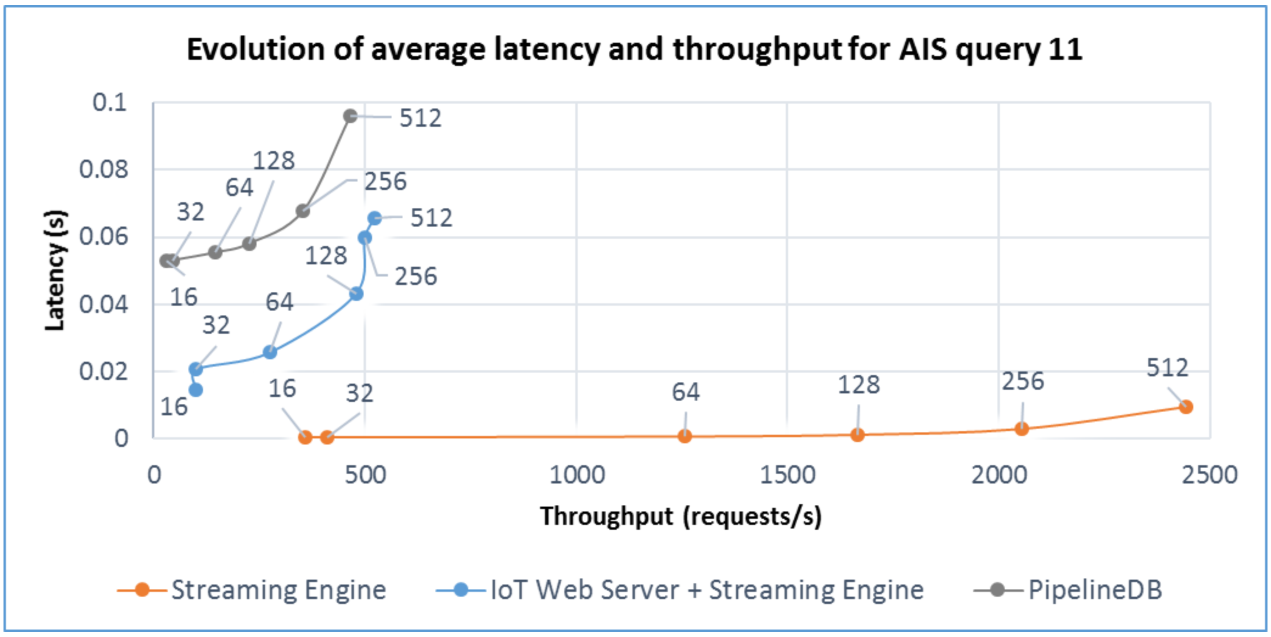


Figure 15.: Evolution of average latency and throughput for AIS query 11 with the number of clients.

Despite considerable efforts, we did not succeed to make PostGIS work with PipelineDB. As a consequence, the queries 7, 9 and 10 crashed on PipelineDB and could not be evaluated. Meanwhile, MonetDB uses the *pthread* library to create a thread for each client on the server. The code is barely tested with a high number of clients, hence MonetDB becomes unstable in these situations, and many of the AIS tests crashed and had to be repeated again. Therefore, we were not able to obtain results on MonetDB with 1024 clients.

The first main difference is *Stream Processing* vs *Batch Processing*. In PipelineDB, whenever a tuple enters the system, the related queries need to be evaluated, while on AIoTA, the related queries get evaluated only once every 10 seconds. As a consequence, writes in PipelineDB have much higher latencies than in AIoTA. At the same time, as more computations were required from the testing system, the overall throughputs on PipelineDB decrease.

Another performance bottleneck in PipelineDB is that for each new client, PipelineDB creates a new process, and it does the same for every *continuous transform* and *continuous query*. It uses ZeroMQ for inter-process communication. It is noticeable that this setup influences the performance of the tests compared to AIoTA. In AIS test 11 with 1024 clients, the system's shared memory crashed, which shows potential problems with high number of clients in PipelineDB as well.

The *IoT Web Server + streaming engine* implementation is generally in the middle of the other two implementations. Due to inter-process communication and the baskets' flushing, it has higher latencies and lower throughputs compared to the *streaming engine* alone test. The overhead induced by the *IoT Web Server + streaming engine* implementation is expected, however is also necessary, because in AIoTA, we want provide a convenient interface to IoT devices. However, *IoT Web Server + streaming engine* implementation generally performs better when compared to PipelineDB even without any special optimizations. In some tests such as AIS Q3, the MonetDB's instability with the *pthread* library will lead to an earlier saturation point, and it is predicable that the *IoT Web Server + streaming engine* implementation would get surpassed by PipelineDB with 1024 clients, if the test concluded. Despite this, the current *IoT Web Server + streaming engine* implementation is still satisfactory.

Finally, we can conclude that *Batch Processing* used in AIoTA leads to a better performance, at the cost of losing a smooth processing of the streams with *Stream Processing*. However the inter-process communication seems to be a significant factor in determining a system's performance. So this should be carefully taken into account in a system's design.

9 CONCLUSION AND FUTURE WORK

9.1 CONCLUSION

The growth of connected networks and embedded systems through the Internet-of-Things has motivated the continued development of streaming engines. Distinct requirements lead to different approaches to build streaming engines for different use case scenarios. Some systems introduced *load shedding* to drop tuples for performance, while others introduced specific query operators to mimic streams in the most authentic way possible.

In recent times, big software companies, such as Microsoft, Oracle and IBM already integrated a DSMS solution into their products (Section 2.9). This means that the DSMS concept is expanding, and soon it will be possible to create the first SQL standard for streaming on databases.

Meanwhile, in this project, we were able to build a simple but flexible streaming engine capable of answering many of the discussed streaming requirements. Most of the MonetDB kernel remained unchanged after the integration of the streaming engine (Section 7.4.2), hence it was possible to finish it in the available time. To make that possible, we leave to the user to decide handling of the behavior of the streaming engine by using SQL defined procedures, including timestamping, windowing and handling of the output. The possible combinations cover a fairly broad spectrum of streaming features.

In our implementation, we managed to build a streaming engine over a database, which granted us a number of important benefits, such as a well-known query language (SQL), optimized relational operators and efficient storage. However, there are other features that are harder to achieve, such as handling stream imperfections (tuples out-of-order, tuples missing) and transactions involving streams. In the end, every system has its own benefits and detriments, hence build a streaming engine that answers all the streaming requirements perfectly is extremely difficult to achieve.

Despite being far from a real commercial product, AIoTA already provides a flexible API in both web servers, while taking advantage of the MonetDB kernel in the streaming engine. As revealed in the evaluation chapter, AIoTA already shows satisfying functionality and performance in comparison to PipelineDB. Currently the code is held at the experimental

”iot” branch of MonetDB, but will be merged with the main branch soon enough and available on MonetDB’s next release for general use.

Finally, AIoTA showed to be a reliable streaming engine for continuous queries over large amount of continuously and rapidly arriving data. This property is important for IoT processing, as the technology continues to grow and is expected to generate high loads of data [Violino, 2013].

9.2 FUTURE WORK

Needless to say, the first AIoTA prototype has left plenty of opportunities open for future work.

Extending the SQL language became a recurrent task in many DSMSs (Section 2.4). Previously, DataCell attempted to extend MonetDB’s query language to accommodate streaming behavior [Liarou et al., 2013]. The reason was to give priority to debug the streaming engine while developing the prototype. Rather than extend the SQL parser, it is preferred to build a reliable streaming engine on simple database object such as a procedure, which makes it easier to debug. For this reason extending the SQL language has a very low priority on AIoTA, and it will be left as future work.

As said before, MonetDB’s kernel code was left mostly unchanged during AIoTA development. A future step would be adding streaming related plan optimizations as mentioned in Section 2.8. To make that possible, some query operators have to be rewritten, which were not given priority during this project. The *Rate-based optimization* will be preferable for IoT and *batch processing*, as it allows to produce higher throughput rates for continuous queries. If possible, this optimization will be taken into consideration in future development of AIoTA.

The scheduler is the main component in the streaming engine, as it is responsible for selecting the continuous queries to execute. The current scheduling is linear, which means that continuous queries are selected by registering order. Later on, we should consider extending the scheduler’s policy to include additional fairness, optimize the system’s resources usage and more importantly a closer implementation to the *Stream processing* model [Babcock et al., 2004].

Synopsis structures have the functionality to provide fast response to queries over a big amount of data while reducing the memory usage [Gibbons and Matias, 1999]. Many DSMSs, such as PipelineDB, have integrated synopsis structures natively as data types. The addition of these structures in AIoTA would be beneficial for several reasons: approximating joins, calculate more complex aggregates such as quantiles, and making query estimations, while predicting the stream size and choosing the better query plan for a continuous query. The available *synopsis structures* include sampling methods, histograms and *sketches*.

There is room for improvement in the web servers as well. New RESTful API's can be added to the *IoT Web Server* to control the scheduler, such as pausing streams and stop the scheduler for several cycles. Similarly, the *Web API Server* can be improved to include new publisher/subscriber notifications. Currently the notifications happen along with baskets creation, which is slow. As a future work, the MonetDB kernel can be extended to include asynchronous notifications to clients. PostgreSQL already accomplishes this with the `LISTEN\NOTIFY` statements.[1] In this way, the notifications can happen in a faster way than the current implementation.

Meanwhile have been developed OSI application layer protocols, specific to IoT devices. The MQ Telemetry Transport (MQTT) protocol is a publish-subscribe protocol over TCP, designed for situations where the network bandwidth is limited.[2] The MQTT is bandwidth efficient, has low resource requirements for IoT devices, while offering message delivery requirements at the same time. Similarly, the Constrained Application Protocol (CoAP) is a protocol designed by IETF for low-power devices and limited network bandwidths with support for multicast over UDP.[3] This protocol has been aligned with HTTP, but for IoT usage by reducing some of the overheads introduced by HTTP. These two protocols have wide IoT usage [Sutaria and Govindachari, 2013], thus adopting them in the *IoT Web Server* would provide additional performance for AIoTA. To further improve performance, IoT devices might provide/receive data in binary format instead of the JSON format, however it must be taken into consideration that different systems have distinct binary representations which should be considered in a future implementation.

In the evaluation chapter, the inter-process communication through the baskets was the main bottleneck showing in the Flame Graphs. As a future work, improvements should be researched. Moving the baskets storage into the main memory instead of leaving them on disk is one of the possible strategies.

1 More information in the official documentation: `https://www.postgresql.org/docs/9.5/static/sql-notify.html` and `https://www.postgresql.org/docs/9.5/static/sql-listen.html`
2 ISO 20922 specification: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=69466`
3 RFC 7252 specification: `https://tools.ietf.org/html/rfc7252`

BIBLIOGRAPHY

Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003. ISSN 1066-8888. doi: 10.1007/s00778-003-0095-z. URL http://dx.doi.org/10.1007/s00778-003-0095-z.

Daniel J. Abadi, Wolfgang Lindner, Samuel R. Madden, and Jorg Schuler. An integration framework for sensor networks and data stream management systems. Demonstration. VLDB, 2004.

Charu C. Aggarwal. *An Introduction to Data Streams*, pages 1–8. Springer US, Boston, MA, 2007. ISBN 978-0-387-47534-9. doi: 10.1007/978-0-387-47534-9_1. URL http://dx.doi.org/10.1007/978-0-387-47534-9_1.

Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

M. Al-Kateb, Byung Suk Lee, and X.S. Wang. Adaptive-size reservoir sampling over data streams. In *Scientific and Statistical Database Management, 2007. SSBDM '07. 19th International Conference on*, pages 22–22, July 2007. doi: 10.1109/SSDBM.2007.29.

Cristina Alcaraz, Pablo Najera, Javier Lopez, and Rodrigo Roman. Wireless sensor networks and the internet of things: Do we need a complete integration? In *1st International Workshop on the Security of the Internet of Things (SecIoT'10)*, page xxxx, Tokyo (Japan), December 2010. IEEE, IEEE.

Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006. ISSN 1066-8888. doi: 10.1007/s00778-004-0147-z. URL http://dx.doi.org/10.1007/s00778-004-0147-z.

Armen S. Asratian, Tristan M. J. Denley, and Roland Häggkvist. *Bipartite Graphs and Their Applications*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-59345-X.

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, December 2004.

ISSN 1066-8888. doi: 10.1007/s00778-004-0132-6. URL http://dx.doi.org/10.1007/s00778-004-0132-6.

Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, September 2001. ISSN 0163-5808. doi: 10.1145/603867.603884. URL http://doi.acm.org/10.1145/603867.603884.

Mike Barnett Robert DeLine Danyel Fisher John C. Platt James F. Terwilliger John Wernsing Badrish Chandramouli, Jonathan Goldstein. Trill: A high-performance incremental query processor for diverse analytics. VLDB Very Large Data Bases, August 2015. URL https://www.microsoft.com/en-us/research/publication/trill-a-high-performance-incremental-query-processor-for-diverse-analytics/.

Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 337–346, New York, NY, USA, 2006. ACM. ISBN 1-59593-433-2. doi: 10.1145/1183614.1183664. URL http://doi.acm.org/10.1145/1183614.1183664.

Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387710027, 9780387710020.

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. Flumejava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010. URL http://dl.acm.org/citation.cfm?id=1806638.

Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872838. URL http://doi.acm.org/10.1145/872757.872838.

CWI. Monetdb architecture. https://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture, 2015. Accessed: 2015-10-30.

Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 40–51, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872765. URL http://doi.acm.org/10.1145/872757.872765.

Dennis de Vries. Amsterdam krijgt crowdsourced iot-netwerk, rest van de wereld volgt snel. http://siliconcanals.nl/nieuws/the-things-network-lanceert-gratis-iot-netwerk-amsterdam/, August 2015. Accessed: 2016-07-18.

Dave Evans. The internet of things how the next evolution of the internet is changing everything. http://www.iotsworldcongress.com/documents/4643185/0/IoT_IBSG_0411FINAL+Cisco.pdf, April 2011. Accessed: 2015-12-14.

Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Scalable and Fault-tolerant Stateful Stream Processing. In Andrew V. Jones and Nicholas Ng, editors, *2013 Imperial College Computing Student Workshop*, volume 35 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–18, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-63-7. doi: http://dx.doi.org/10.4230/OASIcs.ICCSW.2013.11. URL http://drops.dagstuhl.de/opus/volltexte/2013/4266.

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

Minos Garofalakis. *Wavelets on Streams*, pages 3446–3451. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_453. URL http://dx.doi.org/10.1007/978-0-387-39940-9_453.

Mrunal Gawade and Martin Kersten. Tomograph: Highlighting query parallelism in a multi-core system. In *Proceedings of the Sixth International Workshop on Testing Database Systems*, DBTest '13, pages 3:1–3:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2151-8. doi: 10.1145/2479440.2479444. URL http://doi.acm.org/10.1145/2479440.2479444.

Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref, and Ahmed K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. Knowl. Data Eng.*, 19(1):57–72, 2007. URL http://dblp.uni-trier.de/db/journals/tkde/tkde19.html#GhanemHMAE07.

T.M. Ghanem. Supporting predicate-window queries in data stream management systems. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pages x139–x139, 2006. doi: 10.1109/ICDEW.2006.140.

Phillip B. Gibbons and Yossi Matias. Synopsis data structures for massive data sets. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 909–910, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. ISBN 0-89871-434-6. URL http://dl.acm.org/citation.cfm?id=314500.315083.

John Greenough. The internet of things will be the world's most massive device market and save companies billions of dollars. http://uk.businessinsider.com/how-the-internet-of-things-market-will-grow-2014-10?r=US&IR=T, November 2014. Accessed: 2016-05-09.

Brendan Gregg. The flame graph. *Queue*, 14(2):10:91–10:110, March 2016. ISSN 1542-7730. doi: 10.1145/2927299.2927301. URL http://doi.acm.org/10.1145/2927299.2927301.

Georg Gruetter. The internet of things for the rest of us. http://blog.bosch-si.com/categories/technology/2012/09/the-internet-of-things-for-the-rest-of-us/, 9 2012. Accessed: 2015-11-16.

USA Coast Guard. Automatic identification system overview. http://www.navcen.uscg.gov/?pageName=aismain, 2014. Accessed: 2016-07-22.

Sudipto Guha, Nick Koudas, and Kyuseok Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Trans. Database Syst.*, 31(1):396–438, March 2006. ISSN 0362-5915. doi: 10.1145/1132863.1132873. URL http://doi.acm.org/10.1145/1132863.1132873.

Rui Han and Xiaoyi Lu. On big data benchmarking. *CoRR*, abs/1402.5194, 2014. URL http://arxiv.org/abs/1402.5194.

Derrick Harris. Survey shows huge popularity spike for apache spark. http://fortune.com/2015/09/25/apache-spark-survey/, September 2015. Accessed: 2016-02-22.

Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 543–554, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807227. URL http://doi.acm.org/10.1145/1807167.1807227.

Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 413–424, 2007. doi: 10.1145/1247480.1247527. URL http://doi.acm.org/10.1145/1247480.1247527.

Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull*, 2012.

Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik.

Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390, August 2008. ISSN 2150-8097. doi: 10.14778/1454159.1454179. URL http://dx.doi.org/10.14778/1454159.1454179.

Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A heartbeat mechanism and its application in gigascope. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1079–1088. VLDB Endowment, 2005. ISBN 1-59593-154-6. URL http://dl.acm.org/citation.cfm?id=1083592.1083716.

Robert Kajic. Evaluation of the stream query language cql. Master's thesis, Uppsala University - Department of Information Technology, 2010.

Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742788. URL http://doi.acm.org/10.1145/2723372.2742788.

Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 492–503. VLDB Endowment, 2004. ISBN 0-12-088469-0. URL http://dl.acm.org/citation.cfm?id=1316689.1316733.

Erietta Liarou. *MonetDB/DataCell: leveraging the column-store database technology for efficient and scalable stream processing*. PhD thesis, University of Amsterdam, January 2013.

Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Enhanced stream processing in a dbms kernel. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, pages 501–512, Genoa, Italy, 2013.

David Luckham. What's the difference between esp and cep? http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/, August 2006. Accessed: 2016-07-15.

Sergiy Matusevych, Alex Smola, and Amr Ahmed. Hokusai — sketching streams in real time. In *Proceedings of the 28th International Conference on Conference on Uncertainty in Artificial Intelligence (UAI)*, 2012. URL http://www.cs.cmu.edu/~amahmed/papers/hokusai.pdf.

Brian Babcock Mayur, Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding techniques for data stream systems. In *In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS*, 2003.

Martin Kersten Milena Ivanova and Fabian Groffen. *Advances in Databases and Information Systems: 16th East European Conference, ADBIS 2012, Poznań, Poland, September 18-21, 2012. Proceedings*, chapter Just-In-Time Data Distribution for Analytical Query Processing, pages 209–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33074-2. doi: 10.1007/978-3-642-33074-2_16. URL http://dx.doi.org/10.1007/978-3-642-33074-2_16.

Hannes Muhleisen. Citus data cstore_fdw (postgresql column store) vs. monetdb tpc-h shootout. https://www.monetdb.org/content/citusdb-postgresql-column-store-vs-monetdb-tpc-h-shootout, 7 2014. Accessed: 2016-05-01.

Tadao Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, April 1989. doi: 10.1109/5.24143. URL http://dx.doi.org/10.1109/5.24143.

Mohammad Obaid, Ekaterina Kurdyukova, and Elisabeth Andre. City pulse: Supporting going-out activities with a context-aware urban display. In *Proceedings of the 9th International Conference on Advances in Computer Entertainment*, ACE'12, pages 529–532, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34291-2. doi: 10.1007/978-3-642-34292-9_51. URL http://dx.doi.org/10.1007/978-3-642-34292-9_51.

Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *Current Trends in Database — Technology EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*, pages 445–464. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-46788-5. doi: 10.1007/11896548_35. URL http://www.springerlink.com/content/h2462885511852k5/.

Shao Qian and YiLi Lu. A modified chain scheduling algorithm in data stream system. In *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*, volume 4, pages 568–570, Feb 2010. doi: 10.1109/ICCAE.2010.5451576.

Mark Raasveldt. Vectorized udfs in column-stores. Master's thesis, Utrecht University, Utrecht, The Netherlands, December 2015.

Saeed Shahrivari. Beyond batch processing: Towards real-time and streaming big data. *CoRR*, abs/1403.3375, 2014. URL http://arxiv.org/abs/1403.3375.

Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005. ISSN 0163-5808. doi: 10.1145/1107499.1107504. URL http://doi.acm.org/10.1145/1107499.1107504.

Ronak Sutaria and Raghunath Govindachari. Making sense of interoperability: Protocols and standardization initiatives in iot. In *2nd International Workshop on Computing and*

Networking for Internet of Things (CoMNet-IoT) held in conjunction with 14th International Conference on Distributed Computing and Networking (ICDCN 2013), 2013.

Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2595641. URL http://doi.acm.org/10.1145/2588555.2595641.

Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 37–48, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564697. URL http://doi.acm.org/10.1145/564691.564697.

Bob Violino. The 'internet of things' will mean really, really big data. http://www.infoworld.com/article/2611319/computer-hardware/the--internet-of-things--will-mean-really--really-big-data.html, July 2013. Accessed: 2016-01-07.

wot.io. wot.io data service exchange for connected device platforms. https://www.wot.io/wp-content/uploads/2015/11/WOT_045-Product-Brief_4pgs-FINAL-web.pdf, November 2015. Accessed: 2015-12-14.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 431–442, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007617. URL http://doi.acm.org/10.1145/1007568.1007617.

A

IOT WEB SERVER IMPLEMENTATION DETAILS

The source code of *IoT Web Server* is available here: https://dev.monetdb.org/hg/MonetDB/file/iot/clients/iotclient.

A.1 SERVER ARGUMENTS

The *IoT Web Server* takes the following arguments:

```
usage: main.py [-f [DIRECTORY]] [-l [FILE_PATH]] [-po [POLLING]] [-n [NAME]]
               [-ih [HOST]] [-ip [PORT]] [-ah [HOST]] [-ap [PORT]] [-h [HOST]]
               [-p [PORT]] [-d [DATABASE]] [-u [USER]] [-?]
optional arguments:
  -f [DIRECTORY], --filesystem [DIRECTORY]
                        Baskets location directory (default: /var/iotserver)
  -l [FILE_PATH], --log [FILE_PATH]
                        Logging file location (default:
                        /var/log/iot/iotserver.log)
  -po [POLLING], --polling [POLLING]
                        Polling interval in seconds to the database for
                        streams updates (default: 60)
  -n [NAME], --name [NAME]
                        Host identifier name. If not provided, the machine
                        MAC address will be used by default
  -ih [HOST], --ihost [HOST]
                        Administration server host (default: 0.0.0.0)
  -ip [PORT], --iport [PORT]
                        Administration server port (default: 8001)
  -ah [HOST], --ahost [HOST]
                        Application server host (default: 127.0.0.1)
  -ap [PORT], --aport [PORT]
```

```
                      Application server port (default: 8000)
-h [HOST], --host [HOST]
                      MonetDB database host (default: 127.0.0.1)
-p [PORT], --port [PORT]
                      Database listening port (default: 50000)
-d [DATABASE], --database [DATABASE]
                      Database name (default: iotdb)
-u [USER], --user [USER]
                      Database user (default: monetdb)
-?, --help            Display this help
```

A.2 SUPPORTED DATA TYPES

The list contains all available data types on *IOT Web Server*. The available data types were made as much compatible with MonetDB's data types[1]. For each column definition, one of the types from the list must be provided along with the column's name.

For IoT usage, some extra types are added for further validations if necessary. These types are converted internally to an existing MonetDB type, therefore they exist just for added validation. The respective MonetDB mapping is represented after the dash symbol ("-") (e.g. MAC - Char(17), the IoT type MAC maps into MonetDB's Char type with 45 bytes in length).

text, string, clob - clob

String types with unbounded length. The to be inserted data value must be provided as a JSON string.

char, varchar - char

MonetDB's string types with a bounded length. The limit parameter must be provided as an integer. Later on, the insertion must be handed over as a JSON string within the limit.

uuid - uuid

An *Universally Unique Identifier* according to RFC 4122[2].
An UUID example is `550e8400-e29b-41d4-a716-446655440000`.

1 MonetDB data types: https://www.monetdb.org/Documentation/Manuals/SQLreference/Datatypes
2 About RFC 4122: https://www.ietf.org/rfc/rfc4122.txt

mac - char(17)

A *Media Access Control Address* identifier. A MAC example is `48-2C-6A-1E-59-3D`. As MonetDB does not have a MAC equivalent data type yet, the data is stored as `char(17)`.

url - url

A Uniform Resource Locator as a specific type of a URI is validated according to RFC 3987.[3]

inet - inet

An IPv4 address. An IPv4 example is `234.124.12.88`.

inetsix - char(45)

An IPv6 address. The value must be a JSON String. As MonetDB does not have an Ipv6 equivalent data type yet, the data is stored as `char(45)`.

regex - clob

A string always validated with a provided regular expression. The JSON must contain a *regex* key with the regular expression. As MonetDB does not have a *regex* equivalent data type yet, internally is represent as a `CLOB`. For example `"regex": [0-9a-zA-Z]*`, means a string with only alphanumeric characters.

enum - char

A SQL CHAR type validated over a pre-defined array of values. During creation in the JSON, the values key must be presented with a JSON array of strings containing the values of the enum (ex: `"values": ["red", "blue", "green"]`). The default value, if present, must be one of the values.

For all the upcoming types, minimum and maximum values can be added for validation. During the stream creation, if desired these values must be specified in the JSON as `"maximum":` and `"minimum":` respectively. As an example:
`{"name":"hour","type": "integer", "minimum":0, "maximum": 23}` creates an integer column bounded between 0 and 23.

3 About RFC 3987: `https://www.ietf.org/rfc/rfc3987.txt`

tinyint, smallint, integer, bigint, hugeint - tinyint, smallint, integer, bigint, hugeint

These types represent signed integers. The type name specifies the bit capacity (8, 16, 32, 64 and 128 respectively). If the value is grater than the bit capacity, it will be truncated. The insertion must be provided as a JSON integer. During the bootstrap, the server will check if *hugeint* type is supported by the MonetDB server.

real, float, double - real, double

Floating point numbers where the type name specifies its bit capacity. If the inserted value is grater than the bit capacity, it will be truncated to the nearest value. *real* numbers are 32 bit, while *float* and *double* are 64 bit. The insertion must be provided as a JSON float.

decimal, numeric - decimal

Numbers with a specific precision and scale. The precision must be between 1 and 18 (default 18), and the scale between 0 and the precision (default 0). The insertion must be supplied as a JSON float. If *Hugeint* type is available, then the max precision possible will be 38 (e.g. `{"name":"taxrate","type": "decimal", "precision":12, "scale": 10}`).

boolean - boolean

A true or false value. The inserted column has to be a JSON boolean.

date - date

A date in the in the Gregorian Calendar. The insertion must be expressed as a string in format `YYYY-MM-DD` (e.g. `2016-07-31`). Likewise numbers, a minimum and a maximum values can be defined for further validation in Date, Time and Timestamp types.

time - time

The time of the day with timezone. The insertion must be expressed as a string in the regular expression format `HH:MM:SS\.sss(Z|([+-]HH:MM))?` (e.g. `12:30:45.222+01:00`).

timestamp - timestamp

A timestamp according to RFC 3339[4] with timezone. The regular expression is the standard ISO 8601[5] string with timezone.
An example ISO 8601 string insertion is `2016-07-19T12:09:58+02:00`.

4 About RFC 3339: https://www.ietf.org/rfc/rfc3339.txt
5 About ISO 8601 format: https://www.w3.org/TR/NOTE-datetime

interval - interval

Intervals of time according to the grammar provided in MonetDB documentation[6] without the precision. The insertion must be supplied as a JSON integer.

A.3 RESTful API

All features of the *IoT Web Server* are available through a RESTful API. In this section, each RESTful resource's URI is listed along with the corresponding HTTP method: GET, POST or DELETE. In a request URI parameters are indicated by the "<" and ">" symbols, and should be replaced by their actual value.

A.3.1 *Administration Server*

The administration server provides the functionality to create and delete streams. Should be listening exclusively on *localhost*.

/context - POST method

Creates a stream using a pre-defined JSON schema. The JSON must include the stream's schema, the stream's name, the flushing method which can be time based (`time`), tuple based (`tuple`) or automatic (`auto`), and the stream's columns description. For tuple based flushing, the number of tuples to flush must be provided using the number field. For time based flushing, the interval field tells the time units between flushes and the unit field must be "s", "m" or "h" for seconds, minutes or hours respectively. For each column, one must specify its type, name, and optionally, a default value and if this column's value is nullable. See Appendix A.2 for all supported data types, and their formats.

If the *has_hostname* parameter is supplied as true, then an additional column will be created in the stream with the *IoT Web Server*'s replica *hostname* where each tuple was inserted (to be used with horizontal scaling). A JSON request example for this resource is shown bellow:

```
{
  "schema": "measures",
  "stream": "temperature",
  "has_hostname": false,
  "flushing": {
    "base": "time",
```

6 Grammar for time intervals: https://www.monetdb.org/Documentation/SQLreference/Temporal

```
    "number": 5,
    "interval": "m"
  },
  "columns": [
    { "type": "real", "name": "temperature", "nullable": true },
    { "type": "clob", "name": "sensorid", "default": "living room" }
  ]
}
```

The corresponding create SQL statement would be:
`CREATE STREAM TABLE measures.temperature (temperature REAL NULL, sensorid TEXT NOT NULL DEFAULT "living room", implicit_timestamp TIMESTAMP WITH TIME ZONE NOT NULL);`
The `STREAM` keyword is used to identify this table as a stream table as opposed to a regular table. This distinction is necessary for the plan generation of continuous queries (Section 7.4). Note that a `implicit_timestamp` column is automatically added to denote the time of insertion of the tuple.

/context - DELETE method

Deletes an existing stream. Only the stream's schema and name are required. To delete the stream created in the previous example, one can use the following JSON request:

```
{
  "schema": "measures",
  "stream": "temperature"
}
```

/streams - GET method

Returns a JSON response with details about all the streams currently active on the *IoT Web Server*. For each stream, besides its schema and name, it has the *hostname* column and an *implicit _timestamp* (streams created in MonetDB do not have this timestamp), it provides description of its columns, and the flushing method with the number of tuples inserted in the current basket. An example response with one stream is shown bellow:

```
{
  "streams_count": 1,
  "streams_listing": [
    {
      "schema": "measures",
      "stream": "temperature",
      "has_timestamp": true,
      "has_hostname": false,
      "flushing": {
        "base": "time",
        "number": 5,
        "interval": "m",
        "tuples_inserted_per_basket": 0
      },
      "columns": [
        { "type": "real", "name": "temperature", "nullable": true },
        { "type": "clob", "name": "sensorid", "default": "living room" }
      ]
    }
  ]
}
```

A.3.2 *Application Server*

The application server supplies resources to make insertions on streams. Should be listening to all network interfaces.

/streams - GET method

Same as /streams - GET method from Administration Server above.

/stream/<schema_name>/<stream_name> - POST method

Insert a batch of tuples on the given stream in the URI. The insertion must be an array of JSON objects with pairs of column-value. All tuples are validated according to the data types defined for each column and the JSON schema generated for the stream. If there is an invalid tuple, none of the tuples are inserted. The implicit timestamp and the host identifier are automatically added by the server as well the default and null values. Bellow is an example showing how to insert several tuples into the ″temperature″ stream in the ″measures″ schema, with a POST `/stream/measures/temperature` request. For the first tuple, the ″sensorid″ value is ″living room″ as the default value, while the third ″temperature″ value is NULL.

```
[
  { "temperature": 32.6 },
  { "sensorid": "kitchen", "temperature": 34.2 },
  { "sensorid": "bathroom" }
]
```

/stream/<schema_name>/<stream_name> - GET method

Returns a JSON response with details about the requested stream. In the same format as the responses in the `/streams` resource, but contains information at only one stream. Bellow is an example JSON response of GET `/stream/measures/temperature` after the tuples in the above request have been inserted:

```
{
    "schema": "measures",
    "stream": "temperature",
    "has_timestamp": true,
    "has_hostname": false,
    "flushing": {
        "base": "time",
        "number": 5,
        "interval": "m",
        "tuples_inserted_per_basket": 3
    },
    "columns": [
        { "type": "real", "name": "temperature", "nullable": false },
        { "type": "clob", "name": "sensorid", "nullable": false }
    ]
}
```

B WEB API SERVER IMPLEMENTATION DETAILS

The source code of *Web API Server* is available here: https://dev.monetdb.org/hg/MonetDB/file/iot/clients/iotapi.

B.1 SERVER ARGUMENTS

The *Web API Server* takes the following arguments:

```
usage: main.py [-f [DIRECTORY]] [-l [FILE_PATH]] [-po [POLLING]] [-sh [HOST]]
               [-sp [PORT]] [-h [HOST]] [-p [PORT]] [-d [DATABASE]]
               [-u [USER]] [-?]
optional arguments:
  -f [DIRECTORY], --filesystem [DIRECTORY]
                        Baskets location directory (default: /var/iotapi)
  -l [FILE_PATH], --log [FILE_PATH]
                        Logging file location (default:
                        /var/log/iot/iotapi.log)
  -po [POLLING], --polling [POLLING]
                        Polling interval in seconds to the database for
                        streams updates (default: 60)
  -sh [HOST], --shost [HOST]
                        Web API server host (default: 0.0.0.0)
  -sp [PORT], --sport [PORT]
                        Web API server port (default: 8002)
  -h [HOST], --host [HOST]
                        MonetDB database host (default: 127.0.0.1)
  -p [PORT], --port [PORT]
                        Database listening port (default: 50000)
  -d [DATABASE], --database [DATABASE]
                        Database name (default: iotdb)
  -u [USER], --user [USER]
                        Database user (default: monetdb)
  -?, --help            Display this help
```

B.2 WEBSOCKETS API

A Websockets session is asynchronous, which means that a request or response message might be provided at any time during the session. For that reason, all messages are labeled with an identifier to discriminate the client's request on the server, and the respective response to the client using a defined JSON schema.

Clients requests must be handed over as text frames, with a JSON formatted string. All the requests must include a `request` field indicating the intended action to perform in the server, followed by the other specific fields depending on the request. The possible requests are listed bellow. The featured examples use the *temperatures* stream also used in the *IoT Web Server*'s examples. These examples can be interpreted as the output of a continuous query that performs the identity function on the input stream (copying the input to the output with no changes), thus generating the same output content.

B.2.1 *Requests*

To be performed by the web clients.

subscribe

Subscribes for notifications of new baskets from a specific output stream. Whenever a basket is created, the server sends a notification message indicating the number of inserted tuples in the new basket. The web client has to specify the stream's name and schema. To subscribe to our temperature stream, the following JSON would suffice:

```
{
  "request": "subscribe",
  "schema": "measures",
  "stream": "temperature"
}
```

unsubscribe

Unsubscribes a previous subscribed stream for a client. The web client has to specify the stream's name and schema. The format of the request is the same as above, just changing the request keyword from "subscribe" to "unsubscribe".

read

Reads output result from baskets generated by a stream. The web client does not have to be subscribed to the stream in order to read from it. It is possible to provide an offset, a limit and a basket number where the read should start. The request will always provide a result, even if the query results in zero tuples. If the number of tuples in the available baskets is less than requested in the request, the remaining tuples will be retrieved from the next basket if exists. The web client has to specify the stream's name and schema. An example request for the temperatures stream is as follows:

```
{
  "request": "read",
  "schema": "measures",
  "stream": "temperature",
  "basket": 1,
  "offset": 0,
  "limit": 3
}
```

info

Retrieves information about a giving stream if a stream's name and schema are provided, otherwise all existing streams in the system. To request information of the temperature stream:

```
{
  "request": "info",
  "schema": "measures",
  "stream": "temperature"
}
```

B.2.2 *Responses*

To be delivered by the *Web API Server*.

error

A message reporting an internal error occurred on the server. The `message` contains a string explaining the error. The following error message happens when the web client attempts to unsubscribed from a not-subscribed stream:

```
{
  "response": "error",
  "message":
    "Stream measures.temperature is not present in the user's subscriptions!"
}
```

subscribed

Message confirming the subscription to new baskets notifications of a stream.

```
{
  "response": "subscribed",
  "schema": "measures",
  "stream": "temperature"
}
```

unsubscribed

Message confirming the removal of a subscription to new baskets notifications of a stream. The message format is the same as the "subscribed" response, only with the response type changed into "unsubscribed".

removed

If a stream is removed in the MonetDB engine (using the SQL *Front-End*), while there are still clients subscribed, then this message is sent. The message format is the same as the "subscribed" response, only with the response type changed into "removed".

notification

Notification of a new basket creation for a subscribed stream. The message contains the basket number and the number of tuples in the new basket. An example notification response for the temperatures stream is as follows:

```
{
  "response": "notification",
  "schema": "measures",
  "stream": "temperature",
  "basket": 2,
  "count": 50
}
```

read

Response message for a read query. Contains a list of the reconstructed tuples from the output baskets. If a column has a null value, the JSON's `null` value will be used. The query result for the early "read" request for the temperatures stream is as follows:

```
{
  "response": "read",
  "schema": "measures",
  "stream": "temperature",
  "count": 3,
  "tuples": [
    { "sensorid": "living room", "temperature": 32.6,
      "implicit_timestamp": "2016-06-17T09:23:22+00:00" },
    { "sensorid": "kitchen", "temperature": 34.2,
      "implicit_timestamp": "2016-06-17T09:23:22+00:00" },
    { "sensorid": "bathroom", "temperature": null,
      "implicit_timestamp": "2016-06-17T09:23:22+00:00" }
  ]
}
```

data

Returns a info message regarding all the streams in the system. It provides the details of each column definition, number of baskets and the number of tuples in each. Note that the possible types list are restricted to the MonetDB kernel. An example with the temperatures stream:

```
{
  "response": "data",
  "streams_count": 1,
  "streams_listing": [
    {
      "schema": "measures",
      "stream": "temperature",
      "columns": [
        { "name": "sensorid", "type": "clob",
          "nullable": false, "default": "living room" },
        { "name": "temperature", "type": "real",
          "nullable": true, "default": null },
        { "name": "implicit_timestamp", "type": "timestamp with time zone",
          "nullable": false, "default": null }
      ],
      "baskets_count": 3,
      "baskets_listing": [
        { "number": 1, "count": 3 },
        { "number": 2, "count": 25 },
        { "number": 3, "count": 12 }
      ]
    }
  ]
}
```

info

Message with details about a stream including both columns and baskets details. The message format is similar to a ″data″ response, but with information of only one stream. The JSON ″streams_listing″ array gets deprecated, and the single stream data is provided in the core JSON object.

```
{
  "response": "info",
  "schema": "measures",
  "stream": "temperature",
  "columns": [
    { "name": "sensorid", "type": "clob",
      "nullable": false, "default": "living room" },
    { "name": "temperature", "type": "real",
      "nullable": true, "default": null },
    { "name": "implicit_timestamp", "type": "timestamp with time zone",
      "nullable": false, "default": null }
  ],
  "baskets_count": 3,
  "baskets_listing": [
    { "number": 1, "count": 3 },
    { "number": 2, "count": 25 },
    { "number": 3, "count": 12 }
  ]
}
```

C

MONETDB STREAMING ENGINE IMPLEMENTATION DETAILS

C.1 FINAL MAL EXECUTION PLANS

This section lists the final MAL execution plans (after all optimizers have been applied) for the queries evaluated on Section 7.4.2 using the respective optimization pipelines. For a better understanding of these plans, a reading over the MAL reference on MonetDB website is recommended, as a full description of MAL language would be too verbose to detail on this report.[1]

C.1.1 *Regular table*

The following MAL plan depicts the final execution plan for the example query on a regular table using the EXPLAIN statement, using the *default_pipe* optimization pipeline.

```
+-----------------------------------------------------------------------------+
| mal                                                                         |
+=============================================================================+
| function user.s16_1():void;                                                 |
|     X_45:void := querylog.define("explain insert into results select       |
|                                  min(val), count(*), avg(val) from         |
|                                   temperature;", "default_pipe",29);       |
| barrier X_64 := language.dataflow();                                        |
|     X_0 := sql.mvc();                                                       |
|     C_1:bat[:oid] := sql.tid(X_0,"iot","temperature");                      |
|     X_4:bat[:lng] := sql.bind(X_0,"iot","temperature","val",0);            |
|     (C_7:bat[:oid],r1_8:bat[:lng]) := sql.bind(X_0,"iot","temperature",    |
|                                                         "val",2);          |
|     X_10:bat[:lng] := sql.bind(X_0,"iot","temperature","val",1);           |
|     X_12 := sql.delta(X_4,C_7,r1_8,X_10);                                  |
```

1 MAL reference: https://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

```
|     X_13 := algebra.projection(C_1,X_12);                                     |
|     X_14 := aggr.min(X_13);                                                   |
| exit X_64;                                                                    |
|     X_16 := sql.append(X_0,"iot","results","minimum",X_14);                   |
| barrier X_67 := language.dataflow();                                          |
|     X_19:bat[:timestamp] := sql.bind(X_16,"iot","temperature","t",0);         |
|     (C_21:bat[:oid],r1_22:bat[:timestamp]) := sql.bind(X_16,"iot",            |
|                                                          "temperature","t",2);|
|     X_23:bat[:timestamp] := sql.bind(X_16,"iot","temperature","t",1);         |
|     X_24 := sql.delta(X_19,C_21,r1_22,X_23);                                  |
|     X_25 := algebra.projection(C_1,X_24);                                     |
|     X_26 := aggr.count(X_25);                                                 |
|     X_27 := calc.int(X_26);                                                   |
| exit X_67;                                                                    |
|     X_29 := sql.append(X_16,"iot","results","tuples",X_27);                   |
|     X_31:bat[:dbl] := batcalc.dbl(3,X_13);                                    |
|     X_35:dbl := aggr.avg(X_31);                                               |
|     X_36 := calc.lng(X_35,18,3);                                              |
|     X_39 := sql.append(X_29,"iot","results","average",X_36);                  |
|     sql.affectedRows(X_39,1);                                                 |
| end user.s16_1;                                                               |
| #inline               actions= 0 time=10 usec                                 |
| #remap                actions= 1 time=12 usec                                 |
| #costmodel            actions= 1 time=3 usec                                  |
| #coercion             actions= 1 time=5 usec                                  |
| #evaluate             actions= 0 time=2 usec                                  |
| #aliases              actions= 0 time=6 usec                                  |
| #mergetable           actions= 0 time=28 usec                                 |
| #deadcode             actions= 0 time=7 usec                                  |
| #aliases              actions= 0 time=4 usec                                  |
| #constants            actions= 0 time=5 usec                                  |
| #commonTerms          actions= 0 time=5 usec                                  |
| #projectionpath       actions= 0 time=3 usec                                  |
| #deadcode             actions= 0 time=5 usec                                  |
| #reorder              actions= 1 time=26 usec                                 |
| #reduce               actions=24 time=9 usec                                  |
| #matpack              actions= 0 time=2 usec                                  |
| #dataflow             actions=28 time=21 usec                                 |
| #multiplex           actions= 0 time=2 usec                                  |
| #profiler            actions= 1 time=4 usec                                  |
| #candidates          actions= 1 time=1 usec                                  |
| #garbagecollector    actions= 1 time=16 usec                                 |
| #total               actions= 1 time=291 usec                                |
+------------------------------------------------------------------------------+
```

C.1.2 *Streaming table*

The next MAL plan details the final execution plan for the example query on a streaming table using the `iot.show` procedure call, using the *iot_pipe* optimization pipeline.

```
+-----------------------------------------------------------------------+
| mal                                                                   |
+=======================================================================+
| unsafe function user.iot_examine_temperatures():void;                 |
|     X_0 := sql.mvc();                                                 |
|     X_34 := basket.register(X_0,"iot","temperature",0);               |
|     X_38 := basket.lock(X_34,"iot","temperature");                    |
| barrier X_63 := language.dataflow();                                  |
|     C_1:bat[:oid] := basket.tid(X_0,"iot","temperature");             |
|     X_4:bat[:lng] := basket.bind(X_34,"iot","temperature","val");     |
|     X_8 := aggr.min(X_4);                                             |
| exit X_63;                                                            |
|     X_10 := sql.append(X_38,"iot","results","minimum",X_8);           |
|     X_13:bat[:timestamp] := basket.bind(X_10,"iot","temperature","t"); |
|     X_16 := aggr.count(X_13);                                         |
|     X_17 := calc.int(X_16);                                           |
|     X_19 := sql.append(X_10,"iot","results","tuples",X_17);           |
|     X_21:bat[:dbl] := batcalc.dbl(3,X_4);                             |
|     X_25:dbl := aggr.avg(X_21);                                       |
|     X_26 := calc.lng(X_25,18,3);                                      |
|     X_29 := sql.append(X_19,"iot","results","average",X_26);          |
|     X_39 := basket.tumble(X_29,"iot","temperature");                  |
| catch SQLexception:str;                                               |
|     iot.error("user","examine_temperatures",SQLexception);            |
| exit SQLexception:str;                                                |
| catch MALexception:str;                                               |
|     iot.error("user","examine_temperatures",MALexception);            |
| exit MALexception:str;                                                |
|     basket.unlock(X_39,"iot","temperature");                          |
| end user.iot_examine_temperatures;                                    |
| #inline              actions= 0 time=6 usec                           |
| #candidates          actions= 1 time=2 usec                           |
| #remap               actions= 1 time=15 usec                          |
| #iot                 actions= 1 time=149 usec                         |
| #costmodel           actions= 1 time=2 usec                           |
| #coercion            actions= 0 time=3 usec                           |
| #evaluate            actions= 0 time=5 usec                           |
| #aliases             actions= 0 time=32 usec                          |
| #mergetable          actions= 0 time=94 usec                          |
| #deadcode            actions= 0 time=11 usec                          |
| #aliases             actions= 0 time=9 usec                           |
| #constants           actions= 3 time=10 usec                          |
| #commonTerms         actions= 0 time=7 usec                           |
| #projectionpath      actions= 0 time=14 usec                          |
| #deadcode            actions= 0 time=8 usec                           |
| #reduce              actions=33 time=13 usec                          |
| #matpack             actions= 0 time=4 usec                           |
| #dataflow            actions=31 time=28 usec                          |
| #multiplex           actions= 0 time=4 usec                           |
| #profiler            actions= 1 time=1 usec                           |
| #garbagecollector    actions= 1 time=26 usec                          |
| #total               actions= 1 time=726 usec                         |
+-----------------------------------------------------------------------+
```

D

AIS BENCHMARK QUERIES

In this chapter, we include the AIoTA and PipelineDB implementations of the AIS benchmark queries. In each query, current timestamp is retrieved to indicate the time of execution of the query.

D.1 AIOTA QUERIES

The following SQL code represents the baseline for all AIS queries in AIoTA. According to AIS benchmark query itself, the output table `aisr` and the continuous query `aisq` will change. For the *IoT Web Server + streaming engine* tests, one extra SQL statement is used for the *IoT Web Server* table to specify its flushing parameters. With the configuration bellow, it tells the server to flush the vessels stream every 10 seconds.

```
CREATE SCHEMA ais;
SET SCHEMA ais;

CREATE STREAM TABLE vessels (implicit_timestamp TIMESTAMP, mmsi INTEGER,
lat REAL, lon REAL, nav_status SMALLINT, sog REAL, rotais SMALLINT);

INSERT INTO iot.webserverstreams /* Front-End + Back-End only */
SELECT tabl.id, 2 , 10, 's' FROM sys.tables tabl
INNER JOIN sys.schemas sch ON tabl.schema_id = sch.id
WHERE tabl.name = 'vessels' AND sch.name = 'ais';

CREATE [STREAM] TABLE aisr ....
CREATE PROCEDURE aisq() .....

CALL iot.heartbeat('ais', 'vessels', 10000);
CALL iot.query('ais', 'aisq');
```

D.1.1 *AIS query* 1

```
CREATE STREAM TABLE aisr (calc_time TIMESTAMP, mmsi INTEGER, sog REAL);

CREATE PROCEDURE aisq()
BEGIN
    INSERT INTO aisr WITH data_time AS (SELECT current_timestamp AS cur_time)
    SELECT cur_time, mmsi, sog FROM vessels CROSS JOIN data_time
    WHERE (implicit_timestamp, mmsi) IN
    (SELECT max(implicit_timestamp), mmsi FROM vessels GROUP BY mmsi);
END;
```

D.1.2 *AIS query* 3

```
CREATE TABLE aisr (calc_time TIMESTAMP, mmsi INTEGER);

CREATE PROCEDURE aisq()
BEGIN
    INSERT INTO aisr WITH data_time AS (SELECT current_timestamp AS cur_time)
    SELECT cur_time, mmsi FROM vessels CROSS JOIN data_time
    WHERE nav_status = 1 AND (implicit_timestamp, mmsi) IN
    (SELECT max(implicit_timestamp), mmsi FROM vessels GROUP BY mmsi);
END;
```

D.1.3 *AIS query* 11

```
CREATE TABLE aisr (calc_time TIMESTAMP, mmsi INTEGER,
speed_sum REAL, speed_count INTEGER);

CREATE PROCEDURE aisq()
BEGIN
    UPDATE aisr SET
      calc_time = current_timestamp,
      speed_sum = speed_sum + (SELECT COALESCE(SUM(sog), 0)
         FROM vessels INNER JOIN aisr ON vessels.mmsi = aisr.mmsi),
      speed_count = speed_count + (SELECT COUNT(*)
         FROM vessels INNER JOIN aisr ON vessels.mmsi = aisr.mmsi)
    FROM vessels WHERE aisr.mmsi = vessels.mmsi;
    INSERT INTO aisr WITH
      data_time AS (SELECT current_timestamp AS cur_time),
      new_inserts AS (SELECT mmsi, SUM(sog) AS sum_sog, COUNT(*) AS count_tuples
        FROM vessels WHERE mmsi NOT IN (SELECT mmsi FROM aisr) GROUP BY mmsi)
      SELECT cur_time, mmsi, sum_sog, count_tuples
      FROM new_inserts CROSS JOIN data_time;
END;
```

D.2 PIPELINEDB QUERIES

With the *continuous transforms* feature of PipelineDB, the user must specify a procedure object to execute whenever a tuple arrives at the stream. To insert into another stream, PipelineDB provides the `pipeline_stream_insert` procedure. However, to insert in a regular table, the user needs to create a trigger (see AIS queries 3 and 11). For every test using PipelineDB, the following SQL code was used:

```
CREATE SCHEMA IF NOT EXISTS ais;
SET SCHEMA 'ais';

CREATE STREAM vessels (implicit_timestamp TIMESTAMP, mmsi INTEGER,
lat REAL, lon REAL, nav_status SMALLINT, sog REAL, rotais SMALLINT);

CREATE {STREAM | TABLE} aisr ...
CREATE CONTINUOUS TRANSFORM aisq AS ...
```

D.2.1 *AIS query* 1

```
CREATE STREAM aisr (calc_time TIMESTAMP, mmsi INTEGER, sog REAL);

CREATE CONTINUOUS TRANSFORM aisq AS
  SELECT clock_timestamp(), mmsi, sog FROM vessels
  THEN EXECUTE PROCEDURE pipeline_stream_insert('ais.aisr');
```

D.2.2 *AIS query* 3

```
CREATE TABLE aisr (calc_time TIMESTAMP, mmsi INTEGER);
CREATE OR REPLACE FUNCTION insert_into_aisr() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO ais.aisr VALUES (NEW.sel_time, NEW.mmsi);
END; $$ LANGUAGE plpgsql;

CREATE CONTINUOUS TRANSFORM aisq AS
   SELECT clock_timestamp() AS sel_time, mmsi FROM vessels WHERE nav_status = 1
   THEN EXECUTE PROCEDURE insert_into_aisr();
```

D.2.3 *AIS query* 11

```
CREATE TABLE aisr (calc_time TIMESTAMP, mmsi INTEGER, speed_sum REAL,
speed_count INTEGER);

CREATE OR REPLACE FUNCTION insert_into_aisr() RETURNS TRIGGER AS $$
BEGIN
     IF EXISTS (SELECT 1 FROM ais.aisr WHERE mmsi = NEW.mmsi) THEN
        UPDATE ais.aisr SET
           calc_time = NEW.calc_time,
           speed_sum = speed_sum + NEW.speed_sum,
           speed_count = speed_count + NEW.speed_count WHERE mmsi = NEW.mmsi;
     ELSE
        INSERT INTO ais.aisr VALUES
           (NEW.calc_time, NEW.mmsi, NEW.speed_sum, NEW.speed_count);
     END IF;
END; $$ LANGUAGE plpgsql;

CREATE CONTINUOUS TRANSFORM aisq AS
    SELECT clock_timestamp() AS calc_time, mmsi, sog AS speed_sum, 1 AS speed_count
    FROM vessels THEN EXECUTE PROCEDURE insert_into_aisr();
```