UiO : **Department of Informatics**
University of Oslo

Smartphone Assisted Complex Event Processing

Enablement of Esper on Android-based devices

Marcel Eggum

Master's Thesis Autumn 2014

Smartphone Assisted, Complex Event Processing

Marcel Eggum

August 15th, 2014

Abstract

This thesis is concerned about the enablement and evaluation of Esper, a state of the art Complex Event Processing engine that could prove useful within the domain of ubiquitous, smartphone assisted healthcare and monitoring. Previous attempts to enable Esper have proven to be inconsistent and incomplete, and measurements that indicate the performance of this engine are limited.

The main intention of this thesis is therefore to identify and resolve the limitations that prevent Esper from operating correctly, and to measure the maximum achievable performance in this environment. Our evaluation includes a collection of primitive tasks that exercise fundamental principles of event processing and data stream management. We found that smartphone devices are suitable for the purpose of ubiquitous event processing but contain constraints that could prove real time requirements hard to achieve due to an increasing amount of executional pauses caused by memory allocation and de-allocation procedures. An enablement of Esper on smartphone device is from our viewpoint best suited to environments that emit higher level events with low rate of occurrence. Our work has enabled the use of Esper in contexts that were never before possible and will hopefully ensure that future revisions of the engine are embeddable and maintainable.

Preface

I would first and foremost like to thank Kristoffer Robin Stokke for the aid and moral support that I received during our time. I believe that thesis this would never be present without you. I wish you the best, and present my deepest gratitude.

I would also like to thank Ellen Munthe-Kaas and Vera Goebel for everything that they have done for me. You have ensured my growth and given me the possibility to explore the domain of Informatics in a way that I would never have discovered on my own.

Thank you Ivana, Lucyna, Julia and Nils. The love and support that I so needed throughout the time of this work kept me afloat when I just wanted to stop swimming. I hope to be a better companion, son and brother for the future that is to come.

This thesis was originally oriented around the use of an Ultra Wideband Impulse Radar (UWB-IR) [1] that arose from a collaboration between the University of Oslo and Novelda, and resulted in a low emission, CMOS based sensor that measures only 5x5 millimeters. The UWB-IR could be embedded into appliances that are small and light enough to be carried by humans, but require a central aggregator that is capable of extracting and enriching information that it detects. We believed that the UWB-IR could be utilized to detect motions caused by the heart and lungs and the main motivation for enabling a Complex Event Processing engine on a smartphone device, was directly related to this.

A profound amount of time was spent to understand and integrate the UWB-IR with our work, but our results deviated from the original intent of this thesis and could not be included. We describe this work and the problems that arose in Appendix A, and welcome the reader to explore this section as a distinct part of this thesis.

Contents

List of figures

List of tables

Abbreviations

Abbreviation	**Description**
SoC	System on Chip
JNI	Java Native Interface
NDK	Native Development Kit
SDK	Software Development Kit
JDBC	Java Database Connectivity
SQL	Structured Query Language
API	Application Programming Interface
JAR	Java Archive
CEP	Complex Event Processing
DSM	Data Stream Management
UWB-IR	Ultra Wideband Impulse Radar

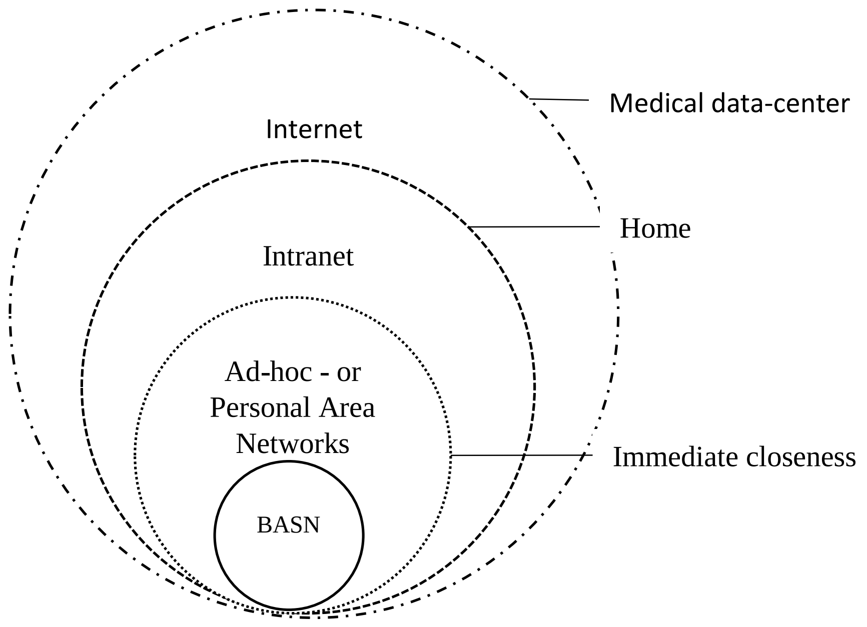
I. Introduction and Background

1 Introduction

Population growth and health projections indicate that many modern societies face situations where the public health sector will not be sufficiently effective in accommodating the growing majority of elderly or ill citizens.
The cost of housing certain patients becomes high as their situation requires continuous monitoring, but their state of recovery is at a level that allows them to stay at home and release the resources of a hospital or elderly home.
The concept of Sensor Information Systems for Assisted Living (SISAL) [2] motivate for the use of composite sensors to monitor blood levels, respiration, location and vital-signs. As sensors are enabled to communicate through network interfaces, they present abilities to form complex images of the situation state. Communication between such sensors could be formed by a Body Area Sensor Network (BASN) [3]. Their heterogeneity and purpose will require different sampling rate quantization and exhibit a wide range of power supply requirements, calibration parameters, and output interfaces.

To prevent recurring engineering costs, each BASN platform would either require significant volume in a single application, or aggregate volume across several applications. This will create design tradeoffs between application specific optimizations and generic programmability. Sampling and data rates are proven to be very variable. A simple blood pressure sensor operates with a sampling rate of 1 sample / minute and a quantization rate of 64 bits / sample, while a ECG sensor operates with 240 samples / second and a total bit-rate of 2.9 – 8.7 Kb/ps. This means that endpoints that retrieve information from these sensors would need to accommodate multiple, disparate, data-streams, occurring in different intervals over single or multiple channels.

Disparate application requirements will call for an ability to aggregate information and integrate BASN systems into existing information technology infrastructure; for example, emergency systems connected to paramedic entities, or pediatrics that perform offline analysis on several days or life-sign measurements. Hanson et al. [3] states that data processing must be hierarchical in order to exploit asymmetry of resources, preserve system efficiency, and ensure that data is available when needed. Systems must detect and react to notable occurrences from incoming data and explicit queries. Such reactions might include heightening the state of awareness, collecting data at a higher frequency for closer inspection, forwarding events to higher levels, or providing immediate response. Data processing performed at the sensor node reveals information specific to the sensor's locality. The bigger picture, however, comes from relationships between data collected from multiple sensors over time.



Sharing and combining information would need to happen at lower levels of the hierarchy, but could require occasional data from higher levels in order to improve classifications.

A sensors value rests, in large part, on its ability to selectively process and deliver information**.** On-node signal processing is required in order to extract information from events that these sensors detect, and processing data at a given rate should consume less power on average than transferring the data wirelessly to an aggregator. This tradeoff will however require the sensor to embed a more complex operating system, which will allow it to concurrently capture, process, and forward information while meeting real-time constraints. Dynamic frequency scaling or power management creates opportunities for lower energy consumption based on situational needs, but requires contextual awareness and predictive models that inform and guide these nodes into correct behavior.

In essence, it could be stated that a central body aggregator is needed because hardware and software needs to inter-operate through multiple levels of infrastructure and share information gained at each level. However, it should not be expected that a primitive sensor node is capable of coordinating such actions, and it can therefore be stated that a more powerful, wearable computer is needed in order to render the vision of pervasive healthcare true.

Preventive measures and early detection is defined as a proactive approach, opposed to today's reactive approach to treatment. Treating diseases that have matured into complications, could have been avoided if certain symptoms were detected at earlier stages. However, the definition of proactivity is defined as change-oriented and self-initiated behavior. Such properties are unpassable without long term, real time monitoring of vital-signs and contextual information. Varshney [4] states that proactivity could be achieved through pervasive healthcare, which he defines as continuous medical monitoring for anyone, anytime, and anywhere. This concept builds on the foundation of contextual, wearable, mobile sensors that aid in detecting vital-sign parameters. Several examples of smart-homes render how specially equipped homes can accommodate senior citizens by embedding wall-powered sensors and central aggregators that communicate wired or wirelessly through a Local Area Network. However, pervasive healthcare is as stated, anywhere and anytime. It implies that the patient under monitor is able to move freely inside or outside of locations with or without necessary infrastructure for network communication. Such freedom could only be rendered true if a central aggregator is within the reach of the sensors.

Physical size and weight of the sensors should be of such proportions that they cause the least interference in life, and the ability to integrate the sensors with existing technology is preferable as fewer specialized systems would need to be maintained.

> *The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. – Mark Weiser.*

Extracting vital sign parameters is however, in isolation, insufficient without the inclusion of context and reflection about the person that is being monitored. No two persons are alike. Gender and age will dictate physical attributes like heart and thorax physiology, and mental attributes will dictate system understanding and acceptance. Persona profiling has been proposed in [4] , but would imply that streams of sensor information is joined with an ever-changing model that could be presented through tagged encoding, object-orientation or logical rules.

Learning what a normal state is poses requirements on memory and historical data-access from single or multiple repositories. However, once normal is understood, then it could be possible to look at vital-sign monitoring from a different perspective by actively searching for deviation in the accepted state, instead of looking for occurrences of complex event patterns.

Becoming minimally intrusive in life could, to some extent, be accomplished by aiding the user in taking suitable decisions on his or her behalf. However, this would require us to find what the user's intent is, and intention is stated to be derived from a person's past and current actions, with the inclusion of location and identity. Finding user intent is synonymous to finding *patterns,* and it can therefore be stated that inclusion of a pattern modelling language could prove beneficial as the complexity of our environments are less than minor.

Pervasive healthcare must, as stated by Varshney, include contextual awareness of our surrounding environment. It implies that our computing devices are aware of physical locations and temporal context. There is profound difference between having a high heart rate while running outside, and having a similar rate at night while lying still in bed. Contextual awareness happens when data is joined and infused with additional parameters. One cannot, however, just increase the amount of data, and one should expect that the user might interact with a monitoring system minutes or hours after an initial query. An answer would then need to be redirected back to a potentially important pattern that the system is trying to detect.

1.1 Problem statement

Being aware of the challenges that persist in sensor assisted living, proves that it would do little or no good to study just another vital-sign sensor without trying to accommodate to some of the challenges. Homogenous, proprietary, and hard-coded applications have little value in this domain, as change, openness, and ease of use should be accommodated. A general overview is presented in [3], [4] and renders that a wearable aggregator should accommodate to multiple, disparate streams.

We imply that a Complex Event Processing (CEP) engine is needed in this context because it abstracts away complexity that regard handling of data streams. Information needs to be joined in real-time and prove tolerance between different architectures and formats by expressing uniform schemas. Reasoning about time, order, and relation comes naturally to these engines, and expressing intent is done in a readable format with queries or rules instead of arbitrary nested object-oriented models. Patterns are usually modelled by a logical language and relate strongly to the need of expressing rules for contextual awareness and self-imitated, proactive monitoring. Previous studies of event processing in relation to home-care and vital-sign monitoring is presented in [5]–[8]. Positive results render that embedding such an engine into a wearable device, could prove useful.

It is stated that this is a novel field of application as CEP vendors are focused on enterprise solutions. Embedded systems have until recent years been restrained in processing capabilities [8], but recent advances and increased computational power makes it feasible to introduce such overhead. Modern smartphone devices prove themselves as viable, mobile aggregators as they retain profound processing, storage, and memory capabilities. Embedding multiple transceivers for cellular, ad-hoc, and GPS communication effectively renders them useful as bridges between multiple sources of information. Fully featured operating systems provide us with modular and sandboxed application environments that tolerate operational faults. Existing ecosystems and development frameworks remove complexities of software development and ensure, to some extent, uniformity.

Previous studies [8], [9] render that smartphone devices can work with Esper, an open sourced, CEP engine. However, results in [9] render low levels of throughput and imply that this combination has substantial overhead. No studies present viable indications of how well Esper works on such smartphone devices as a general benchmark is not present. The only known implementation [10] of Esper for such devices is severely outdated, and deviates from recent releases of the engine.

A new, runtime environment was recently released for Android, and leverages the use of *Ahead-of-Time compilation*, opposed to *Just-in-Time compilation*. This distinction could prove Android more suitable to Esper, and render profoundly higher performance rates.

Our primary intention is to answer whether it proves feasible to utilize an Android based smartphone device in conjunction with Esper, and to ensure that an enablement retains stability and functionality that enable it for purposes that regard the domain of pervasive monitoring.

1.2 Contribution

This thesis aims at producing software that answers and resolves the questions and statements posed in chapter 1.1. A general benchmark that exercises the fundamental aspects of a CEP engine should expose the limitations of Esper in a resource constrained, embedded environment. Such measurements are not only useful to us, but could act as a guideline for others that wish to implement new, or move existing systems in this domain. A smartphone could offload central servers by performing more computations locally. The cost of wireless communication is, in terms of energy consumption, profound opposed to local computations. Smartphone devices are continuously tasked with new, and increasingly more complex use-cases. Abstracting away program logic into human readable queries and rules could improve correctness and impose rapid development cycles. Alternating existing application behavior is considerably easier with the use Esper as it enables runtime reconfiguration without service downtime.

1.3 Methods

An initial requirement analysis is needed in order to render the limitations and possibilities that the existing implementation [10] of Esper imposes. A new enablement of the engine could impose substantial overhead on our work, but prove beneficial if we impose increased levels of stability and functionality.

How Esper should be benchmarked, and what factors should be considered requires an understanding of existing work. Re-using tasks from existing benchmarks could prove beneficial and enable direct comparisons that render the difference between a powerful workstation and a smartphone device.

A design-implementation-evaluation procedure is needed in order to support a set quantitative tests that should aid in an conclusion for this thesis.

1.4 Outline

This thesis is divided into three distinct parts that withhold 9 chapters.

Part I identifies why this thesis is needed and supplies the background information that is required to understand the fundamental concepts of the software and hardware components in this thesis.

- Chapter 1 introduces this thesis and renders some of the most profound challenges in sensor assisted living.
- Chapter 2 introduces the fundamental principles of Complex Event Processing and highlights primitive functions that reside in such systems.
- Chapter 3 introduces Esper and presents its inner workings by highlighting how principles from chapter 2 can be expressed. This chapter aids as a reference for understanding implementations that reside in part II and III.

- Chapter 4 introduces Android and explains the inner workings of the operating system. Effectively supporting design and implementation decisions that reside in part II.

Part II presents the requirement analysis and aspects that regard the design and implementation of our work.

- Chapter 5 presents a requirement analysis that is decomposed into two segments. The first segment is related to the enablement of Esper on Android and outlines the requirements that must be met in order to render Esper operational in our context. The second segment is concerned about tasks and factors related to benchmarking Esper.
- Chapter 6 binds to chapter 5 and describes the how an enablement of Esper is possible on an Android based device.
- Chapter 7 concerns the design and implementation of the benchmark that is utilized for our evaluation of Esper in chapter 9. It includes the implementation of tasks and factors that concern our requirement analysis and principles from chapter 2.

Part III presents the evaluation and conclusion of our work.

- Chapter 8 describes the set of quantitative tests that utilize the tasks and implementations from chapter 7. It highlights influential factors that regard the test and aid in the conclusion of our work.
- Chapter 9 presents the summary of contribution, critical assessment and provides insights of future work.

2 Complex Event Processing

This chapter describes the principles of Complex Event Processing and Data Stream Management (DSM) The first section introduces the domain of such systems and presents a unified classification and terminology to aid in the understanding of Esper and implementation specific details troughout this thesis. The second section describes the fundamental principles of DSM and aids in the understanding of how continuous streams of data can be processed and aggregated. The third section describes fundamental principles that concern CEP and describes how such systems perform operations that can prove beneficial in context of sensor assisted living. Section two and three intend to support our requirement analysis, and provide insight into measurements performed in part III of this thesis. This chapter is concluded by a reflection on certain limitations that could prove utilization of CEP problematic in context of sensor assisted, real-time monitoring

2.1 Domain

Online analysis is needed as fast as possible in scenarios that require reactive behavior. Relational database systems are utilized in applications that require persistent data-storage with infrequent insertion and alteration, but several environments are better modeled as transient data-streams, where information is seen once and never again.

Examples of environments and applications that include such streams are network monitoring systems that analyze incoming data-packets for the purpose of detecting intrusions, telecommunication systems that process call records, industrial entities that rely on the control of events in manufacturing processes, and sensor networks that report occurrences of real-world events.

Developing stream analysis applications is tedious and results in finite and homogeneous properties. Such systems can only process a strictly defined form of data, and analysis is performed according to rules defined by expert architects. Common for all applications however, is the need to:

- Acquire a concept of timeliness and an ability to continuously query incoming information, even if it holds structural variability.
- Process information that comes from multiple, outer edges without the need of persisting it.
- Transform lower level data items and combine them into higher level data items that hold new and perhaps not previously seen information.
- Discard or recycle old data items that are no longer needed.

Information streams diverge from stored relations in several ways and can be identified as unpredictable in size and rate – with imprecision and variable characteristics in format and schema. A stream of information is potentially unbound in size and has open-ended relations. Quantity of information might prove infeasible, or of little value to store and reasoning must occur in-memory. The receiving end will have little or no control over the order and rate in which data items arrive. Reordering, processing, and analytics are therefore dependent on available system resources.

Information can be useless when viewed in isolation. Events that happen in the real world are usually complex because they include information about time, location, and sequence, and can be formed from a correlation of multiple sources. An event could also be identified as an expected occurrence that never happened.

These requirements led to the development of a new class of systems that were specifically designed to process streams of information based on a set of processing rules. More than three decades of contribution from different research communities resulted in generic approaches to real-time information analysis. However, despite having a common goal, it can be stated that each community brought its own view of the problem domain and propositions reflected each community's background. As this thesis relies profoundly on stream processing, we state that a clear classification of such systems and terminology would aid in the understanding of our work.

2.1.1 Classification

Two distinct models are identified in [11], and summarized as follows.

- *Data Stream Management*; *Babcock* [12].
  DSM install continuous, standing queries that actively process and evaluate data items as they arrive. An approximated answer to a continuous query is produced over time, always reflecting the stream of data seen so far. It is not uncommon to see resemblances to traditional, relational databases as these systems require schemas that present structure of incoming data and leverage Structured Query Language (SQL) like operators for selection, aggregation, and joins.

- *Complex Event Processing; Luckham* [13] .
  CEP based systems view incoming data items as events happening in the external world. They are more concerned about relational patterns between data items and their time and order of occurrence. Aggregation occurs by combining and transforming lower level events into higher levels events that produce new answers about the world that is monitored. Origin is closely coupled to Publish/Subscribe systems.

A summary of existing systems that embed such models can be found in [11]. Certain systems, however, classify themselves as CEP, but build around concepts that resemble DSM engines by embedding query languages that contain operators for controlling the information flow. Etzion et.al [14] states that CEP is best understood in the context of how and where its concepts are used and properties from a DSM model can co-exist as a set of features. When looking at issues presented in sensor assisted living and pervasive monitoring, we see that the DSM and CEP models alone, are not sufficiently capable of conforming to all requirements. It is the combined power of both models that presents itself as a viable solution.

2.1.2 Terminology

We intend to present uniformity throughout this thesis and define a collection of phrases, closely bound to [11].

- *Event*
  - Something that happened. An object with attributes that signifies an activity.
- *Data Item*
  - Any kind of data that reflects some knowledge generate by a source, or notifications about events happening in the observed world. Event and data items can sometimes be used interchangeably throughout this thesis, but a data item is represented as an object that a processing engine can interpret while an event is a note of occurrence that can be modeled as a data item.
- *Source*
  - Data items that enter the engine are generated or forwarded by an Information source.
- *Stream*
  - Linearly ordered sequence of events occurring within a time span.
- *Query*
  - A defined set of logical statements that specify how to filter, combine, aggregate, and project information.
- *Engine*
  - A tool that embeds a set of processing rules which describe how incoming streams of information should be processed to timely produce outgoing streams of events to information listeners.
- *Listener*
  - As an engine processes a set of data items according to a set of queries, it outputs the new streams to one or more listeners that act as recipients.

2.2 Principles of Data Stream Management

This section introduces the fundamental principles of DSM and describes how distinct concepts correlate to each other in order to enable DSM true.

2.2.1 Time and Partition

Time can be seen as one of the most profound and fundamental concepts of a processing engine, as it influences temporal reasoning and temporal ordering of data items coming from internal or external sources. Time can, in relation to data items, be presented as a timestamp and Babcock et.al [12] classifies these as :

- *Explicit* – where the information source appends a timestamp to the data item; this timestamp corresponds to a real world event at a particular time as seen from the viewpoint of the source.
- *Implicit* – where the engine itself adds a timestamp to the arriving data item and thereby implies its view about the occurrence of this event or data item.

Explicit timestamps will identify that data items might not arrive in order. In real-time analysis, one should only expect an almost sorted sequence of items. The correlation between an implicit and explicit timestamp is of importance in situations where we want to state something about potential cumulative delay related to commutation between receivers and sensors. However, time becomes less clear when we aggregate or join information. When composing new, higher level information from a collection of lower level data items, we force ourselves to choose what the occurrence of this composition should mean.

A *Context partition* is defined by Etzion [14] as *taking a cloud of event instances and classifying them into one or more sets.* A temporal context will therefore divide events into a set of partitions with their own time span. Spatial context will reason about the partitions in form of location and distance, and a *segmented context* will classify the events by reasoning about their group size. Two contexts can overlap each other in time and allow us to reason about how events are shared between streams or patterns. It implies that an event could expire in one pattern and persist in another, effectively keeping memory allocations from being released. Contextual partitions in our thesis will, for most part, be defined by windows, which we describe in chapter 2.2.3.

2.2.2 Computing Answers

A monotonic continuous query implies that order is preserved and that it suffices to re-evaluate the query and append computed answers by only evaluating arriving data items. However, this proves infeasible, given the characteristics that a continuous stream imposes and requires that the set of data items is re-calculated every time a new item is added. Computation time would increase linearly as the set gains size and response times would no longer be deterministic.

As a stream can potentially be unbound in size, it imposes requirements to limit the scope of what the engine currently sees. A limitation would, however, imply that we will only retrieve approximate answers to the questions that we pose, as the whole data-set is not available. Backtracking over this set of information could prove infeasible as it could imply that the engine would block incoming information and impose performance issues [15] .

An evaluation interval is either defined as *eager*, where we re-compute an answer for every data item that qualifies for our query, or *batched,* where each set of data items are collected and computed once as a group when the query validity expires.

2.2.3 Windows

The limited scope of a stream is called a *Window*, and represents lower and upper bounds that constrain the range of information that is currently seen. Windows can be classified as *time-based (physical),* where the bounds are defined by seconds, minutes or days; or as *count-based (logical)*, where the bounds are dependent on the amount of items in the window. Memory allocation is fixed and predicable for logical windows, but care must be taken for physical windows as dynamic allocation could impose resource depletion. Classification of such windows is further influenced by the way bounds are moved, and results in definitions presented in the following chapters.

2.2.3.1 Fixed and Landmark

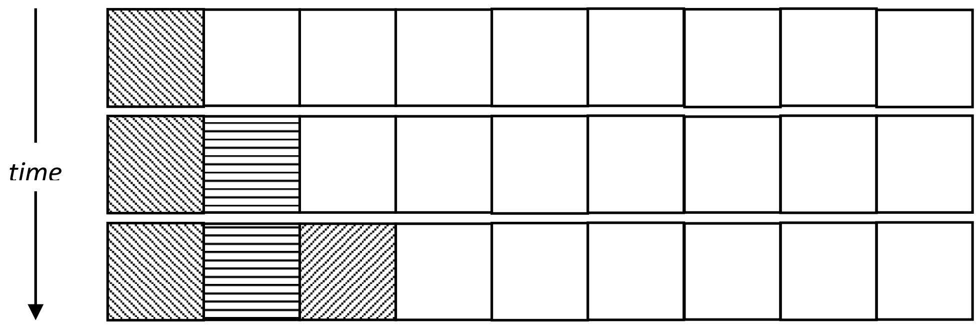


Figure 2.2.3-i Landmark window

A *Fixed Window* will not move its bounds at all and can be used to limit the scope of a stream between two distinct periods of time *t,* that form an interval [$t_{\text{start}}, t_{\text{end}}$]. An implicitly time-stamped event $Event_a$, will be included in the window if and only if $t_{\text{start}} \leq Event_a < t_{\text{end}}$. A *landmark window* has a fixed lower bound and a moving upper bound that accepts any incoming data items until the window is destroyed. The potential danger of such a window is that it, without additional constraints, could grow into infinite sizes. However, it proves itself useful in scenarios related to pattern detection where we know little about the environment.

2.2.3.2 Sliding

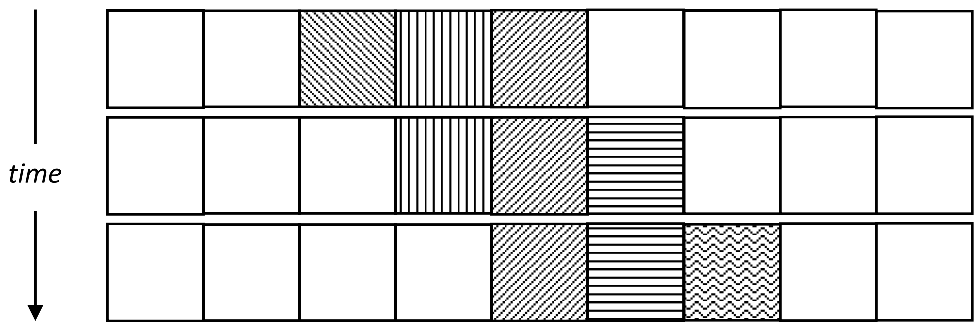


Figure 2.2.3-ii Sliding window

Sliding Windows have lower and upper bounds that dynamically advance as new data items are observed by the engine. Given that the window has a specified length *n*, then an instance of $Event_a$ will stay within the window bounds for an interval defined as $0 \leq unit \leq n$ is true. A unit could either be presented physically or logically and would imply that the data item is discharged as soon as its validity expires. A variation of this window is identified as *Tumbling*, where event instances are processed in batches upon interval, when an interval expires.

2.2.3.3 Jumping

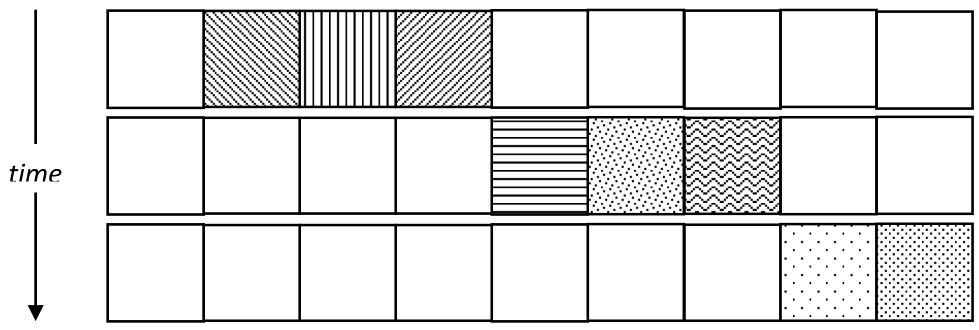


Figure 2.2.3-iii Jumping window

A Jumping Window will fill up to a size *n* before it releases all contained data items in a bulk operation. The main distinction from a sliding window is that each window will never contain a data item that could have resided the window before it. It implies that any operation that is performed on the point of expiration of this window is performed on a set of data items that have not been previously seen by the engine.

2.2.4 Selection and Aggregation

Selectivity is defined as an ability to output a set of data items that match certain criteria. *Projection,* however, is defined as an ability to extract attributes from data items. The notable distinction is that selectivity delivers references to where a set of data items can be collected, and projection must reflect over each data item and construct new data items that contain only a sub-set of the attributes that were seen. This, however, enables us to control what is seen and processed by other temporal contexts.

Selectivity does not keep state, and would therefore not block query execution. Projection, however, needs to embed a reflection strategy that would impose it to identify attributes by names or indexes. It implies that query execution is blocked until the appropriate identifier is located and its value is derived.

Aggregation functions are defined as computations related to counting, summarizing, or averaging. Such functions are unable to produce a correct answer until the whole input is seen, and implies that they must be utilized within the bounds of a window. Answers are reflected by data items seen so far and considered approximate.

The computational cost of performing aggregations is identified by Marques et.al [16] and implies that the summarization of a set of values can be computed at a fixed cost regardless of window size or type. Locating the maximum value of the same set is considered distributive and can only be computed at a fixed cost for jumping windows. The median of the set is holistic and implies that cost will depend on window size. Any nested combination of these functions will therefore affect our computing ability.

2.3 Principles of Event Processing

This section introduces the fundamental principles of CEP and describes properties that motivates the use of CEP in relation to sensor assisted living.

2.3.1 Filtering

Our definition of filtering is related to removal of irrelevant data items at certain stages of processing. It is considered as a fast and small computational step, but proves significant for the whole chain of processing, as it dictates the amount of data items currently seen by a pattern, function, or window.

Many irrelevant events must get through initial input channels and should not be lost as they might be useful or relevant for certain queries. An engine must therefore accept an initially high throughput of events. Filtering criteria depends either on goals set by processing rules, where we discard a data item because it contains attributes that do not match our interests or by window bounds or temporal contexts that render the data item obsolete.

2.3.2 Pattern Detection

Detection of a pattern is defined as finding one or more events that match a given *pattern signature* [14] . Each pattern will reside in one or more partitions that are defined by time or space, which means that several installed patterns can be activated or discharged based on context. It implies that an engine should handle administration of numerous patterns, but that allocation of computational resources will be decided by the amount of active patterns. Signature complexity and event occurrence rate can, to a large extend, drive the computational requirements of an installed pattern. A signature is defined by:

- *Occurrence; W*here we state that a pattern is rendered true if all, any, or no events present themselves.
- *Threshold assertion; R*elates to counting and specifying that a pattern is true if *n* occurrences of event *e* is present.
- *Order and sequence; W*here *order* indicates that occurrence of event $Event_a$ is followed by an occurrence of $Event_b$, where the implicit or explicit timestamp of $Event_a > Event_b$. *Sequence*, however, is defined as the presence of one or more instances of $Event_a$, followed by one or more instances of $Event_b$ within a logical or physical interval.
- *Parameters;* Imply any additional values, ranges, or limits that can be signified by variables or constants
- *Dimension;* Relates to time, space, or a combination of both. Temporal context will relate to sequence and spatial context will relate to distance.

Additionally, it is stated that a pattern embeds a set of *policies* that include:

- *Evaluation* - which is classified as *Immediate* if the pattern is tested every time a new qualifying event enters its domain, or as *differed* if the pattern is tested incrementally at the end of a temporal context. Evaluation is closely bound to output generation, and computational impact is signified by the strategy that is chosen.
- *Cardinality* – controls the amount of matching sets generated and outputted in a given partition. A *single* - policy implies that only one matching set is generated. A *bounded* – policy implies that *n* matching sets can be generated within a context partition until an upper bound is satisfied. An u*nrestricted* policy will imply that the pattern should output everything, every time.

The correlation between evaluation and cardinality policies will affect how well a processing engine handles the overall throughput of incoming events. There is a clear distinction between only testing if a pattern renders true and outputting results. Given that a contextual partition is not ended, the outputting of a set of matching data items would imply that the engine has to copy some or all of the correlated data items into a new collection that is transferred to the receiving part. Failing to do so could invalidate the current pattern because a single data item could be altered by another observing query.

Patterns can, on a higher level, be classified as:

Basic – where they include only logical operators, thresholds, or selections and can be utilized in a temporal context, but does not depend on time. For example, given that temperature sensor reading is initially stating 37.0 ^{o}C, if the next 5 readings are higher than 37.0 ^{o}C, and each reading is followed by a reading that is increasing in value, then the pattern indicates a trend towards a fewer.

Dimensional – relates to temporal and spatial contexts, thereby concerning itself with location and time in addition to logic and thresholds. For example, given that medication is consumed at time t_1, if medication is consumed again at t_2 and $t_2 < t_1 + 4$ hours, then an alert of excessive use should be invoked.

2.3.3 Immutability

Etzion et.al [14] states that an event should never be altered. As an event signifies an activity, it becomes questionable what activity the event signifies and it could lose event causality, which Luckham [13] defines as designing the fact that an occurrence of event $Event_a$ caused the occurrence of $Event_b$. The event will lose significance as traceability becomes an issue of uncertainty.

Alteration or enrichment of an event must be performed on a copy of the event. Significance is maintained if and only if traceability to the original event is kept. Estimation of memory requirements must therefore consider that transformation of data items could impose higher allocation requirements.

2.3.4 Event Hierarchies

A complex event is usually an aggregation of a set of events. Occurrence of a pattern might incur an action that results in the creation of new, lower level events that contain properties of the pattern that was detected and a summary of important information that happened. Transforming multiple lower level events into a higher level event might be more meaningful and result in lower output cardinality.

An example could be stated as detecting a series of movements inside a home. Imagine that a person moves into the kitchen, turns on the stove, moves out of the kitchen, and does not return to the kitchen within an interval of four hours. A pattern should aggregate the movements into a higher level alert concerning a possible fire hazard in the kitchen. The set of lower level movements and actions can now be discarded and system resources can be de-allocated.

This implies that events can be divided into multiple layers, each with its own set of abstractions and rules. It allows us to reason about events in isolation and reduces the number of events that need to be taken into consideration.

2.4 Limitations and Challenges

We wish to highlight certain challenges in event processing that could relate to sensor assisted living. The uniform relation between these challenges is that they impose the system architect to make domain specific decisions that may reside inside or outside of a processing engine. Our intention is not accommodate these challenges as they are considered out of scope for this thesis, but to rather inform the reader about considerations that must be taken if such a system is to be further developed.

2.4.1 Temporal Issues

- *Time intervals*
  As occurrence of an event is often defined by a single point in time and represented by a timestamp. Certain scenarios however, require us to reason about transition of states over time. An example of this can relate back to our *basic*-pattern in chapter 2.3.2, where consecutive temperature readings should indicate presence of a fewer. Given that a fewer was actually detected, then one could argue that the occurrence of the fewer could either be represented by the first reading, the last reading – or as the complete duration of all readings, as an interval. Occurrence of time becomes ambiguous and pattern signatures must be explicitly defined with derived timestamps that indicate the start and end of the interval. A generic approach, is to our knowledge still considered an open issue.
- *Order and time synchronization*
  Communication latency and packet loss could impose cumulative delay on retrieval of events from a sensor. As this sensor struggles to re-send old events, the receiving engine still works with events that could be tens of seconds old – not reflecting the emergency situation at hand. The engine could identify this issue by comparing It's implicit timestamp against the

events explicit timestamp. This implies however that the sensor utilizes an internal clock - and deviation between the time of this clock and the receivers clock should be synchronized as enough inaccuracy would potentially disturb reason of order. A clock drift of 3.5 microseconds every hour would impose a deviation of a whole second after approximately three days. Clock synchronization could, as proposed by Lamport [17], either be achieved by communicating with a specific time server, or by establishing a logical sense of order by exchanging a set of collectively produced messages between participating sensors and receivers. This would however imply that a central organ has some notion of control over the sensors and that the sensors themselves are willing to cooperate.

2.4.2 Uncertainty

Uncertainty can occur when inexact information about an event occurs in the system. It implies that event content has inconsistent attributes that falsely present what really happened. The cause of such occurrence could be related to unreliable or imprecise sensor that might be miss-calibrated. The sensor might fail to report a sub-set of events and certain, installed patterns might never fire because a single missing event prevents the engine from seeing the whole picture. A different angle on this issue would state that the engine could see certain event occur in the wrong order and time and produce false positives that propagate through the system, placing inexact statements that are observed by queries and patterns. Tracing back and re-producing such errors could become an immersive task. [14] proposes a set of solutions that include probability based methods like Bayesian networks, evidential reasoning or fuzzy logic. Incorporating such mechanisms in a true medical monitoring system, renders the truth about potential complexities that must be taken into consideration.

3 Esper

This chapter describes the inner workings of Esper. It starts by a discussion of why Esper fits within the domain of Android and provides a general introduction to the engine. It then follows by presenting a higher level architecture and describes, on a fundamental level, how event processing is performed and how Esper operates within an application. This chapter includes references to all query operators that are utilized in this thesis, and describes how fundamental principles from DSM and CEP are expressed. It concludes with a notion of how Esper can join information from external sources.

3.1 Domain

Existing solutions are extensively surveyed by Cugola et.al in [11]. Several engines are derived from academic research projects - and we fear to some extend that they already are – or in near future will be, discontinued. We state that our work could prove to be of more value for an engine that is actively maintained in an open-source community as more people could potentially relate to our results and future engine development is motivated by productivity and solidity. Existing engines require that their surrounding runtime environment is compatible. It implies that the embedded system is able to interpret the programming language in which the engine is developed in - and that the engine is compiled for the right architecture. A majority of todays smartphone devices operate with *ARM*-based processors, access to source code is therefore necessary in order to perform platform specific compilation and implies that any library dependencies must be gracefully supported and resolved. We re-visit this issue in chapter 6.1*,* where we explain how our environment proves hostile for many of the existing engines. We imply that *Esper* conforms to such requirements - and is selected for this thesis because of its component based architecture, openness, lightweightness and probability of successful incorporation with a smartphone based operating environment. We are currently not able to find equal characteristics in other engines, and Esper is classified as state of the art in its category [11], with positive results from work conducted by others [5], [8].

Esper is a commercially open sourced, event processing engine with support for projects that embed the *Java* or *.NET* programming environment[1]. It distinguishes itself as a component based engine that is not installed and executed as a stand-alone application, but rather implemented in an existing application as a library. This distinction disables the need for inter-process message passing or socket communication - and enables the programmer to control how the engine retrieves and outputs its data items. It implies that the programmer is in full control of the information flow, and must explicitly choose strategies for displacement of data, if throughput becomes too high. All computations are performed in-memory – and *Esper* is, with additional libraries, measured to only allocate 4.5 MB of heap

[1] Our concerns and statements are, throughout this thesis, only related to Java.

memory. We imply that Esper can be identified as a hybrid engine as it presents extensive concepts from the data stream management model, and embeds a rich pattern expression language with operators that can relate to event processing.

Esper was initially released in 2006, and retains to this date, continuous release cycles from an active community.

3.2 Architecture

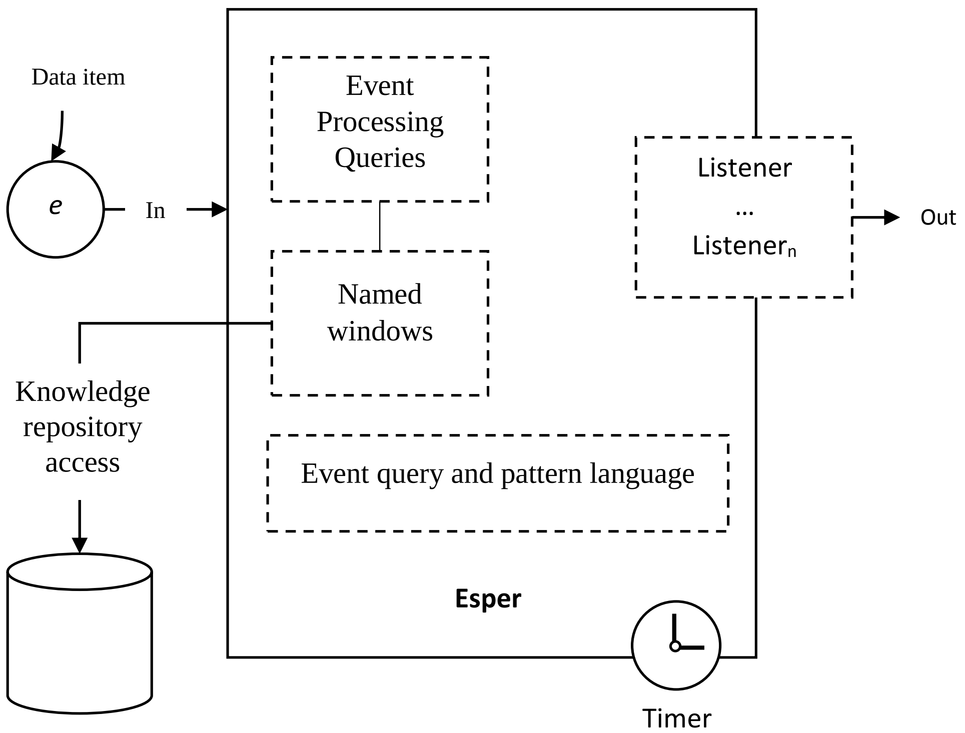


Figure 3-2.4.2-i Simplified, architectural overview of Esper [18]

Figure 3-2.4.2-i represent a higher level overview of *Esper* and external components that work together to form a complete processing environment and these components are hereby described as:

- *Event*: *e*
  An event is collected from an external source and pushed into to the engine as a data item by an explicit processing thread. This can be seen as a de-multiplexing operation in an environment where sources are numerous. It is implied that the schema of *e* is described to the engine before the engine accepts *e,* and a unique name for *e* must be provided.
- *Event query and pattern language*
  Is a component that translates and registers string based queries into rules

that are installed as event processing statements. Each statement is bound to one or more listeners that retrieves output generated if the statement resolves to true.

- *Event processing query*
  Are defined as the collection of active queries and patterns that observe incoming events and notifies listeners. Statements will place qualifying events into named windows and accumulate any additional processing rules over time - as new events enter the engine.
- *Listener*
  One or more listeners can apply interest in one or more statements. As a statement evaluates to true, it delivers a set of data items that the listener can reason about. A listener implements a specific interface that enables the engine to communicate with it – but a listener is explicitly defined as a component in *Java*, outside of the engine - and acts as a multiplexer.
- *Named window*
  A named window is a window of any type described in *chapter 2.2.3.* The window name is unique, and takes, by default, the event name.
- *Knowledge repository*
  Is defined as external persistent repositories that could be represented as relational databases. A window can join information from repositories to enrich what the engine knows about the current stream of information.
- *Timer*
  The timer is defined as an internal clock that operates in its own thread and acts as a guideline to the lifetime of windows and other operations that require temporal constraints. Timer is, to our knowledge derived from the underlying operating system - and operates by default, with a 100 millisecond resolution.

An application that embeds *Esper*, can employ one or more engine instances – each separately configured. Each instance will internally share resources between standing queries by constructing a delta-network of data items that only communicate occurrences of change among themselves. Two queries that declare or reference the same named window – will only share a single view to it and inverted indexes are utilized in order to match one or more queries against an incoming data item.

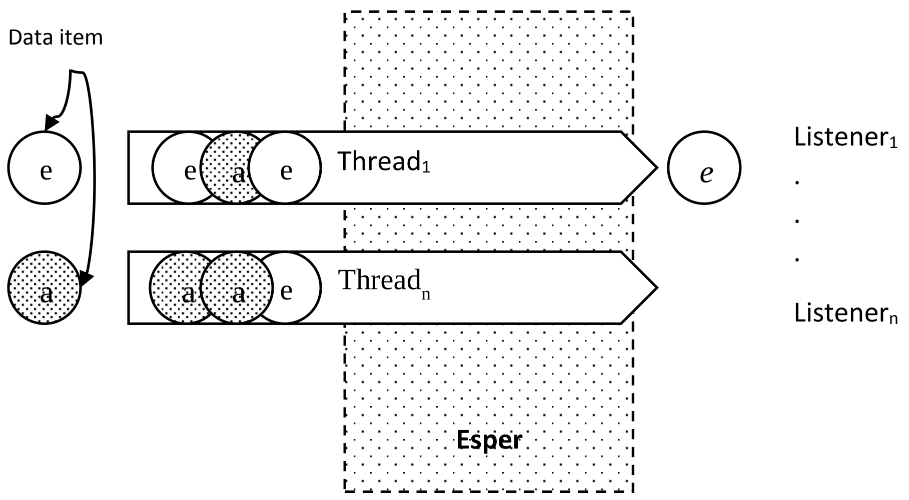


Figure 2.4.2-ii Processing model

Esper is, as stated - a component that one would implement within an application. This implies, by default, that it is our responsibility to manage how input, output and concurrency is defined. Any thread that invokes the *sendEvent(data item)* – function - will be responsible for the whole chain of processing until the data item is either discarded or delivered to a listener that has subscribed an interest for this particular output. This implies that the thread is occupied until all relevant queries are evaluated and payload is projected. This model will however present us with a sense of freedom as any thread can push and process any data item and thereby, event. It implies that a pre-allocated thread-pool of workers could be utilized in order to leverage the potential of underlying hardware.

Esper will under most circumstances, not copy or clone an arriving data item. We verified this by tracing the *sendEvent*() call and found that each data item is simply wrapped around a container object. A data item copy is, to our knowledge, only performed when a data item is altered by a query to ensure consistency across different hierarchies[2]. This operation imposes expenses that are not taken under consideration in this thesis, but imply that high frequencies of alteration could impose a considerable bottleneck.

All communication with an engine, happens through public interfaces. Fine lock granularity on context partitions enable multiple threads to perform query alteration and stateful event processing concurrently. An engine will utilize a latch system to ensure that event causality is preserved from the viewpoint of receiving listeners. It implies that if two threads work on the same query and produce results that can be ordered by an attribute – then the engine will ensure that both threads are finished with their processing before any results are delivered to awaiting listeners. All execution that concerns *Esper*, takes place in the process that embeds the engine.

[2] See [19], section 5.21.1

Other processing models are available. *Esper* can itself, maintain a work queue and thread-pool. This however - implies that data item handoff is done asynchronously and that we lose notion of how much we are currently processing and how much our work-queues are accumulating, as they are not referable from an external viewpoint.

Extension of the engine is possible. User defined functions can be called from any query and a pluggable architecture ensures query extensibility and computability with previously unseen data-formats.

Esper also depends on a set of external libraries that must be included. Table 3.2-i summarize these libraries, and state their purpose.

Library	**Description**
ANTLR	Another Tool for Language Recognition. A parser and generator for reading or executing structured text and binary files. Utilized to construct languages and tools by generating walkable parse trees from existing grammars. Dependent upon in *Esper* for query interpretation.
CGlib	*Byte Code Generation Library. A Java* specific code generation and transformation library that enables creation and transformation of classes and functions at runtime, without the need or re-compilation. Dependent upon in *Esper* for alteration and enrichment of objects that reside in the application.
Xerces	A processor for parsing, serializing and alternating information presented in Extensible Markup Language (XML). Utilized in *Esper* for both configuration management and event presentation.
Commons logging	Logging *API*. Implements a set of adapters that act as bridges between different logging frameworks. Utilized extensively throughout the source code of *Esper* for error and occurrence reporting.

Table 3.2-i Overview of external libraries in Esper

3.3 Event Representation

Event schema is presented to the engine in form of a unique name, a set of attributes and the data type of each attribute. This schema must be defined beforehand, but can be provided dynamically during runtime - and must be presented in one of the following formats:

- Plain object
- Object-array.
- Key-value map
- XML

A plain example of an object-array that describes a temperature reading can be presented as:

Name	*Timestamp*	*Value*	*Location*
Type	*Long*	*Double*	*String*

Arbitrary nested attributes are allowed and takes on an object-oriented style when referenced. Any incoming event of the name "Temperature" can now be queried from a window with the same name. This implies that an existing or dynamically created *Java* based classes can be added to the engine without any additional specifications and that legacy systems can adopt *Esper* with more ease. Schema definition aids in correctness and optimization of the engine. Attribute lookup is expensive as it requires reflective properties and *Esper* has to some part resolved this by indexing attribute names and accessing them through index positions.

3.4 Event Processing Language

Esper is instructed through an expressive, textual language that derives many properties from the *SQL* standard. A complete overview can be found in [19] - our intention however, is to only describe operators utilized in this thesis. Brackets *'[ ]'* signify that the operator is optional and the ordinal order of the operators presented in Table 3.4-i, holds true when constructing actual queries. All grammar interpretation is as noted, conducted by *ANTLR* and based on *Extended Backus-Naur Form*. Temporal statements are modeled using *Allen's interval algebra* [19].

Operator	**Description**
[*INSERT INTO window*]	Signifies insertion of data items into a named window based on selected and projected attributes defined by the SELECT operator below. It implies that we can query one stream and insert only aggregated values into a separate stream that retains a different set of queries.
SELECT *attribute* [AS name], [*f(a)*], [,...]	Performs selection, projection and any functional operations that specify how the resulting output should render. [*f(a)*] signifies an aggregate - or custom formed function that takes one or more parameters. [AS name] re-labels the attribute by desire and enables us to reference the projected output of this attribute within this – or any other query or listener that observes the output of this stream. Replacing '*attribute*' with a '*' signifies that we don't want the engine to perform projection and instead present a reference to the complete underlying data item.

FROM *window* [AS name] [,...]	Implies that the query should observe one or more named windows, where each window contains events that are labeled by *name*
[WHERE *condition [AND/OR …]*]	Signifies any restrictive condition(s) [!,=,<, >= ..] that must be met in order for this query to produce results. Each additional condition must be separated by a logical AND/OR operator
[GROUP BY *expression*]	Grouping can be performed on accumulated sets of data items. This implies that the query is evaluated in a batched process and outgoing data items that contain the same *expression*, will be clustered together.
[OUTPUT *[[all | first | last | snapshot*] every *time period*]]	Provides output granularity by specifically stating how often output should occur, even if the query embeds eager evaluation.

Table 3.4-i Overview of general query operators

Window types described in chapter 2.2.3, are represented in Table 3.4-ii.

Operator	**Description**
WIN:TIME(unit[3])	Represents a physical sliding, eagerly evaluated window that expires every *t* units.
WIN:LENGTH(*n*)	Represents a logical sliding, eagerly evaluated window of logical length *n*
WIN:LENGTH_BATCH(*n*)	Represents a logical tumbling window that expires when *n* data items are present

Table 3.4-ii Overview of query operators related to windows

A join between two windows can be conducted within the bounds of the *FROM* operator with the inclusion of a join condition in the *WHERE* clause. An example of this can be presented as:

> *FROM $Event_a$ [unidirectional], $Event_b$*
> *WHERE $Event_a$.timestamp = $Event_b$.timestamp*

We thereby state that only events (*$Event_a$, $Event_b$*) - who has the same implicit timestamp should be considered for further evaluation. If the term '*unidirectional*' is used before one of the event definitions, then we explicitly state that the join should only be evaluated on the arrival of that event. Additionally, outer, left and right joins can be performed between distinct windows.

[3] A *unit* represents a physical value that ranges from milliseconds to years.

3.5 Patterns

Pattern statements are based on the *Rapide* [20] pattern language and modeled with operators that describe order or logic, and atoms that concern themselves with filters and temporal contexts. Patterns are in *Esper*, based on state machines that contain a set of dynamic state trees with branches that nest arbitrary deep. A generic query is presented below and each operator is described in Table 3.5-i.

*SELECT **
FROM PATTERN
[
[EVERY] [variable$_a$ =] Event$_a$
->/and/or
[EVERY] [variable$_b$ =] Event$_b$ [(threshold assertion [, AND/OR assertion ..])]
[WHERE dimension [AND/OR dimension] [,..]]
]

Operator	Description
EVERY	Indicates that the pattern should re-started every time it encounters a qualifying *Event$_x$*. If the operator is omitted, then, the engine will stop looking for new occurrences of *Event$_x$* once the pattern evaluates to true.
Variable$_x$ =	Denotes that we will bind an *Event$_x$* to a named variable. This enables us to reference the event throughout the query.
Event$_x$	Denotes any event or data item that the engine recognizes. Enables us to reference relations between two distinct events that may or may not correlate.
->/and/or	An "arrow" can be identified as the phrase "*followed-by*". It indicates, in our context, that the engine should look for another occurrence of a named Event$_x$ within some specified constraints. Alternatively, it is possible to specify logical relations with the *and / or* operators.
Event$_x$(threshold assertion)	Denotes attribute references and threshold assertions that we impose one the event. Enables filtering by specifying properties that we want our event to have in order to classify validity in our pattern. Other events

	can be referenced if, and only if, they are bound to a named variable.
WHERE dimension	A where-clause holds a dimensional constraint that is in *Esper*, defined by a temporal *timer* that expires the validity of the pattern and is throughout our thesis defined as : ▪ :within(*t*) ▪ :withinmax(*t, n*) Where *t* designates time expressed in granularity from milliseconds to days. *'within'* indicates that occurrence should happen within some time *t. 'withinmax'* indicates that occurrence can happen *n* times or within some time *t*. Decided by whichever occurs first.

Table 3.5-i Overview of pattern operators

As an example, given that we observe a user with cognitive disability and we want to ensure that the user is not exposing himself to excessive medication intake. Assuming that we follow the rules stated in *chapter 2.3.2*, where we exemplified a *dimensional pattern*. Excessive medication usage would then be classified as taking the same medication twice during a time interval that lasts 4 hours.

Given that an event *Medication* has an attribute *name* that acts as an unique identifier, then a pattern could be modeled in the following way:

```
SELECT *
FROM PATTERN
[
        EVERY a = Medication -> b = Medication( name = a.name )
        WHERE timer:within( 4 hours )
]
```

Assuming that this order of event occurrence is true:

1. Medication [name = 'Amoxicillin', time = 12:05]
2. Medication [name = 'Norvasc', time = 12:10]
3. Medication [name = 'Amoxicillin', time = 13:30]
4. Medication [name = 'Amoxicillin', time = 15:00]

It is then implied that pattern would fire twice. Once when medication *#1* and *#3* occurs - and once again when *#4* occurs. It should be noted that when *#4* occurs – then, it will only be matched against the occurrence of *#3*. If we want the engine to match against both *#1* and *#3* – then we have to rewrite our query to match every followed-by event:

```
EVERY a=Medication -> EVERY b=Medication
```

This will imply that *#1* is retained and will not expire until 4 hours have passed. Our listeners will retrieve a set of all three event occurrences.

3.6 Knowledge Repository

Information from external sources can be requested into an existing window or pattern by explicitly referencing it in a query. This enables us to include any persisted, historical information that could not be retained in memory or to retrieve information that is not accessible in our domain of operation because it might reside on a geographically dispersed server.

Integration with a relational database requires that we provide *Esper* with a *Java Database Connectivity (JDBC) driver*[4] that supports the specific database we attempt to communicate with. *SQL* statements are not inspected by the engine and implies that vendor specific queries are allowed. However, each statement is sent as a prepared statement with prefixed variable designations and implies that the *JDBC* driver must support this feature.

A query that embeds relational database access can be identified by its reference in the *FROM*-clause:

SELECT Event$_a$.attribute, db.attribute [,..]
FROM Event$_a$, sql:database-name
[
        "SELECT … FROM table WHERE table.attribute = ${ Event$_a$.attribute}"
] AS db, [,..]

The above query will, for every occurrence of some named *Event$_a$,* perform a lookup in a relational database denoted by *database-name* and – in this example, perform a *SQL* based *SELECT*-statement with a parameter from *Event$_a$*, denoted by '${..}'. Whatever the database returns, will be reflect in the outer *SELECT*-statement of our query. It is also possible to twist the roles around and add a *WHERE*-clause to the outer query – thereby only selecting events that have some properties that correspond with content in the relational database. It would however imply that a numerous set of rows from the database would have to be transferred to the engine and inspected separately.

Esper also provides a function for optimizing database communication by caching common results. It is identified as *result caching* - and implements a *Least Recently Used (LRU)* cache of size *n* and an optional expiration time. Index keys are defined by query parameter values, If *m* rows are returned for a given value, then all *m* rows will retain only a single slot in the cache.

[4] Java based, data access API that defines how a client may access, query and alter a database.

4 Android

This chapter describes the Android operating system and aims to present the necessary background information to support our design, implementation and evaluation chapters. The first section provides an in-depth look at the operating internals of the system, and starts by describing how the operating system differentiates itself from traditional Linux based systems. It then describes the programming environment utilized to construct applications and services, with a notion on how it deviates from existing desktop environments. It follows by an in-depth look at the virtual runtime environment and describes both Dalvik and ART. It then concludes by describing how memory is managed between threads and processes. The second section concerns itself with application development and describes how autonomous components can bind together to form systems that fit within the domain of sensor assisted living. It concludes with a notion of certain limitations that concern the lifecycle of a process.

4.1 Domain

Android is an open sourced, *Linux* based operating system, currently maintained by *Google* and *The Open Handset Alliance*[5].

The majority of devices that currently run on *Android,* are mobile cellphone devices, but the operating environment is embeddable on any system that target a 32 or 64-bit *ARM, x86* or *MIPS* processor architecture.

Our main motivation for using *Android* in this thesis, is based upon the notion that Android is the basis for more than 900 million devices , and implies that the operating environment is tested to prove itself as a stable and viable platform.

We imply that the target smartphone device that recent releases of *Android* aims to support, meets the specifications presented below.

Capacitive 2.5 inch, 100 dpi, 4:3 aspect ratio display.
Support for 802.11x, NFC or Bluetooth.
512 – 2048 MB of SDRAM.
Single / quad - core 1.6 GHz *ARM* baseband processor.
USB connectivity to external sources.
GPS connectivity.
Internal sensors for sound, light and tilt.
8 - 64 GB of internal storage.
Average battery capacity of 1750 - 2500 mW

[5] See http://source.android.com for more information.

4.2 Operating Environment

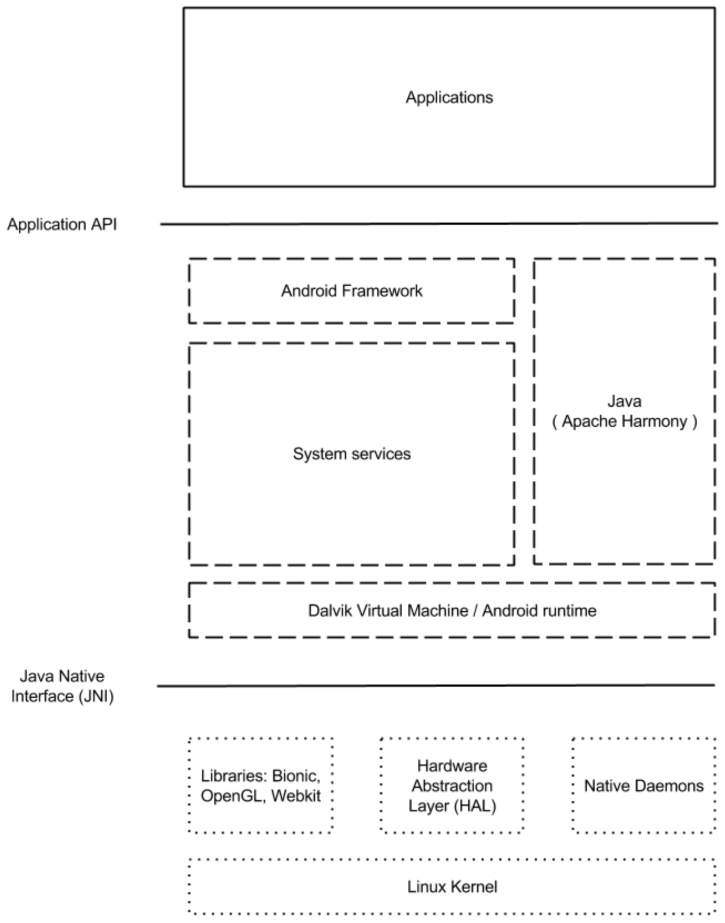

Applications
Application API
Android Framework
System services
Java
(Apache Harmony)
Dalvik Virtual Machine / Android runtime
Java Native Interface (JNI)
Libraries: Bionic, OpenGL, Webkit
Hardware Abstraction Layer (HAL)
Native Daemons
Linux Kernel

Figure 2.4.2-i Architectural overview of Android [21]

4.2.1 Kernel

Android embeds a custom *Linux kernel* that deviates from the mainstream revision as requirements for a smartphone device differentiate from a wall powered, desktop computer. Noting a majority of these differences is important for the overall understanding of the operating system. A summarization from [21] is therefore presented in the following note.

- Access to hardware components like radio transceivers is restricted on per application basis. Each application must request access by interacting and quoting the user for confirmation. Any automatic detection and interaction with nearby devices is therefore depended on some initial user interaction.
- It becomes of the systems best interest to place the device in a sleeping state where most hardware components are deactivated as we operate on energy constrained devices. A driver or application can however request *wakelocks* that enable them to continue operation, even if the device itself is not actively used.
- The kernel is more sensitive to memory allocation and weeds out any lower priority applications that are currently not in the foreground of user

interaction if memory availability becomes low. It relies on the principle that smartphone applications are small and temporary tasks that fulfill specific goals and are teared down as soon as they lose focus. It should be noted that any non-vital process is subjected to removal and that the environment for survival is hostile.

- The kernel does not support the full set of the *GNU C library (glibc)* which most *Linux* distributions use. Instead, *Android* utilizes its own library, called *Bionic*. The result of this - is that libraries and applications that are written for distributions based on *Linux*, might not compile or run on *Android*. *Bionic* was introduced to address issues with speed, size and licensing compared to *glibc*. The library utilizes its own thread programming interface and differs from the *Native Posix Threads Library*. Effectively rendering several POSIX features obsolete.

Each application has access to system resources on the principle of least-privilege. By default, root-access to kernel functions are not given to any components, and users are themselves not allowed to grant such access, even temporarily.

4.2.2 Programming Environment

A variation of *Java* is utilized as the higher-level application programming language in *Android*. *Java* can be classified as an interpreted language that is compiled into architecture independent byte-code and executed on a runtime that translates each program instruction to the underlying plattform. Our interpretation of *Java* - is that it consists of a language, a compiler, a set of support libraries and a virtual runtime environment.

The official Sun/Oracle based *Java Development Kit (JDK)*[6] retains widespread adaptation, but *Android's* version of *Java* can for most part - only be compared on semantical aspects of the language. *Android* utilizes its own compiler and runtime environment as we describe in chapter 4.2.3. Support libraries that provide higher level abstractions to concurrency, data-structures, I/O or graphics are extracted from the *Apache Harmony Project (HAP)* [21], [22] . It is important to understand that the libraries derived from *HAP* are alternated to fit the constrains, and use cases of smartphone devices. Some libraries are completely removed, as their purpose does not fit in the context of mobile computing.

Android also presents a *Native Development Toolkit (NDK)* that enables developers to interact with *C* or *C++* through the *Java Native Interface (JNI).* Any existing lower level libraries or drivers would need conforment with *Bionic*, and access to lower levels of the system, is prohibited [21]. *NDK* is in itself, utilized for cross-platform compilation of *C* code to specific architectures.

It cannot be understated that *Java* maintains importance in the system, and that any extension of functionality must be coordinated between lower levels of the system if any higher-level applications should leverage from it.

[6] See www.java.com for more information

4.2.3 Runtime Environment

We note in the introduction of this chapter that *Android* supports a variety of different architectures. All *Java* based code must be translated to instructions that the underlying architecture can understand and such interpretation is performed by a register based virtual machine that in *Android* is *identified* as *Dalvik.*

Code compilation will not result in traditional *.class* files that mainstream *Java* compilers produce. Androids runtime environment is instructed to interpret DEX (Dalvik Executable) files that must be translated upon compilation before it is presented to the virtual machine. This custom format provides optimizations that relate to repetition minimization, shared constant pools and lower memory footprints [23].

Dalvik was designed for devices that embedded a 250-500 MHz processor and approximately 20-40 MB of application allocatable memory. The overhead of runtime interpretation is acceptable because performance critical libraries related to graphics and networking are natively compiled for each device. Typically, only one third of the time is spent in the interpreter [24].

Performance however, becomes a problem for compute intensive applications that operate with tasks related to sorting, transformation and structural traversal. Optimization for such tasks is performed by a *trace-based Just-in-Time compiler (JIT)* that caches commonly used blocks of code in order to prevent reoccurring translation.

It should be noted that *Dalvik* utilizes a *method-based JIT* if the device is connected to a power source. This is however neglected in this thesis because our interests lies within the domain of pervasive computing and mobility.

JIT-compilation is performed by profiling hot execution paths that are commonly visited. Compiled fragments of code are chained together and placed into a per-process Translation Cache. Return of performance is rapid because of low level granularity and tight integration with the interpreter. It also implies that simple loop detection and register promotions are possible without the need to respect method call boundaries. Trace based profiling does however imply smaller optimization windows as whole methods are not analyzed and synchronization with the interpreter must be performed frequently. Profiling is performed on every run of an application - and implies that knowledge about hot areas are lost upon process termination [24].

The initial version of *Android* was released in 2008. Smartphone devices have since then experienced a multiple magnitudes of increase in processing capabilities and 4-times increase in memory. Operating internals of *Dalvik* have proven insufficient for the use-cases that modern smartphone applications encounter as they move towards a trend of becoming our primary, personal computers.

Android was, in the third quarter of 2013, enriched with a preview version of a new runtime environment identified as *Android Runtime (ART)*. The main deviation from *Dalvik*, is that *ART* is based on *Ahead-of-Time compilation (AOT).* It implies that an application is fully compiled into architecturally specific byte code once, at installation time and that runtime translation is not needed.

ART has proven itself as a viable replacement for *Dalvik* in applications that require extensive processing capabilities. It implies that the CPU can process its tasks faster and return to a lower state of activity. *ART* performs less checks regarding to class initialization and exception verification in relation to *Java, and* these differences are rendered in Figure 4.2.3-i, indicating that the runtime environment can achieve remarkable performance gains opposed to *Dalvik* [25].

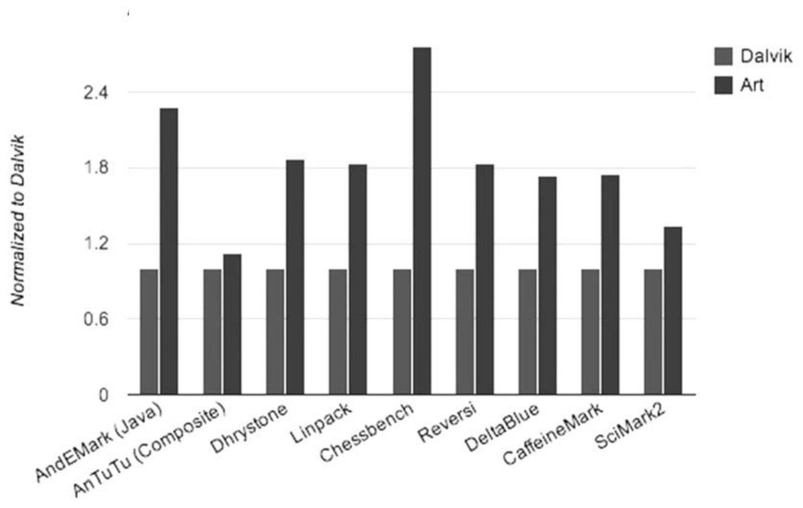


Figure 4.2.3-i Performance distinction. ART against Dalvik [25]

4.2.4 Memory Management

Each process retrieves its own heap space and garbage collector that initially holds capacity of a few MBytes. Heap size is device specific, but modern smartphones can hold up to 96-192 MB per process.

Android utilizes a trace based, mark-and-sweep garbage collector that, from recent revisions has, to some extend, been able to perform its operations concurrently while the application is running. Tracing implies that the garbage collector must traverse the entire collection of objects that are accessible by the process. An object that is object is accessible through some other direct, or indirect object is said to be a*live*, while non referencable objects are denoted as dead and should be reclaimed. Objects are not reclaimed immediately, and unreferenced objects are allowed to accumulate until available memory has been exhausted [26].

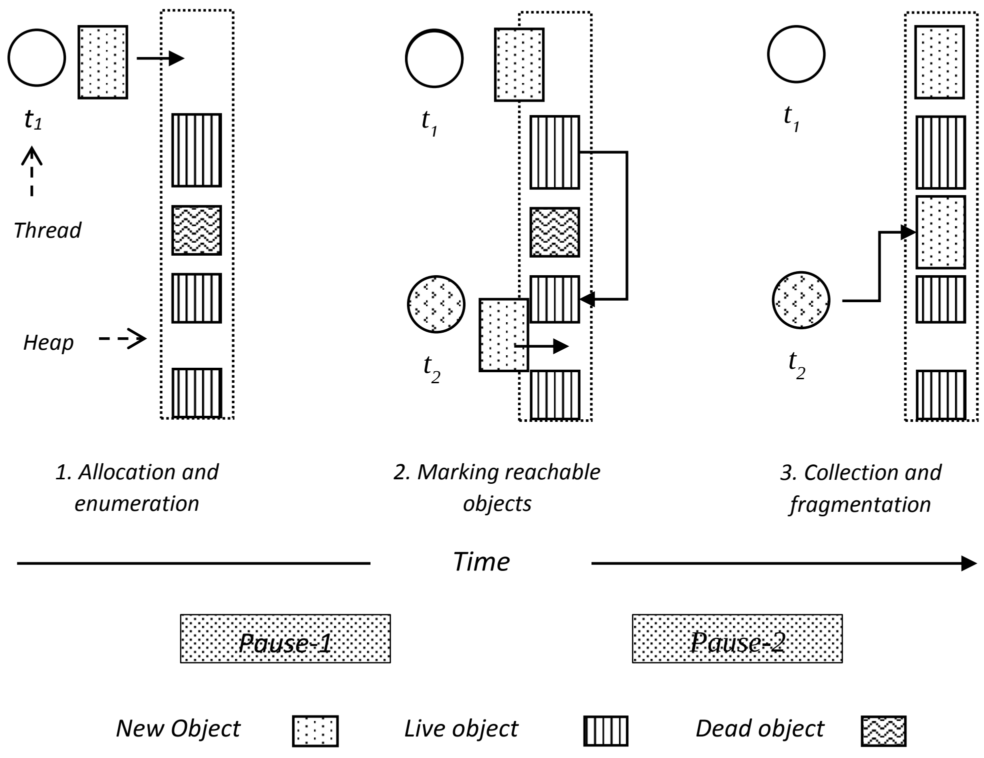

t_1
Thread
Heap
t_1
t_2
t_1
t_2
1. Allocation and enumeration
2. Marking reachable objects
3. Collection and fragmentation
Time
Pause-1
Pause-2
New Object
Live object
Dead object

Figure 4.2.4-i Garbage collection in Android [25]

Figure 4.2.4-i illustrates how garbage collection is performed in *Dalvik*. Assuming that thread t_1 wants to allocate a new object, then, an enumeration operation must be performed in order to locate all live objects. *Dalvik* will suspend all applications threads while this procedure is performed and this is denoted by *Pause-1*. Marking objects that are alive is then performed concurrently, but *Dalvik* imposes another pause in program execution because it needs to verify that no alteration to the state of *dead* and *live* objects occurred between step *1* and *3*. If correctness proves valid, then all *dead* objects can be de-allocated [25], [26].

Both pauses in Figure 4.2.4-i can accumulate to > 5-10 milliseconds of suspended execution and depend on the current heap size and number of objects in the system. Such an operation is referred to as *GC_CONCURRENT*.

This operation cannot go as planned for t_2 because there is no sizable slot available for its allocation. It implies that application execution is completely paused until de-allocation is finished and a sizable slot can be located. If no slot can be allocated, then the heap must be resized. Both operations are referred to as *GC_FOR_ALLOC* – and impose considerable suspensions that last > 50 milliseconds.

It is possible to request a manual garbage collection procedure by a system call, but there is no guarantee that it will be performed, and if multiple threads perform the request within a short time span, then the request will be blocked and run as a single, garbage collection operation, sometime in the near future.

ART introduces a set of new algorithms and strategies in regard to garbage collection. Each application thread is explicitly asked to traverse and enumerate its own heap stack, effectivity removing the need for *Pause-1* in Figure 4.2.4-i*,* and allowing concurrent application execution. Fragmentation is reduced because large arrays of primitive objects are placed in a separate, managed heap and thereby effectively excluding occurrences of *GC_FOR_ALLOC*.

```
GC_CONCURRENT freed 5414K, 17% free 29894K/35608K,
paused 12ms, total 36ms
```

Figure 4.2.4-ii Garbage collection message

Information about collection and allocation procedures are expressed by the runtime environment as console log messages. Figure 4.2.4-ii represents a single message derived from Dalvik. “paused” indicates the amount of milliseconds spent in the state of *Pause-1* and *Pause-2*. “total” indicates the total amount of time it took for the thread to perform the collection procedure.

4.3 Application Framework

A general application or service is constructed with a set of different techniques than what is usual for desktop environments. All applications consist of loosely tied components that have an ability to invoke each other on specific events. Such an event is called an *Intent* and represents an asynchronous message that can be broadcasted throughout the system. Each application is presented as a unique user to operating system - and each process is assigned its own runtime environment that operate its instructions in isolation - within a secured, sandboxed environment. Communication is therefore conducted through lower level *Inter-Process Communication (IPC)* messages [21], [27].

Any interaction with the operating system happens, as presented in Figure 2.4.2-i, through public interfaces that define a restricted and standardized form of development.

4.3.1 Application Components

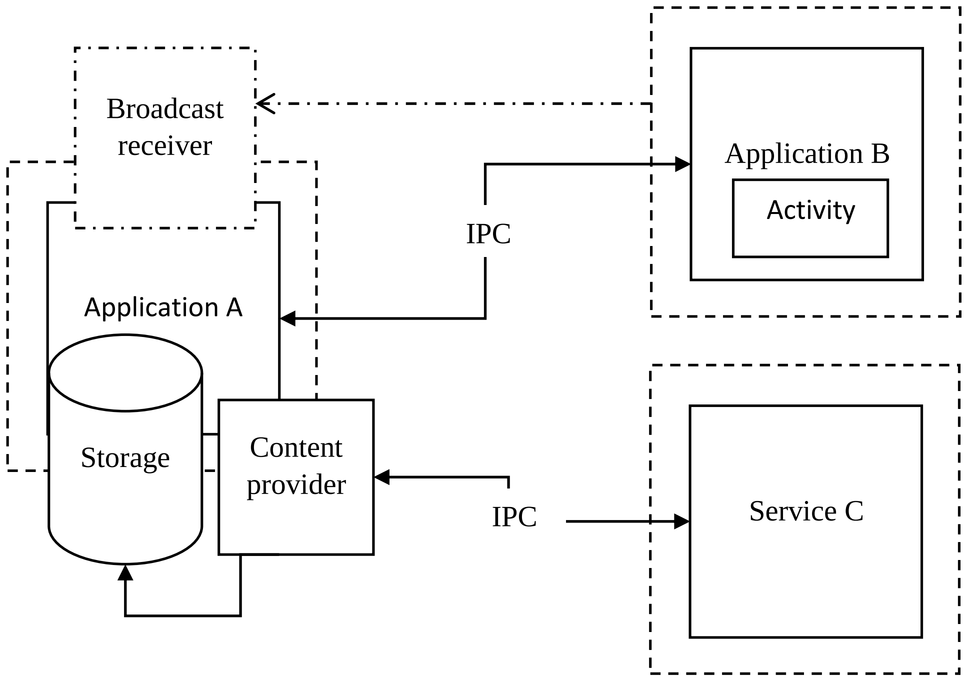


Figure 4.3.1-i Android building blocks

The four main component types that represent capabilities in an *Android* based operating environment, are derived from [27], and listed in Table 4.3-i below.

Component	**Description**
Activity	Presents a graphical user interface that can be interacted with. An activity is run in a separate thread-loop that should not be extensively blocked with long running operations.
Service	Represents an interface-less process that can be implemented as part of an application – or as a standalone background process. Interaction from the outside happens through IPC messages and public interfaces that are described with the *Android Interface Definition Language (AIDL)*. If the service is implemented within an application – then the application is allowed to directly invoke service methods and any message overhead is subtracted.
Broadcast receiver	Are akin to interrupt handlers. The component listens to key events and initiates actions on behalf of the application. Such events can both be system specific (power, screen or camera) and application specific, where another application can broadcast the availability of specific data.

Content provider	Enables external processes to interact with application specific storage repositories. It implies that a process can query and alternate an internal database through publicly defined interfaces.
Storage	Relational database integration is in *Android*, performed by utilizing *SQLite,* a self-contained, server-less, *SQL*-based engine. Each application or process can embed one or more database instances that are private to their host. A single database instance is presented as a file in the system, and read access is restricted per process. By default, any interaction with the database, is performed by higher level *Java*, based libraries. A default *JDBC*[7] compliant driver is not present as library access is, form our observations - performed by lower level, *JNI* calls that operate directly on the *SQLite* instance.

Table 4.3-i Overview of component building blocks in Android

We imply that the combination of sandboxed environments and loosely typed components proves as a scalable and reasonable architecture for the domain of medical monitoring. Security and privacy measurements are controlled and intact. *Services* are first class citizens that provide processing capabilities for a numerous set of tasks – and an application can broadcast a request for assistance throughout the eco-system of the operating environment. A *Service* can retrieve its own memory address space, garbage collector and runtime environment, effectively operating as an individual member of the system that automatically start when the device is turned on.

It proves, as we mentioned in the introduction of this chapter, that *Android* could fit within a heterogeneous environment that resembles our view of the sensor assisted, monitoring domain. Complex systems can be formed from small, autonomous components that are individually maintained and interacted with. An event processing engine could service one or more components and thereby act as a central aggregator within the operating environment.

4.3.2 Lifecycle

We mention in *chapter 4.2.4,* that each process will retrieve a very finite amount of memory and resource reclaiment is performed by the principle of priority and "least-used". It implies that a background process can be killed if the operating system requires more resources for a currently active foreground process – as it is denoted as the currently most important component for the user.

A differentiation between *Dalvik* and *ART* in this domain, is that *ART* enables code to be paged on disk and swapped by the kernel on demand. How this affects termination and process lifecycle, is from our perspective, currently not documented.

Each *Activity* or *Service* - is presented with callback functions that handle how teardown and re-construction operations should be performed. One therefore has, to some extend control over fatal encounters. It is however important to understand that tearing down processes who continuously retrieve and process streams of data from one or more sensors – is not really an option.

Feasible memory footprints and priority settings, can to some extend manipulate the probability of being elected for execution [21], but it is important to understand that we cannot assure that the process will never be killed. Work presented in this thesis could be integrated as a service that reside on lower levels of the environment in Figure 2.4.2-i. This however classifies as future work, as our intention is to observe how *Android*, by default, behaves in our domain.

II. Design and Implementation

5 Requirements Analysis

This chapter describes a preliminary requirements analysis and divides into two sections. The first section is related to the existing enablement of Esper on Android and provides an insight to limitations of the port and a short discussion of why a new enablement is beneficial. The second section describes existing measurements and discusses limitations that concern smartphone devices in context of benchmarks. This discussion is followed by a proposition for how the performance of Esper can be measured and the factors that should be taken into consideration. It proposes that a custom benchmark should be utilized in our context, and includes a notion on how this benchmark should be implemented.

5.1 Correlation between Esper and Android

The only know enablement of *Esper* on a smartphone device, was performed by Bade [10] in 2010. Our correspondence with Bade reveals that the project is discontinued, and it is not stated how much of this work deviates from the current release of Esper. We know that the source code was alternated and that features were removed because of stability and conformity issues with Android [28]. No notion of verification and testing is found. Support for relational database access is not present and disables us from standardized and structured ways of accessing historic information. Event representation can only be performed by key-value maps, and proves that it could be difficult to move any existing applications. Esper has also undergone changes and certain releases are classified as major[8]. We imply that a new enablement of Esper could prove beneficial if we manage to circumvent some of the issues that Bade experienced, and present an updated, fully featured revision of Esper for Android.

We state that an enablement of Esper on an Android based smartphone is true when a newer, stable revision of the engine is embeddable and referable within an application or service on the device. The engine should retain as much functionality as possible, as any subtraction would imply deviation from the original project and cause fragmentation. The way Esper is utilized within a project should preferably not change. It implies that our distribution must be presented in such a way that existing Esper based systems need not be altered.

Esper should be able to query Androids default, relational database implementation as this, in our opinion, is a requirement for several contextual monitoring use cases that depend on historical data.

5.2 Performance Evaluation

The only known measurements that regard *Esper* on a smartphone device, are conducted by *Jaein et.al* [9]. We state that these measurements are of limited use

[8] http://esper.codehaus.org/esper/history/history.html

for us as we miss results that regard several principal components presented in *chapter 2*. Sparse information on how the measurements were conducted also rises questions towards online and offline data-generation, communication and runtime settings.

Mobile devices suffer of certain limitations that disable them from being well suited as clients for existing benchmarks that are aimed at domain specific problems. We state that a modern smartphone device with a 802.11_x wireless transceiver is not able to transfer enough data items per second [29] opposed to what *Esper* should be able to process in context of certain tasks [30]. The use of an *USB* enabled *TCP/IP*-bridge could provide enough throughput, but would provide the device with power and activate the method-based *JIT-compilation* procedure for *Dalvik* as described in *chapter 4.2.3*. Any power preserving algorithms that could affect the overall performance, would neither be activated in such a context.

Modern smartphone devices have a set of thermal constraints that deviate from desktop computers. These devices are mere millimeters thin and expected to be withheld in an enclosed pocket, hand, or against a cheek. Air is not actively pushed into these devices, and a *System on Chip (SoC)* will throttle its operating frequencies and lower the device performance in an effort to reduce the heat that is generated if certain thermal thresholds are met. This can have substantially negative impact on Esper, and it is therefore of interest to measure when frequency throttling begins, and how it affects the throughput of the benchmarks.

Marques et.al [31] argue that the domain of benchmarking DSM and CEP engines is faced with challenges as there is no standardization between query semantics, operators and data formats between existing engines [11]. We are at this point not successful in locating a standardized benchmark that can accommodate to the domain of our context. We state however that work conducted by *Mendes et.al* [30], [31] proves of value for this thesis as it selectively focus on the fundamental principles of event processing by performing a set of micro-benchmarks that address common data stream management and event processing features. Such benchmarks are better suited for our need as they can highlight possible limitations in isolation.

Table 5.2-i presents and overview of tasks and factors that the micro-benchmarks in [30] take into consideration.

Task	**Factor**
Selection	Selectivity.
Projection	Attributes.
Aggregation	Window type, window size and aggregation type
Join	In memory-table. Window-to-window Window size and join selectivity.
Database integration	Access and cache incorporation.
Patterns	Expiration, cardinality and selectivity.

Multiple queries	Parallel, similar queries. Parallel, distinct queries. Parallel join of multiple events.

Table 5.2-i Tasks and factors regarding performance evaluation of Esper

Parallel processing of events is to our knowledge not investigated in any recent studies of *Esper*. Work conducted in [30], utilized only a single thread of execution and we therefore pose questions towards the possible gain of performance caused by utilization of multiple threads and processor cores. Parallel access to similar queries and parallel access to distinct queries would render how locking affects the performance when multiple event types are present.

It is clearly indicated in [30] that database communication is identified as a major bottleneck in queries that include SQL statements that join information from a database table. LRU-cache (see chapter 3.6) incorporation has previously not been evaluated. It therefore implies that it would be useful to see how this feature affects the overall performance of information retrieval.

Chapter 4.2.3 identifies that the introduction of *AOT-compilation* through the use of *ART* as the runtime environment. Figure 4.2.3-i indicates that processor intensive tasks are performed with a substantial ratio of improvement opposed to *Dalvik*. We state however that both runtime environments should be evaluated as *Dalvik* is present on the majority of *Android* enabled devices, and ART is at this stage, not the default runtime environment for any device.

Observing garbage collection and allocation procedures are also of interest. Accumulation of objects in large windows could cause execution pauses that range in hundreds of milliseconds. How real time analysis of incoming data is affected, is therefore of interest to investigate.

The work in [30] utilizes the FiNCOS framework [32] as a benchmarking tool when performing their measurements. We state however that we are not capable of utilizing the same instrument because it requires *Remote Method Invocation (RMI)*[9] procedures. Custom adapters for engine configuration, feeding and result analysis relies on network communication as FiNCOS will not compile and run on an Android based device.

We state that a custom Android based application must be constructed in order to perform the benchmarks, and requirement decomposition for the application tasked with evaluating our enablement of Esper, is stated in the following note.

- The diversity of tasks that relate to the benchmark, must be accommodated by query installation and event schema definition procedures. A task should preferably be separated from the system that conducts the benchmark to ensure easier maintainability and impose that the benchmark could be used for other tasks that are not evaluated in this thesis.
- Parallel, multi-threaded processing implies that each thread should work in sufficient isolation. Shielded from any additional, concurrent access to

[9] A *Java* component for object-oriented Remote Procedure Calls that is not supported by *Android.*

shared resources that could impose locking and queue mechanisms that impose potential bottlenecks in our measurements.

- Generation of data items for the measurements must conform to a strategy that causes the least of interference for the experiments. We stated in *chapter 4.2.4.* that Android's garbage collection procedures are affected by cardinality of objects in the heap. It implies that a large amount of pre-allocated data items could affect the measurements because verification pauses could be of considerable length.
- Monitors must be of such nature that they cause the least disturbance, and at the same time, ensure continuous observations of all threads that conduct work without the need of synchronization and concurrent access procedures.
- Relational database integration must be supported in order to perform measurements related to relational database joins. A *SQLite* database must be constructed and prepopulated between measurements and result-cache incorporation must be adaptable.

6 Asper. Enabling Esper on Android

This chapter describes how Esper can be embedded in Android. It highlights challenges that Android imposes and describes the changes that must be conducted in order to render an enablement true. This is followed by a description on how the relational database in Android can be bound to Esper. Effectively enabling it to join historical information and enriching streams of information. This chapter is concluded with a notion on how this embedment is verified and publicly distributed.

6.1 Limitations and Issues

We state, based on our own observations, that *Esper* will, by default not run on an *Android* based device. Applications will simply not compile because *Android*, as a framework, misses several, critical libraries that *Esper* depend upon.

We indicate in *chapter 4.2.2* that *Android* utilizes an implementation of *Java* that differs from the mainstream distribution. *Esper* and its accompanying third-party libraries are highly dependent on the mainstream *Java* implementation, and state it as a requirement that must be fulfilled in order to prove *Esper* operational.

Libraries and applications that are developed using the official *JDK*, will not compile or execute if they reference any of the excluded or alternated libraries. Introducing missing libraries is prevented upon compilation if the package namespace is equal to signatures found in the *JDK* packages [33]. All missing library components must be provided at compilation time - and require that we re-package each dependency into its own namespace and reference it correctly from the source code. Re-packaging is required to prevent namespace collisions.

We identify, by tracing method calls, that *Esper* utilizes dynamic class generation to prevent itself from depending on reflection when accessing attributes or functions.

This operation is performed by utilizing *CGlib,* as we describe in chapter 3.1, and occurs if we either:

I. Register an event type by providing a plain *Java* class.
II. Utilize subscribers instead of listeners[10].
III. Reference an external function in an installed query.

An application that performs one or more of the operations presented above - will simply terminate its execution and throw a runtime exception, stating that the runtime environment is not capable of accepting and interpreting a *Class instance*. We state in chapter 4.2.3 that the runtime environment in Android is only capable of interpreting *DEX* instances. It therefore indicates that the exception and application termination is caused *Android's* inability to convert a *Class* instance to a *DEX* instance at runtime.

These issues require resolvement in order to render an enablement of Esper on Android true. Our resolution to these problems is presented in the following chapters, and will hopefully act as a guideline for enablement of newer versions of engine in the future.

6.2 Required Changes

Missing dependencies are resolved by locating and extracting components[11] from the OpenJDK[12][34] initiative. Any component that has unresolvable dependencies of its own, must also be accommodated in the same manner.

Namespace and signature collisions are resolved by appending the prefix "com.asper". Each extracted OpenJDK component is therefore refactored to match this namespace.

This operation results in extraction and refactoring of 187 components across 32 packages. Every reference from within the source code of Esper, is changed to match the new namespace. This implies that only import statements are changed and that the risk of introducing subtle errors in the system are kept at minimum as no functional code is altered.

Third-party libraries, described in chapter 3.1 needs similar resolution. ANTLR presents, from our perspective, no dysfunction with *Android*. However, libraries regarding logging and XML parsing presents exceptions that prevent proper application execution. Enablement of these libraries is present through other open sourced projects, and utilization of them is from our opinion, a better practice as they stand some external testing and relive our time consumption. references these packages and state their origin.

Library	**Project**

[10] See [19], section 14.3.2
[11] A component in this context, is any *Java* based class that follows with the OpenJDK distribution and provides some utility in regard to networking, graphics, I/O.
[12] Open source implementation of the Java development and runtime environment. Any reference to OpenJDK throughout this thesis, regard to OpenJDK version 6.

Commons Logging	SLF4J-Android. v1.5.8 [35]
Xerces	Xerces for Android. v2.11.0 [36]

Figure 4.3.2-i Third party library replacements for Asper

This refactoring procedure is however, too extensive for a particular, internal utility in Esper. The *MetricUtil* class, residing in the “espertech.util” namespace, is responsible for engine and statement metrics that represent CPU utilization and processing capabilities[13]. *MetricUtil* relies on the *ManagementFactory* and *ThreadMXBean Class* instances, both unsupported by *Android.* These instances contain an extensive amount of dependencies, and we state that many of these dependencies root to *Swing* based *GUI* components that represent desktop based graphics (windows, buttons) and have no value for *Android*. Utilization of *MetricUtil* is therefore disabled by retraining method signatures and replacing their content with a log message warning that activates upon invocation.

We state, in context of dynamic class generation, that CGlib cannot be excluded as the library is too extensively used throughout the source of Esper. Tracing proves that only a sub-set of CGlib is utilized by Esper, and that this sub-set is primarily concerned about effective method and attribute access through indexation, rather than recursive, string based, name comparison. We therefore feel confident in selecting a resolution strategy where we re-write CGlib to only include the method signatures that Esper depend upon, and thereafter change these methods to perform reflection rather than indexation and class generation. The classes that concern these changes are identified as *FastClass* and *FastMethod* and reside under the “net.sf.cglib” package namespace. **Error! Reference source not found.** presents he methods that are altered.

[13] See [19], section 14.14

```
package com.asper.sources.net.sf.cglib.reflect;
public class FastClass {
   protected FastClass(Class type) { }
   public static FastClass create(Class type) { }
   public static FastClass create(ClassLoader loader, Class type) { }
   public Object invoke(String name, Class[] parameterTypes, Object obj, Object[] args)  { }
   public Object newInstance()  { }
   public Object newInstance(Class[] parameterTypes, Object[] args)  { }
   public int getIndex(String name, Class[] parameterTypes) { }
   public int getIndex(Class[] parameterTypes) { }
   public int getIndex(Signature signatureA) { }
   public Object invoke(int index, Object obj, Object[] args)  { }
   public Object newInstance(int index, Object[] args)  { }
   public int getMaxIndex() { }
   protected static String getSignatureWithoutReturnType(String name, Class[] parameterTypes) { }
}
```

Code sample 6.2-i CGLib; changed method implementations to support reflection

It could be stated counterintuitive to introduce reflection when the purpose of CGlib was to remove it. However, we state in chapter 5.1 that stability and compatibility is a primary goal, and that execution exceptions are of no tolerance. It implies that the performance increase achieved by indexing, is at this point of insufficient relevance. An optimal resolution this problem will require an in depth study of how the runtime environment could be invoked to perform Class to *DEX* compilation at runtime, from within an application.

6.3 Relational Database Integration

We state in chapter 3.6, that Esper requires a JDBC compliant driver in order to communicate with a relational database. Androids integration relies on a set of higher level API functions. Tracing these functions render that database interaction is performed by a set of lower-level, pre-compiled, JNI calls, and Esper is not capable of utilizing these functions as it is instructed to invoke standardized JDBC method interfaces.

The solution to this problem, is from our view, to compile and present a JDBC compliant driver to the application as a library, as this automatically distributes the driver with the application, and prevents specific configuration of each individual device. A fully featured implementation of SQLite is redundant, as the database is already embedded in Android, and could cause implications that relate to rights management (see chapter 4.2.1).

A project named SQLDroid[37] intend to present a JDBC compliant SQLite driver to the Android platform, but fails to implement interfaces that support prepared statement execution. Utilizing this driver will cause runtime termination upon query registration as Esper depends on prepared statements when it embeds projected attributes from an event, as variables in a SQL statement. Joining information between a stream and persisted data, would therefore not be possible.

We propose that it should be sufficient to only provide an application with the appropriate API, and a reference to where a SQLite instance is located. The Berkeley Database project [38] leverages a compliant JDBC driver as it incorporates support for SQLite as a drop-in. It embeds the necessary API, but leaves inclusion of SQLite optional.

Compilation of this driver requires utilization of the Android NDK as it imposes lower level JNI calls. It implies that our implementation is platform specific, supporting only smartphone devices that embed an ARM based processor[14]. Procedures for compilation are available at [39]. The driver is identified as "sqlite.jar", located under the "libs" directory. Any utilization requires that the driver requires that the library is included in the build path of the application.

This driver will not return metadata for precompiled, prepared SQL statements. It proves as a problem for Esper because the engine relies on this information for query validation and output projection[15]. An engine instance must therefore be configured to perform lexical analysis of the SQL statement, thus enforcing a custom engine configuration.

A complete configuration sample is presented in Code sample 6.3-i.
<Path to database> denotes the absolute file path to a SQLite database instance.
<Database name> identifies this database in an installed query.

[14] Related to the hardware presented in chapter 8.1.1
[15] See [19], section 5.13.7

```
// Obtain a configuration reference
ConfigurationDBRef reference = new ConfigurationDBRef();

// Provide a notion of the driver to use – and where an instance of our
// database can be located in the file-system.
reference.setDriverManagerConnection (
"SQLite.JDBCDriver", "jdbc:sqlite: <Path to database>",
new Properties()
);

// State that meta-data retrieval should happen through sampling
reference.setMetadataOrigin(ConfigurationDBRef.MetadataOriginEnum.SAMPLE);

// Relate the configuration reference to a named database that,
// on a later point can be referenced from a query.
configuration.addDatabaseReference("<Database name>", reference);
```

Code sample 6.3-i Database configuration sample

6.4 Verification and Distribution

The complete port of Esper, including all third-party libraries and our SQLite driver is presented as public, open source project identified as *Asper.* Access is present for anyone and located in *Repository 1* below.

https://github.com/mobile-event-processing/Asper

Repository 1 Asper source repository

Asper represent version 4.8.0 of Esper, as this is the most current release of the engine at the point of this thesis. Any reference to Asper in this thesis, represents our alternated version of Esper.

Verification is conducted by running 766 distinct unit-tests that each isolate and verify small pieces of testable code. These test are distributed with the source code of Esper and leverage JUnit[16] for execution. The complete test suite is larger and relies on over 85 tests that concern database communication. These tests will fail because they rely on MySQL, a relational database that is not supported by Android. Tests that rely on the MetricUtil component will also fail because of our exclusion. (see chapter 6.2).

The core of Esper is packaged as a JAR (Java Archive) and retains the same procedures for utilization as its original counterpart. Any documentation that regards Esper, should therefore also regard Asper in the same way. All third-party libraries accept for CGlib are separated, and must be included in the build path upon

[16] A unit testing framework for Java.

compilation. Separation is conducted to ensure easier maintainability, and to retain the same project structure that Esper imposes.

Embedding future releases of Esper should be performed with ease as it should be sufficient to only copy components from the "com.asper" namespace, and alternate all import statements. *Integrated Development Environments* could automate this task by performing regressive text replacement operations, and we state that an automated script is not needed.

EsperTech, which retains the rights to Esper, imposes that Asper must retain the original, GPL-2.0 license for distribution.

7 Benchmark Implementation

This chapter is divided into three sections and describes how the performance benchmark from chapter 5.2 is designed and implemented. The first section describes the design and implementation of a set of tasks that each exercises distinct properties of DSM and CEP systems, and are bound to a set of experiments in part III. The second section presents an architectural overview of the benchmark and highlights how different components coordinate their actions in order to render a measurement true. The third section describes the benchmark implementation and highlights details about specific components from section two. This chapter is finally concluded with a notion about the availability and distribution of this benchmark.

7.1 Tasks

The intention of our benchmark is, as noted in chapter 5.2, to evaluate if it proves feasible to utilize our port of Esper on a smartphone device. Asper must therefore be presented to a set of disparate tasks, that each places pressure on distinct principles that regard both stream management, and event processing.

Chapter 5.2 uncovers the set of tasks and factors that we consider important in order to evaluate our implementation of Esper. We proceed this chapter by presenting the design and implementation of each task, and render their relation to principles that concern DSM and CEP in chapter 2.

7.1.1 Task 1: Selectivity

The intention of this task is to measure how filtration effects the throughput of the engine by altering the predicate selectivity. A higher selectivity rate will force the engine to output results more often and render how the output mechanism could affect the overall throughput of the engine. A lower selectivity rate will render how many events the engine is able to consume and analyze in scenarios where the chance of finding the event we are looking for is low.

Schema, *$Event_a$:*

Attribute	**Type**	**Value**
Id	Integer	Random: [1, I_n]
a_1	String	Fixed

```
SELECT id, a₁
FROM Eventₐ (id <= K)
```

Query 7.1.1-i

```
SELECT id, a1
FROM Eventa
```

Query 7.1.1-ii

In Query 7.1.1-i,, *K* is used to project the desired predicate selectivity by specifying that only event instances with identifiers with lower or equal I_n cardinality should be selected for output.

It should be noted that Query 7.1.1-i is implemented with a filter on the actual event stream, and filtration will be performed before the event enters a window. Projection is limited to two distinct attributes to exercise selectivity in isolation.

Query 7.1.1-ii has the intention of selecting and outputting each and every data item that enters the engine. Its purpose is to remove any aspect of filtration from our processing chain, and solely measure the ability to output data to listeners.

7.1.2 Task 2: Projection

Projection should render how attribute selection affects overall performance. It is in our interest to see if the engine is capable of selecting a high amount of samples that could represent a variety of data items. Inspecting multiple attributes simultaneously, could prove beneficial in situations where we are scanning for patterns, because we are not sure where favorable information is present.

Schema*, $Event_a$:*

Attribute	**Type**	**Value**
$a_1 \dots a_{512}$	Double	Fixed Random: [1.0, 1000.0]

```
SELECT a1 … a512
FROM Eventa
```

Query 7.1.2-i

Attribute a_x denotes how many attributes the event schema and projection query should include. Query 7.1.2-i will select all incoming instances of $Event_a$ and project its attributes as output.

7.1.3 Task 3: Aggregation

The intention of this task is to evaluate a set of aggregation functions with jumping and sliding windows of variable length. We note in chapter 2.2.4, that aggregation functions will have different computational costs based on their intention and affiliation with specific window types. Sliding windows will impose eager re-evaluation as new data items arrive, while a jumping window performs the re-evaluation in a batch process when window boundaries are met.

Schema*, $Event_a$:*

Attribute	**Type**	**Value**

a_1	Double	Fixed Random: [1, 1000]

```
SELECT f( a₁ )
FROM Eventₐ.win:length [_batch] ( Wₙ )
```

Query 7.1.3-i

Query 7.1.3-i represents both a sliding, and a jumping window, differentiated by the inclusion of "_batch". W_n denotes the window size. *f(a_x)* denotes the aggregation function and is replaced with *SUM()* and *MEDIAN().* a_x denotes the data item attribute that should be computed.

The choice of aggregation functions is based on [16], and can be summarized by the following note :

- *SUM(a_x)* returns the sum value of *n* data items and can be computed at fixed cost independent of window length and policy.
- *MEDIAN (a_x)* computes the median of a_x, and its computational cost will depend on window size.

7.1.4 Task 4: Join – Table

The intention of this task is to perform a join between two distinct event types $Event_a$ and $Event_b$, where $Event_a$ is preloaded in a window that acts as a static lookup-table and every incoming instance of $Event_b$ is matched against one and only one instance of $Event_a$. The purpose is to measure how joins between a static window that contains pieces of historic information, and a dynamic window that continuously moves affects the overall performance of the engine.

Schema, *$Event_a$, $Event_b$:*

Attribute	**Type**	**Value**
Id	Integer	Random: [*1, W_n*]
a_1	Integer	Fixed
sequence	Integer	Sequential: [*1, W_n*]

```
SELECT Eventₐ.a₁ , Event_b.a₁
FROM Eventₐ.win:length( Wₙ ) , Event_b UNIDIRECTIONAL
WHERE Event_b.id =  Eventₐ.sequence
```

Query 7.1.4-i

W_n ensures that the lookup table retains a fixed window size. W_n instances of $Event_a$ must be preloaded into this window before the measurements starts. '*UNIDIRECTIONAL'* ensures that our query is only evaluated on the arrival of $Event_b$. The WHERE-clause performs the actual join by specifying that the *id* attribute of $Event_b$, which takes a random value between 1 and W_n , should correspond to the generated sequence number of some $Event_a$, and thereby ensure that a match will be located for any incoming instance of *Event*$_b$. Projection enforces extraction of joined information.

7.1.5 Task 5: Join – Window to Window

This task exercises continuous joins between two sliding windows of variable size. Window size and predicate selectivity are factors under analysis, as a considerably sized window could imply a performance degradation caused by garbage collection and allocation procedures (see chapter 4.2.4). Exercising window size and predicate selectivity in isolation, ensures that we inspect allocations and collections distinctively.

Schema*, Event*$_a$ *, Event*$_b$ *:*

Attribute	**Type**	**Value**
id	Integer	Random: [*1,* I$_n$]
a_1	Integer	Fixed

```
SELECT Event_a.a_1 , Event_b.a_1
FROM Event_a.win:length( W_n ) , Event_b.win:length( W_n )
WHERE Event_a.id =  Event_b.id
```

Query 7.1.5-i

This task is divided into two distinct measurements that each individually place above factors under analysis:

a. *Varies window size and keeps selectivity fixed at 100%*.
W_n, which denotes the window size, and I$_n$, which denotes the event identifier range, takes the same value to ensure that each *Event*$_a$ will find at least one *Event*$_b$ in the other window.

b. *Retains window size fixed and vary join selectivity*.
W_n is set to *n* and I$_n$ is set to W_n * 10, 1 and 0.1 to ensure that each instance of Event$_a$, will on average find 10, 1 and 0.1 instances of Event$_b$ in the other window.

7.1.6 Task 6: Pattern Detection

The pattern matching task will search for the occurrence of an event that has the same identifier within a temporal context. Predicate selectivity is imposed by stating that a secondary attribute must be above some constant value. The intention is to place pressure on cardinality and evaluation policies by altering how many, and how frequent event sets the pattern sees in a context. High range of identifier variability will force the engine to keep notion of many distinct events, while a large temporal context will imply that many events can accumulate and pose higher requirements for output generation.

Schema, *Event*$_a$ *:*

Attribute	**Type**	**Value**
Id	Integer	Random: [*1,* I$_n$]

a_1	Double	Random: [*1.0, 100.0*]

```
SELECT *
FROM PATTERN
[

EVERY x = Event_a ->
[EVERY] y = Event_a (id = x.id AND a1 > K) WHERE timer:within(W_n)

]
```

Query 7.1.6-i

The duration of a temporal context is defined by W_n and denotes retainment of $Event_a$ instances. Selectivity is imposed by a correlation of *K* and a_1, and ranges from 1.0 to 100.0 to ensure a desired percentage. I_n denotes the amount of distinct identifiers. Evaluating different pattern policies can be exercised in isolation by varying a single predicate W_n, *K or* I_n, while retaining the other predicates static. The additional [Every] operator should denote the cost of attempting to match every previously seen instance of x and y, effectively retaining a history of seen events and placing pressure on output generation.

7.1.7 Task 7: Database Integration

This task concerns itself with joins between a prepopulated relational database and Asper. The intention is to measure how communication overhead could affect the engine performance, and to identify the maximum achievable amount of data items that can be joined from within a query. We mention in chapter 3.6 that Esper embeds a result cache that stores a set of recently retrieved database results in order to prevent the need for reoccurring database entry. How this cache improves performance is of interest to exercise in order to reveal the potential gains that lies within.

Schema, $Event_a$:

Attribute	**Type**	**Value**
Id	Integer	Random: [1, I_n]
a_1	Double	Fixed

Schema, database:

Attribute	**Type**	**Value**
Id	Integer	Sequential: [1, I_n]
$t_1 \dots t_5$	Double	Fixed

```
SELECT Event_a.a_1, Db.t_1
FROM Event_a, SQL:database
[
        "SELECT t_1 FROM table WHERE id = ${ Event_a.id }"
] AS Db
```

Query 7.1.7-i

Every instance of $Event_a$ is joined with a single record in a pre-populated database. The amount of records is denoted by I_n and represents a sequential primary key. The implications of result caching could be measured by balancing cache misses. Thereby forcing the engine to re-query the database and update the cache before an answer is returned.

7.1.8 Task 8: Multiple Events and Threads

The intention of this task is to render if, and how disparate event types and multiple threads affect the performance. We are concerned about scenarios where high-priority events from one or more sensors retrieve their own processing thread and output listener, but must share the same engine instance.

1 - n separate threads work with 1 - n instances of Query 7.1.1-i from *Task 1.* Each instance of Query 7.1.1-i references a separate event type and each thread will only process one type of event to ensure that no reentrant locks could occur between the *n* queries. Selectivity is set to 25% and each instance of Query 7.1.1-i retrieves its own listener.

7.2 Architecture

This section aims to present a higher-level overview of the benchmark that implements and measures each distinct task from chapter 7.1.

7.2.1 Overview and Domain

The collection of tasks presented in chapter 7.1, renders how disparate the tests and event-types can be.Each task has a set of queries that must be installed. One or more unique event schemas that must be registered and one or more attributes that require a name and random value assignment. Certain events must be pre-loaded into the engine in order to perform static joins between windows and must therefore be omitted when the actual test is conducted. All tasks are in this way, uniformly similar – but small distinctions prevents us from constructing a simple, hardwired and generic model that that accommodates to all needs.

We state in chapter 4.2.3 that the default runtime environment in Android retains a JIT-compiler that optimizes segments of code for faster execution. Such an operating environment will also embed a set of buffers that might render a distinctly higher throughput rate if it is allowed to run for a certain amount of time. We also state, in chapter 5, that smartphone devices are not actively cooled. It could imply that processing capabilities differ as time progresses. Retainment of reproducible measurements require that we are able to place the environment in a certain state. A pre-measurement procedure is therefore required in order to warm up the runtime environment, the engine and the device.

Monitoring of the effective throughput can be conducted by measuring how many data items we are able to process per second. However, unpredictability in output generation requires us to place a monitor at the input-end of the engine. We state in chapter 3.1 that Esper retains an architecture where each thread that delivers a data item to the engine – is also responsible for processing it all the way to an awaiting listener. Latency is therefore – from our viewpoint – not of importance to measure explicitly as answers are produced near instantaneously. Detecting when the engine should have generated output, would require us to hard code every test and any generalization would be lost. Effectively rendering a benchmark framework useless for other purposes.

Multiple, parallel-working threads must, as stated earlier, be exposed to the least amount of locking and waiting to prevent any impact on the measurements. We state that a shared data item repository imposes that 2-4 accessing threads will use 15-30% of their time waiting for access. Each thread should therefore retain its own data item repository and local monitor to prevent parallel access. Dealing with equal and disparate event-types in Task 5 and Task 8 – also imposes a notion of correct event distribution among the workers. Each worker must retain some notion of synchrony – and a set of barriers and conditions must be met to ensure that all workers start and pause their measurements and processing on equal terms.

A higher-level overview of the system is presented in Figure 7.2.1-i. Each component in this figure is described in Table 7.2-i

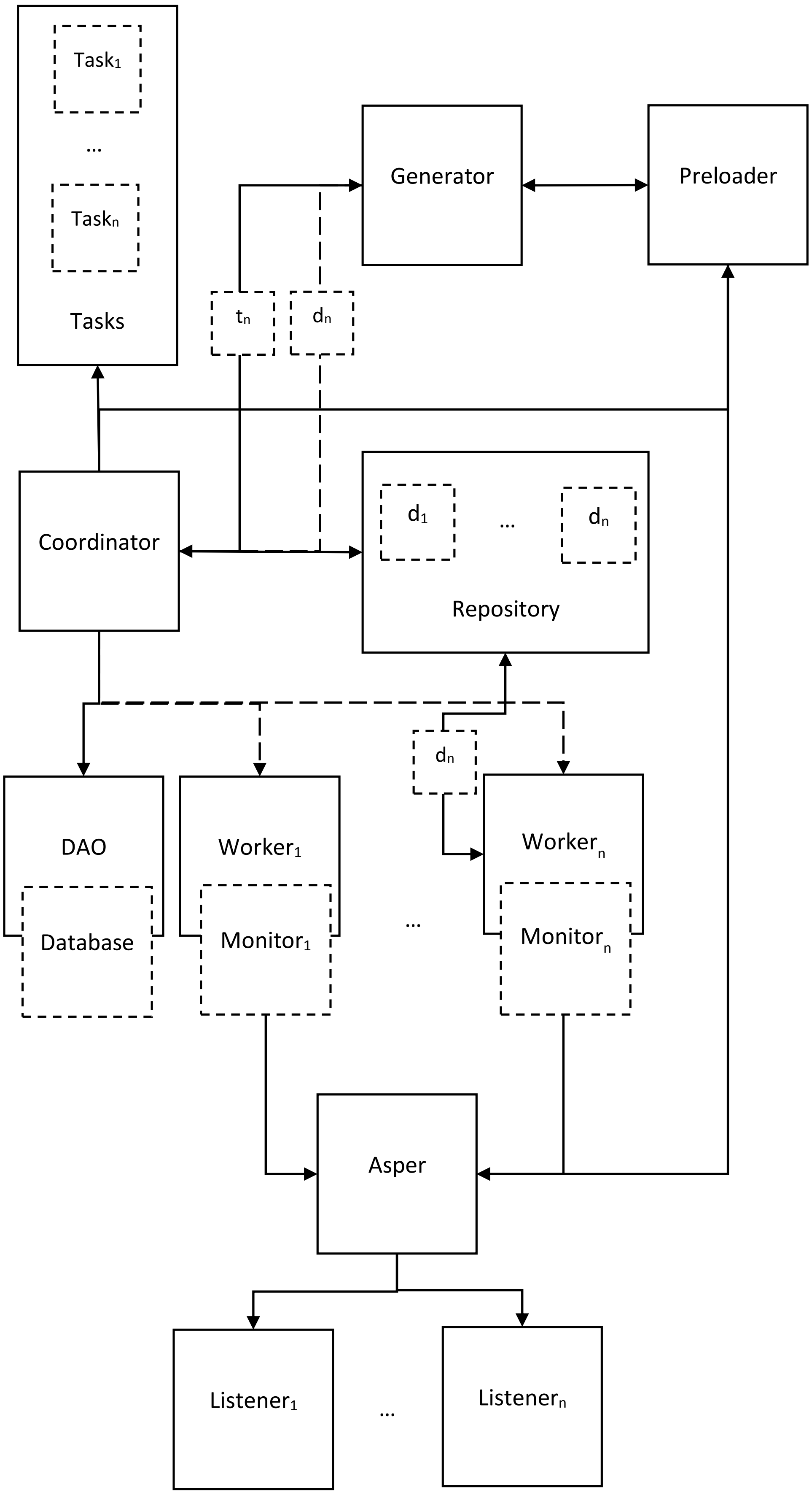


Figure 7.2.1-i Benchmark component architecture

Component	Description
Task	A *Task* is an abstraction that retains information about the current task to perform. Each *Task* withholds the queries to be installed and information about the events that should be utilized for a particular test. A single *Task* is also denoted by t_n.
Tasks	The *Tasks* component acts as a central coordinator for the collection of *Task* instances that reside in the system. It is responsible for load and retainment of the currently active *Task* that the system utilizes in an active test sequence.
Coordinator	The *Coordinator* is responsible for instantiations and reset procedures between each test. A *Coordinator* will ensure administration of *Worker* and *Task* instances, additionally to data generation. The *Coordinator* will not perform any extensive operations – but rather call abstracted methods in an ordered sequence.
Generator	A *Generator* resolves *Task* instances to data items that can be presented to the engine. A single data item is also denoted by d_n
Preloader	A *Preloader* ensures that a predefined amount of data items Is presented to the engine before a test starts.
Data Access Object (DAO)	*DAO* acts as an abstraction to the relational database. Its responsibilities relate to teardown, setup and preloading procedures. *DAO* works with a private instance of a *Database* component that represents a *SQLite* reference and information about the database schema. *Asper* by itself, will not work with this component, as all our queries must be performed within the engine, which retains its own, separate connection pool.
Repository	A *Repository* retains a set of generated data items, placed in a particular order by a *Coordinator*.
Worker	A *Worker* represents a distinct thread that utilizes an instance of *Asper* to feed and process data items. Each *Worker* withholds its own instance of a *Monitor* and a local storage of data items.
Monitor	A *Monitor* withholds a notion of time and throughput of currently processed events.
Asper	*Asper* presents an abstraction of an engine instance. It ensures query installation, event-type registration and output listener initialization.
Listener	A *Listener* retrieves output from installed queries, but performs no further computations and sheds the output immediately.

Table 7.2-i General description of benchmark components

7.2.2 Component Correlation and Application Flow

The flow of operation between components described in chapter 7.2.1 is presented in Figure 7.2.2-i. It renders an image of method invocations for a single test. Effectively showing how a set of the most important components correlate to one another.

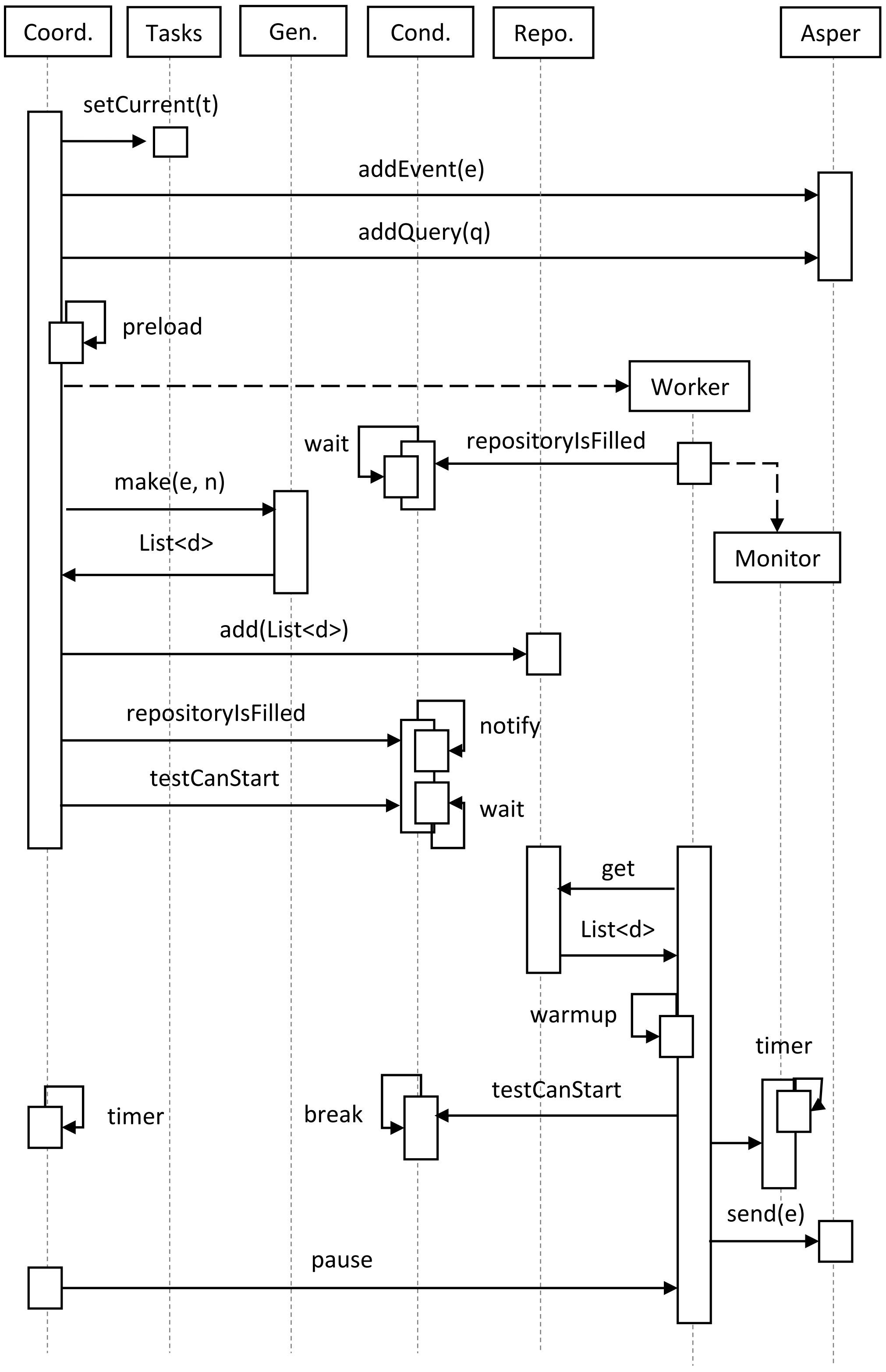


Figure 7.2.2-i Benchmark flow diagram

The flow of operation in Figure 7.2.2-i can be described by the following three segments:

I. The *Coordinator* (denoted by *Coord*.) will set a *Task t* as the currently active task.

   Each event type is then registered. *Asper* is able to interpret an event-type and registers a model of its schema.
   All queries from the current *Task* are subtracted and presented to *Asper* for installation. *Asper* will at this point install each query and initialize a *Listener* for it.

   The *Coordinator* will locate any event-types that require preloading and ensure their presence through the use of a *Preloader* component if necessary.

II. All Worker components are instantiated and activated. Each Worker will synchronize with a *Condition (*denoted *Cond*.) component that withholds a lock labeled *repositoryIsFilled*.

   The *Coordinator* then ensures that each event-type in the system is generated by invoking the *Generator* (denoted by *Gen*.) component to make n_e items of each event-type. Where n_e is decided by the total amount of data items to be generated in the system divided by the amount of event-types in the test. Each generated item is collected locally and the complete set of data items is, if required, randomly shuffled before being sent to the central *Repository.*

   The repositoryIsFilled condition is then notified and all awaiting *Worker* components will retain an equal set of data items. The *Coordinator* will at the same time await at a barrier labeled testCanStart.

   All *Worker* components will start a warmup procedure where they perform work without any monitoring. Each *Worker* will then synchronize and await at the *testCanStart* barrier, effectively breaking it and signaling to the *Coordinator* and any awaiting *Workers* that the test is ready to be initiated.

III. The *Coordinator* winds up an internal timer that fires after a predetermined amount of time and invokes a pause procedure. Each *Worker* component invokes at this time its local *Monitor* component, that itself winds up a reoccurring timer that snapshots the currently processed set of data items.

   Each *Worker* processes a set of data items and invokes an update procedure on the *Monitor* until paused by the *Coordinator*. State is at this point retained for further calculation until a new test starts.

7.3 Implementation

The intention of this chapter is to describe how certain and important components are implemented to form the system presented in Figure 7.2.1-i. Each individual part in this chapter, reside in Repository 2, presented in chapter 7.4. Certain code samples represent only an outtake each implementation.

7.3.1 System

The system that we present in chapter 7.2, can be implemented in several ways. We state in chapter 4.3.1 that Android retains a set of components that acts as sandboxed building blocks for complex applications.

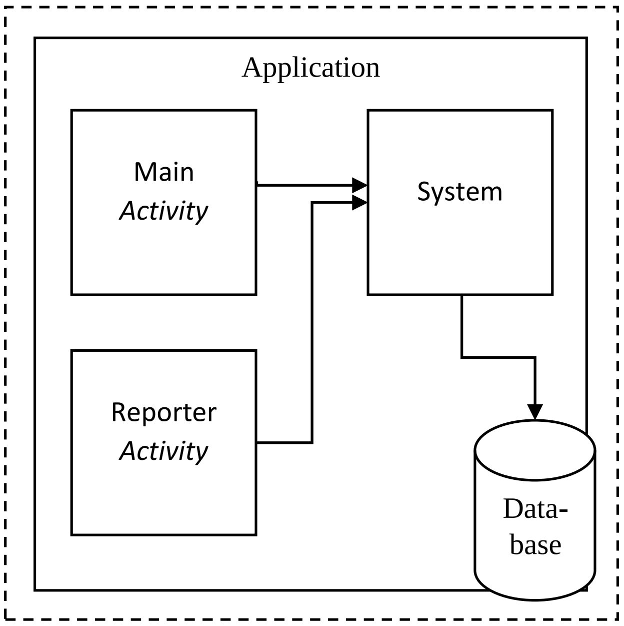


Figure 7.3.1-i General Benchmark implementation

Figure 7.3.1-i presents an implementation proposal. *System* denotes all components from Figure 7.2.1-i. The *Main* and *Reporter Activity* components denote graphical interfaces that retain control over user interaction. Their presence is described in chapter 7.3.7. The *Database* component represents a physical *SQLite* database instance.

It could be stated sensible to place the *Generator* component in a separate *Service* and thereby ensure that it retains its own Heap and runtime environment. Such a setup would be more true to a scenario where *Asper* itself, is implemented as a *Service* that acts as a central aggregator for all events that happen on a device. We state however that the main bottleneck for such a setup is directly related to the amount of *IPC* messages a particular device is able to process. We state that the hardware presented in chapter 8.1.1, is only capable of processing approximately 8k IPC[17] messages per second. Marshaling and queue retainment is the primary factor for this cost of operation and a single test will be severely affected by the overhead. Our primary intention is to

[17] Payload is represented by a plain *Java* object with 3 numeric attributes. *IPC* messages are sent by invoking service interfaces, described directly by *AIDL.* It implies that we do not use the higher-level *Messenger* component in *Android* as it imposes a fixed 1 MB buffer that terminates the whole application upon overflow.

measure how well our port of *Esper* operates on an *Android* based device. We therefore state the proposal in Figure 7.3.1-i is sensible as it allows us to perform measurements in sufficient isolation, effectively rendering the engines true performance.

7.3.2 Task

A single *Task* instance is, in the application, identified as a plain *Java* object. Each instance must include a notion of names, queries, attributes, values and preloading options that aid the system in understanding how the test should be conducted. *Esper* requires that we, for each named event, state each attribute name, position and datatype. We state that the functional aspects of the system and the tasks themselves, should be separated from application as it enables us to create a generic framework that can run a disparate set of tasks without the need of recompilation and profound expertise of the programming environment.

Each *Task* is therefore written in *JavaScript Object Notation* (*JSON*)[18] – and can be described in a human readable way. We state that this enables an external user to write a set of *Tasks* that may relate to domain specific scenarios that we do not cover in this thesis. We also state that if our work is to be extended in the future and include a networked client/server model, then, it should, with small alterations be possible to omit local data item generation – and simply transmit a string of *JSON* to inform the engine about the events that are to arrive.

An example of a *JSON* based *Task* is presented in Code sample 7.3-i. A description of each key is presented in Figure 7.3.2-i below.

Key	**Description**
Label	Represents a human readable task name
Queries	Presents an array of one or more textual queries that are to be installed within an engine.
Events	Presents and array one or more event-types, where each event is represented a separate object of type *Event*
(Event) Name	The event name. An engine will recognize and match a data item to a query by this value
(Event) Preload	An optional attribute that denotes if this event-type should be preloaded. A numeric value denotes how many instances to preload.
(Event) Variables	Presents an array of one or more event variables / attributes that are to be generated and expected by an engine.
(Variable) Name	Presents an attribute/variable name. Utilized in relation to event-schema generation and in any installed queries.
(Variable) Range	Presents that this value should be picked randomly from a span that ranges from the first, to the last number in the array. The numeric value can either be presented by an Integer, or Decimal. Decimal numbers will retain randomness on both sides of the decimal point.
(Variable) Value	Presents a fixed value that can either be a Number, String, Char or Boolean.

Figure 7.3.2-i Description of a JSON based Task

[18] An open and standardized, textual, key-value object description format that can be used as an alternative to XML. See http://json.org for more information.

```
    label: "Task name, type",

   queries:
[
      "Query1", "Queryn"
],

   events:
[
    {
        name: "A",
        preload: 1000 (optional)

        variables:
        [
           {
              name: "ID",
              range: [1, 1000] || [1.0, 1000.0]
           },
           {
              name: "A1",
              value: "text" || true || 1.0
           }
        ]
   }
]
```

Code sample 7.3-i Example of a JSON based Task

Transformation between a *JSON* based *Task* and a *Java* based object is performed by *GSON*[19], a third-party library that parses *JSON* based content and maps it to a predefined *Java Class* that holds an equal structure. All type conversion is performed automatically, but it implies that any limitations must be accommodated by custom conversion adapters for any non-primitive types that one would wish to add as an extension in the future. Our implementation of *GSON* is abstracted to enable such adapters in the future without the need of extensive refactoring and resides in the *JSON Class* located in the utilities package.

A single *Task* can be written in any text editor, and should be saved with the ".json" extension. Such a <file> can be pushed to the smartphone device by utilizing the shell command presented in Code sample 7.3-ii[20].

```
adb push <file> /sdcard/Asper-Benchmark/Tasks
```

Code sample 7.3-ii Command to place a task on a smartphone device

[19] An Android compliant, Google supported, serialization library for JSON. See https://code.google.com/p/google-gson/

[20] It implies that the Android Debug Bridge (ADB) – utility is installed on the host machine. See http://developer.android.com/tools/help/adb.html

7.3.3 Generator

The *Generator* component is responsible for production of data items from a model imposed by the currently active *Task* in the system. The algorithm that performs the actual data item creation is presented in Code sample 7.3-iii.

Code sample 7.3-iii Data generation sequence

```
/**
  * Helper function for Data-item creation.
  * Parses a given Event type and generates
  * an Object[] representative of the event.
  *
  * @param event the event to model
  * @return a Data-item representation
  */
 private static Data create(Event event)
 {
    // Collection of event attributes
    List<Variable> variables = event.getVariables();

    // Data-item payload.
    Object[] dataset = new Object[variables.size()];

    /*
     * For each attribute, retain its value
     * and place it in the data-item payload
     */
    for (int i = 0; i < dataset.length; i++)
    {
       Variable variable = variables.get(i);
       Object value = variable.getValue();

       /*
        * Determines whether or not this
        * attribute is a range and should be
        * selected randomly.
        */
       if (variable.hasRange())
       {
          List range = variable.getRange();

          value = pickRandomNumber(
                (Number) range.get(0),
                (Number) range.get(1)
          );
       }
       dataset[i] = value;
    }
    return new Data(event.getName(), dataset);
 }
```

A *Coordinator* component will call a *Generators make()* command as noted in Figure 7.2.2-i. The make() command itself is responsible for sequencing the data items. It will append an attribute called

"SEQUENCE" at the end of each data item and enable reference from any installed query. Sequencing is – as a reminder, an important part of Window to table joins presented in chapter 7.1.4

Random number generation is performed by the "*pickRandomNumber*(n_{min},n_{max})" function that resides in the same component. This function follows, for verification, Equation 7.3-i and relies on "*java.util.Random"* with a fixed seed.

$$\min + \left(random * \left((max - min) + 1\right)\right)$$

Equation 7.3-i

Online generation of data items imposes problems that relate to production of enough objects per second without affecting the benchmark. A generation sequence requires us to reflect over how an event should be modeled and *Esper* itself, requires us to present a data item that corresponds to a pre-registered schema. It implies that name, datatype and order must be exact for the engine not to throw an exception.

Simple measurements indicate that the algorithm presented in Code sample 7.3-iii, is only capable of producing approximately 40 000 data items per second. A simplified version that creates a single randomly attributed data item and places it in an array – is capable of producing approximately 100 000 items per second at the cost of 15-20 % CPU utilization[21]. It could be stated that each task should be explicitly programmed to minimize the overhead of generation. However, we state that we are currently not able to control processor or thread affinity in *Android*, and cannot ensure that our *Worker* components are operating in isolation on separate processor cores.

Pre-allocating several million data items is unfeasible because a typical device is only capable of retaining 96 - 192 MB of heap memory. This limitation is possible to circumvent by manual configuration of the smartphone device – but several million inactive objects in the heap will affect garbage collection traversal and impose pause times that last up to a second (see chapter 4.2.4). Custom, large heaps are not presentable for the majority of devices that run *Android* and any results would prove of little use.

Object cloning is however, a feasible solution. Java is not capable of performing a shallow, or deep-copy of a plain object without additional assistance and implies that reflection must be utilized in order to create a generic solution. This situation is however different for arrays that contain primitives. Java is fully capable of performing a native array-copy operation and present the result as an individual object in the system.

We feel confident in proposing a solution where only a sub-set of the events that are to be processed, are generated once, and then looped through and cloned by each individual Worker. This ensures that Esper, and the Garbage Collector in Android is working with data items that do not retain any static references, and prove themselves collectable when discarded. Preliminary tests that measure the offset of throughput between instantiated and cloned data items, render a negligible difference[22].

The noticeable limitation, imposed by this decision, is that Task components should only contain primitive attributes. Using nested objects will imply that only references to these objects are copied.

[21] Task: 7.1.1, at 25% selectivity on hardware presented in chapter 8.1.1
[22] Task: 7.1.1 and Task: 7.1.5. The offset in time between cloning a three attributed array and instantiating it, is measured to be approximately 40 milliseconds for 100 000 arrays.

It also implies that only object-arrays are viable as a data item format. This however, proves to be a positive decision as object-arrays, are in chapter 8.2.2, rendered to be the fastest format for *Asper*.

7.3.4 Worker

A *Worker* component represents a single thread that performs the actual data item processing. Its procedure of execution is presented in Code sample 7.3-iv and renders a set of stages that the *Worker* goes through. The intention of this chapter is to render and verify that the *Worker* component itself, is of a simple nature, not imposing any substantial overhead or locking procedures. All variables mentioned in Code sample 7.3-v, are local the *Worker* instance. The *engine* variable represents an instance of *Espers EPRuntime Class* and represents an engine instance, derived directly from the *Asper* component.

```
/*
 * Main run procedure.
 * Initiated upon Thread start.
 */
public void run()
{
   try
   {
      storage.clear();

      // Waits for a notion about
      // data-repository content availability
      synchronized (Conditions.repositoryIsFilled)
      {
         Conditions.repositoryIsFilled.wait();
      }

      // Retrieves a calculated set of data-items
      storage.addAll(Repository.get());

      // Initiates warm-up procedure
      warmup();

      // Arrives at barrier related to
      // test invocation.
      Conditions.testCanStart.await();

      // Initiates the monitor timer
      monitor.reset();
      monitor.initiate();

      // Starts work procedure
      work();

   } catch (Exception e)
   {
        ...
   }
}
```

Code sample 7.3-iv Worker run procedure

The actual work() procedure is presented in Code sample 7.3-v. The warm-up procedure will simply perform the work() procedure for a specified amount of time by winding up a fixed timer that will cause the *Worker* to pause execution. The *Monitor* component is reset and initiated just before the actual test is about to begin.

```
/*
 * Performs the actual engine
 * feeding / processing procedure.
 * Loops continuously through a set of
 * data-items until paused.
 */
private void work()
{
   // Current storage index
   int index = 0;
   // Continuity indicator
   keepWorking = true;
   // Storage size
   int size = storage.size();

   while (keepWorking)
   {
      // Resets the index if
      // it surpasses the storage size.
      if (index >= size)
         index = 0;

      // Retrieves the next data-item
      Data e = storage.get(index++);
      // Sends / processes the event
      engine.sendEvent(e.data.clone(), e.name);
      // Notifies the monitor
      monitor.increment();
   }
}
```

Code sample 7.3-v Worker work procedure

7.3.5 Monitor

The *Monitor* component is responsible for retaining a notion of how many data items a *Worker* managed to process for a certain amount of time. Code sample 7.3-vi presents an outtake of the most important functions that the Monitor retains. The *initiate*() procedure will wind up a reoccurring timer that begins after one second, and reoccur every second[23]. The *timer* is implemented by "*java.util.Timer"* and leverages an internal timer daemon. Any components that rely on a reason of time, utilize the same procedure as this *Monitor*. We state that manual, continuous time retrieval through the use of "System.nanoTime()"[24] affects the measurements (4.6%±2.1) presented in chapter 8.3, as it imposes higher overhead than the timer. Time synchronization between components is not directly resolved. Instead, the Coordinator component will have a small

[23] Denoted by Settings.MONITORGRANULALITY, which by default, is set to 1000 milliseconds.
[24] Java Virtual Machine high-resolution time source.

offset in its temporal interval to ensure that it will not end the test before any of the *Monitor* components complete their cycles.

```
private volatile long i;
private Timer timer;
private TimerTask task;
private final Monitor self = this;
private LinkedList<Number> measurements;

/**
 * Initiates the Monitor timer. Ensures saving cycles.
 */
public void initiate()
{
   task = new TimerTask()
   {
      @Override
      public void run() {
         self.save();
      }
   };

   timer.scheduleAtFixedRate(
         task,
         Settings.MONITORGRANUALITY,
         Settings.MONITORGRANUALITY
   );
}

/**
 * Increments the local counter that withholds a notion of how many
 * data items a Worker has processed for a given interval.
 */
public void increment()
{
   ++i;
}

/**
 * Persists the current counter
 * value and resets it.
 */
private void save()
{
   measurements.add(new Long(i));
   i = 0;
}
```

Code sample 7.3-vi Monitor implementation

7.3.6 Measurements

A single test can consist of one or more individual runs. A single run is denoted as a complete cycle of the flow presented in chapter 7.2.2. Each run should retain the same operational environment, and must impose pause times between runs to ensure that a device is not affected by thermal constraints (see chapter 8.2.1). The Coordinator component will discard any Worker instances, and unregister any event types, queries and listeners that reside in context of Asper, before internally destroying the engine instance to ensure that each run retains the same environment. An explicit garbage collection procedure is initiated, but cannot be assured to run as denoted in chapter 4.2.4.

The collection of samples from each distinct Monitor component is persisted between individual runs. The average throughput is calculated for each individual Worker by utilizing Equation 7.3-ii, where *t* denotes a single Monitor sample that represents the amount of data items processed during an interval.

$$\overline{T} = \frac{\sum_{i=1}^{n} t_i}{n}, where \begin{cases} n = \ sample\ count \\ t = throughput\ sample \end{cases}$$

Equation 7.3-ii

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(\ t_i - \ \overline{t}\ )^2}{n}}, where \begin{cases} n = sample\ count \\ t = throughput\ sample \\ \overline{t} = mean\ of\ all\ samples\ in\ the\ set \end{cases}$$

Equation 7.3-iii

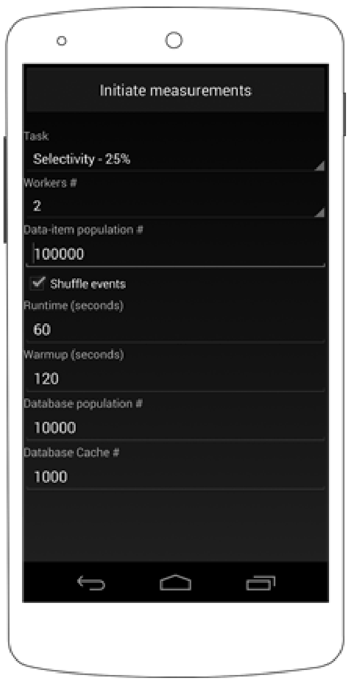
Deviation between the population of samples denoted by σ, is given by Equation 7.3-iii. Simply calculating the sum of data items processed over time, will, from our viewpoint not uncover information about garbage collection pauses and thermal effects over time, and potentially, present higher, average throughput rates. Persisted measurements can be retrieved through the following command:

```
adb pull /sdcard/Asper-Benchmark/Measurements
```

Code sample 7.3-vii Command to retrieve a measurement

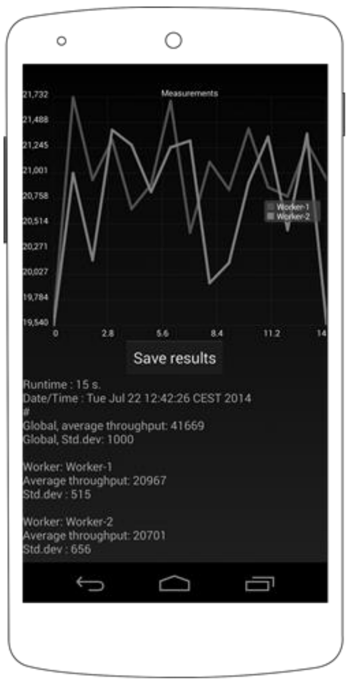
7.3.7 Interface

The application that encapsulates the system is presented with a simple interface that enables a given user to control how a test is conducted and to visually see how the engine performed under different circumstances.



Task
Selectivity - 25%
Workers #
2
Data-item population #
100000
✓ Shuffle events
Runtime (seconds)
60
Warmup (seconds)
120
Database population #
10000
Database Cache #
1000

a.



```
Runtime : 15 s.
Date/Time : Tue Jul 22 12:42:26 CEST 2014
#
Global, average throughput: 41669
Global, Std.dev: 1000

Worker: Worker-1
Average throughput: 20967
Std.dev : 515

Worker: Worker-2
Average throughput: 20701
Std.dev : 656
```

b.

Figure 7.3.7-i Benchmark interface

Figure 7.3.7-i-a displays the main screen that is rendered upon initialization and reflects the *Main Activity* from Figure 7.3.1-i. A user is able to select the task that is to be performed, and if multiple tasks are identified by the system, then all of them will be present in in a drop-down list. *Data item population* decides the amount of items to generate and distribute amongst the *Workers* (excluding events that are to be preloaded). *Database population* decides the amount of rows that are to be populated in a given table. *Database cache* decides how big an engines *LRU* result cache should be.

Figure 7.3.7-i-b represents the interface that is rendered when a test sequence is completed and represents the *Reporter Activity* from Figure 7.3.1-i . A simple graph shows the effective throughput of each *Worker*. The y-axis represents data items and the x-axis represents time in seconds. We state that that a graph is of use in our context as it enables a user to see if the results are affected by frequency throttling and temperature over time. We visit this issue in chapter 8.2.1 – and state that devices behave quite differently.

7.4 Distribution and Availability

This application and an implementation of each *Task* described in chapter 7.1 is present as an open sourced project, located in Repository 2 below.

https://github.com/mobile-event-processing/Asper-Benchmark

Repository 2 Asper Benchmark

The project structure consists of two main folders. “*Measurements”,* that contain a set of *JSON* based *Task* implementations – and “*Application”,* which contains the source code of this system.

Task instances must be pushed to a device using the command presented in Code sample 7.3-ii. We state that this is a sensible model for distribution as each *Task* retain its own alteration history.

Storing *Task* files in the source code will imply that a complicated writing procedure upon application installation.

III. Evaluation and Conclusion

8 Benchmark Measurements

This chapter describes experiments and measurements related to the set of questions posed in the requirement analysis presented in chapter 5.2. The first section presents the domain of our measurements and highlights objectives that must be met in order to render our results true. Section two includes a set of preliminary experiments that aid in understanding of how Esper behaves on an Android based device and describes influential factors that affect our measurements. Section three presents measurements and results that concern the task presented in chapter 7.1. This chapter is then concluded with a discussion that concerns the feasibility of utilizing Esper on Android.

8.1 Domain

The primary intention of this experiment is to utilize our implementations from chapter 7 for the purpose of determining the maximum achievable throughput for each task presented in chapter 7.1.

A preliminary set of experiments must be conducted in order to answer the questions posed in the requirement analysis (see chapter 5.2). Specifically, we want to answer how thermal constrains and garbage collections can influence the performance of Asper in negative ways. Executional pauses imposed by sizable windows could influence deviations, and a preliminary study should answer how these procedures are conducted in Dalvik and ART. Our alteration of CGlib (see chapter 6.2) imposes the use of reflection instead of attribute indexing. We therefore state that a comparison between distinct data item formats must be in place in order to render if our choice is affected severely. These properties are regarded as influential factors that should aid in system design when Asper is present.

8.1.1 Hardware and Software

All set of experiments are conducted on a Nexus 5, LG-D821 smartphone device. The choice of device is reflected by its support of both ART and Dalvik as very few devices currently retain the possibility of incorporating both environments.

Hardware and software specifications are presented in Table 8.1-i. Runtime environment settings are presented in Table 8.1-ii and retain default values. The possibility of overriding default values is present on unrestrained devices. Our intention is however, to retain default values as they are representable for a larger majority of existing devices.

This device is under all tests set into airplane mode to ensure that no transceivers are operational. Only stock background processes are running, as uninstallation and disablement causes exceptions from what we believe, are daemon processes.

Component	**Description**
SoC	Quad-core, 2.26 GHz Qualcomm Snapdragon 800, Model 8974-AA. 2x 16 KB L1 Cache, 2x 2 MB L2 Cache.
Main Memory (RAM)	2 GB LPDDR3-1600 RAM. 800MHz 32-bit dual-channel. (12.8 GB/s)
Local storage memory	Sandisk SDIN8DE4 16 GB NAND flash
Battery capacity	3.8 V, 2300 mAh
Android, version	Android 4.4.2. Build KOT49H

Kernel, version	3.4.0-gadb2201

Table 8.1-i Nexus 5 device specifications

Property	Value	Description
dalvik.vm.heapstartsize	8 MB	Specifies the initial size of the managed heap
dalvik.vm.heapgrowthlimit	96 MB	Specifies the maximum size a standard application is allowed to allocate
dalvik.vm.heapsize	192 MB	Specifies the maximum size an application is allowed to allocate if it is set with the property android:largeHeap in the manifest document.
dalvik.vm.heaptargetutilization	0.75	Imposes how full (%) the managed heap should be before it needs to be increased.
dalvik.vm.heapminfree	512 KB	Lower bound soft limit garbage collector heuristic
dalvik.vm.heapmaxfree	8 MB	Upper bound soft limit garbage collector heuristic

Table 8.1-ii Nexus 5, runtime environment settings

8.2 Preliminary Experiments

This section presents the collection of influential factors noted in chapter 8.1.

8.2.1 Thermal Effects

Smartphone devices are, as stated in chapter 5, not actively cooled and could affect the SoC performance notably under long and intensive processing operations. Understanding how a test device scales its operating frequency should aid us in understanding how Asper is affected over time. Observing the SoC temperature readings both under, and after an intensive task, should present how much time a specific device needs in order to recover from an intensive operation.

We measure this effect by utilizing a simple application that places a set of threads under a continuous, high workload, effectively placing all processor cores at approximately 90% load. A separate thread conducts frequency and temperature sampling at a rate of 10 samples per second, for 600 seconds, before it interrupts the working threads and resumes sampling of temperature and frequency readings for another 300 seconds.

Frequency samples are derived on a per core basis from the following endpoint on each device:
/sys/devices/system/cpu/cpu[core]/cpufreq/scaling_cur_freq
Temperature samples are for the Nexus 5 derived from:
/sys/class/thermal/thermal_zone7/temp

It must be noted that we are currently not able to confirm that the mentioned thermal endpoints, are in fact, representing the SoC temperature. Attempts to query official vendor forums have yielded in no statements that could be presented as answers. We see however, that the samples correspond to operational temperatures located in the device service-manual. Android is itself, from our perspective, not instrumented with any standardized, higher level API endpoint that derive this reading.

Figure 8.2.1-i and Figure 8.2.1-ii presents measurements that utilize one or four processor cores simultaneously. The left y-axis represents the frequency for each individual core, and the right y-axis represents the SoC temperature in Celsius.

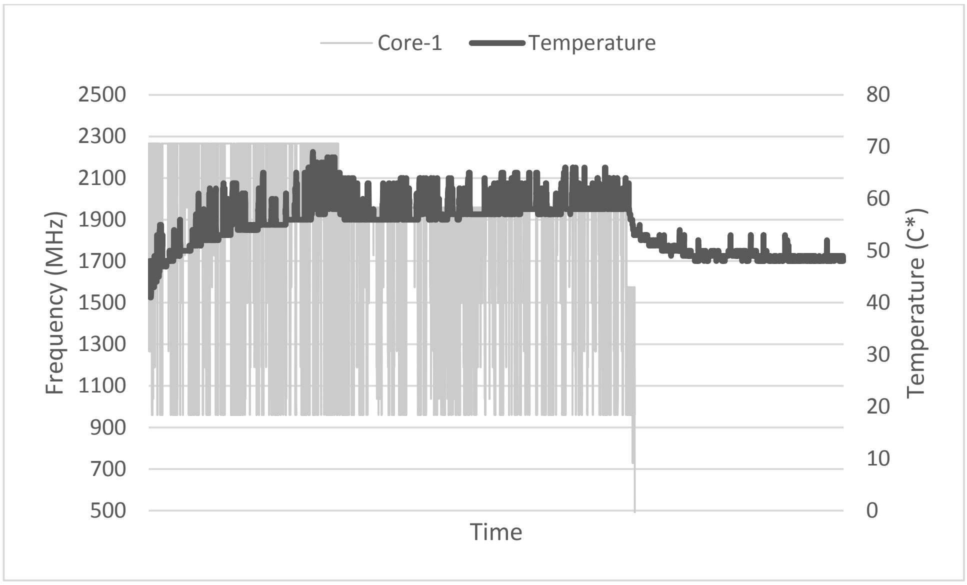


Figure 8.2.1-i Thermal effects, single core utilization.

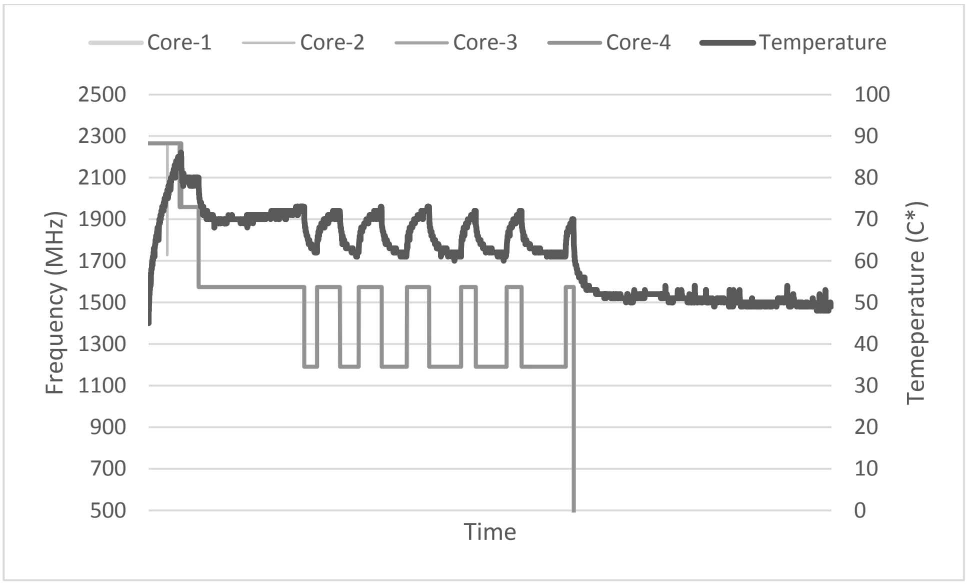


Figure 8.2.1-ii Thermal effects, quad-core utilization.

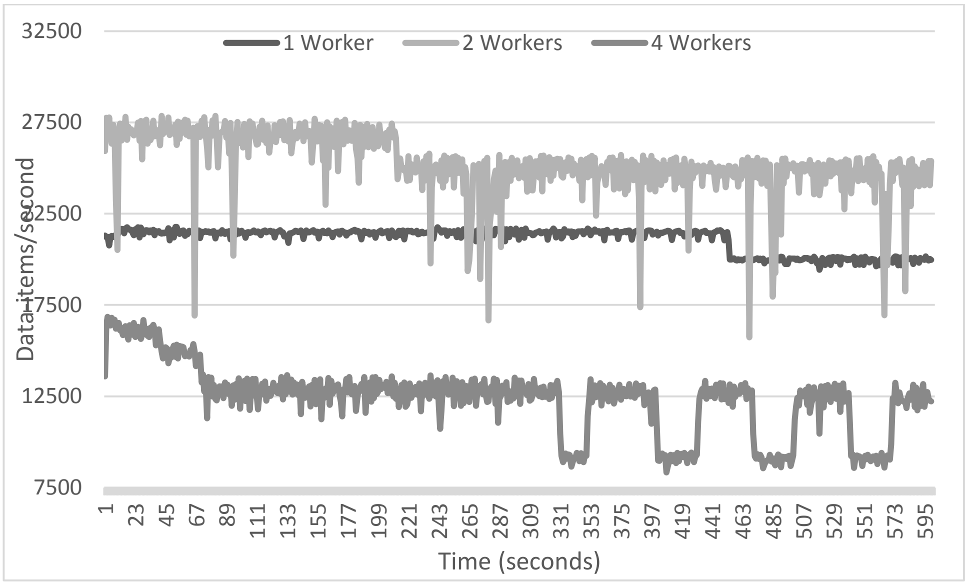


Figure 8.2.1-iii Throttling of throughput imposed by frequency scaling

Placing pressure on all four cores simultaneously forces the SoC to throttle its frequency drastically in order to remain an operational temperature. The frequency scaling strategy is shifted profoundly between measurements in Figure 8.2.1-i and Figure 8.2.1-ii, where a single core will continuously scale the frequency between 2265.6, 1574.4 and 1267.2 MHz until it drops after approximately 240 seconds. We see, by utilizing the Top[25]-command and disabling our internal measurer, that dual cores are activated when the test is initiated, and that processing is moved between both cores on several occasions. Measurements in Figure 8.2.1-ii miss such an ability as all four cores are occupied, and present and an acute, yet stable drop to 1574.4 MHz after approximately 50 seconds. It retains this operation for another 110 seconds, before it starts to throttle in a predictable wave-pattern.

Temperature readings retain stability after approximately 85 seconds, but are likely influenced by the active thread that still queries the device every 100 milliseconds.

Figure 8.2.1-ii renders that highly intensive, multi-threaded event processing will be affected over time, and that one could expect a performance drop of up to 83% for certain periods of time. Deviation between measurements that regard Asper, could be presented as a result of frequency scaling rather than engine design. Any results presented in chapter 8.3, should be interpreted with the notion that performance may degrade over time.

Error! Reference source not found. presents task 8-a (see chapter 7.1.8) with a selectivity predicate of 5%, over an interval of 600 seconds with 1, 2 and 4 Workers. It verifies that high deviation in certain, long running measurements, could simply be cause by frequency scaling, and not by Asper itself. Throttling sequences start later because continuous processing of such data items imposes an average toll of 70-80% on the processor. Lower levels of processed data items per second for more than one Worker, is discussed in chapter 8.3.9.

[25] Linux based shell command for statistics that concern the most consuming, current processes.

8.2.2 Collection and Heap Allocation

We state, in chapter 4.2.4, that operations related to garbage collection and heap allocation cause executional pauses that effect a threads ability to perform work. Queries that utilize sizable windows, enforce a policy of retainment, and require allocation of space over time. Queries that produce output, will enforce collection procedures to de-allocate the memory space once a receiver is finished processing it. Understanding how long such pauses can be, aids in the decision of choosing window sizes for tasks presented in chapter 7.1, and renders how Dalvik differs from ART in our context. A query that represents a sizable window and enforce continuous output generation, can be presented as follows:

*SELECT * FROM Event$_a$.win:length(W$_n$)*

Query 8.2.2-i

Figure 8.2.2-i and Figure 8.2.2-ii presents a series of measurements that utilize Query 8.2.2-i , on a single Worker, for 240 consecutive seconds. W_n takes, in Query 8.2.2-i, a value that ranges from 5k to 500k[26], and effectively decides the window size. All pause times are derived by observing log messages generated by the runtime environment (see chapter 4.2.4), and presents the total amount of time spent for collection. Each form represents an occurrence of a garbage collection operation.

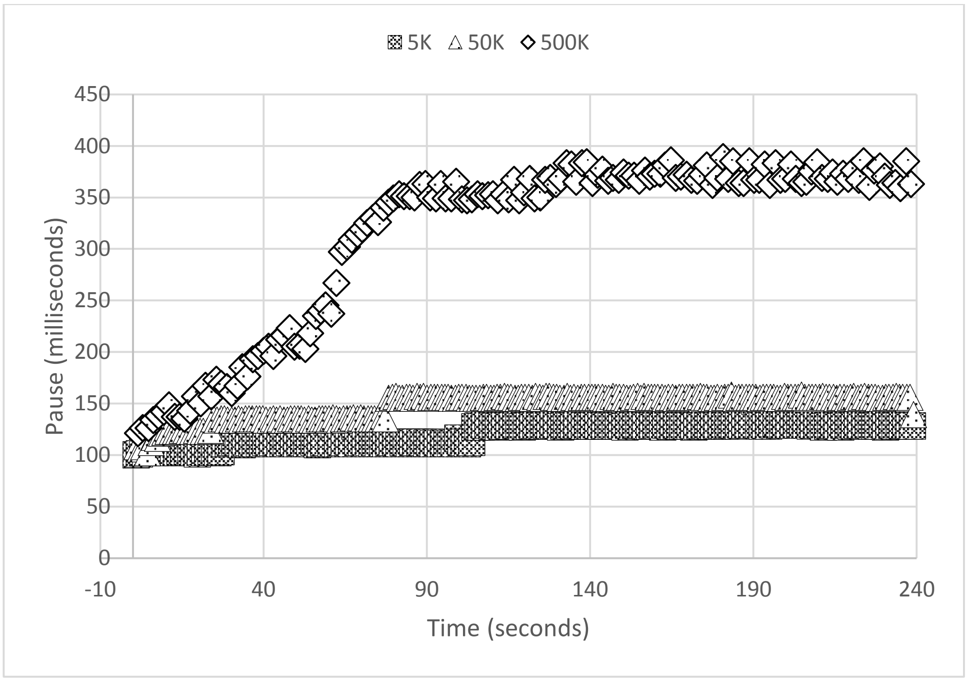


Figure 8.2.2-i Pause times imposed by window size on Dalvik

[26] Attempts to set W_n of 5M resulted in application termination because of exhausted heap resources.

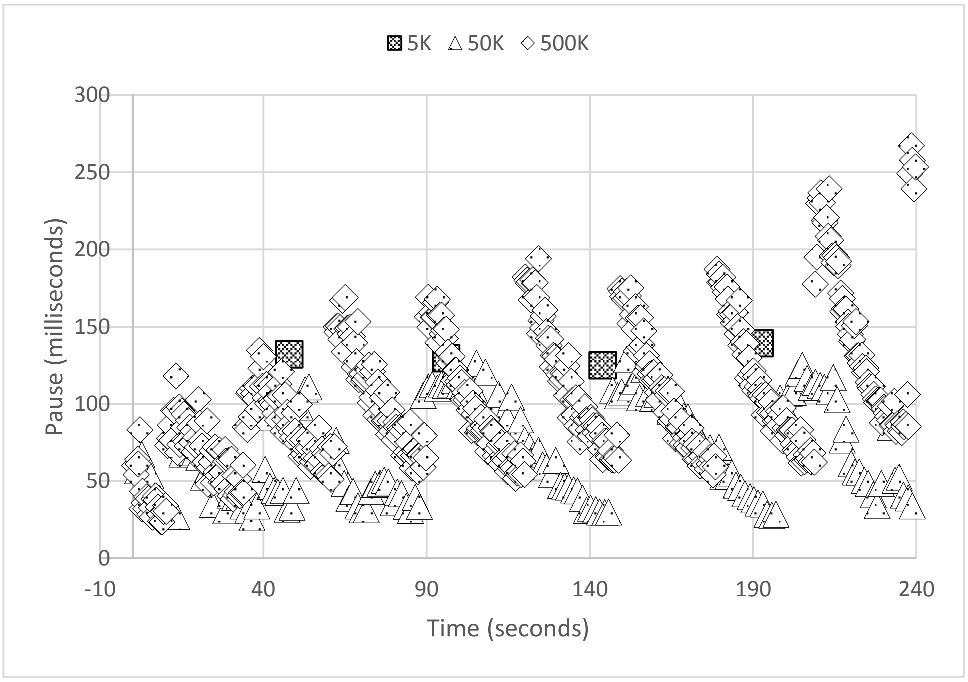


Figure 8.2.2-ii Pause times imposed by window size on ART

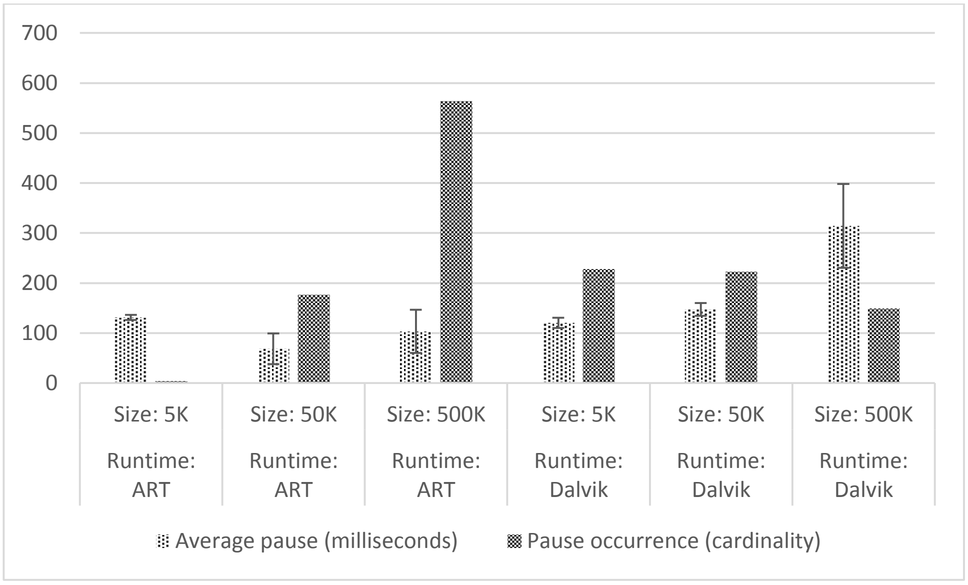


Figure 8.2.2-iii Correlation between pause times and window sizes for Dalvik and ART.

Figure 8.2.2-iii derives its values from Figure 8.2.2-i and Figure 8.2.2-ii. The y-axis presents, for the average pause columns, the amount of milliseconds a pause lasts, and the amount of such pauses for the occurrence columns. The figure implies that ART embeds an improved garbage collector. Windows sizes that retain 5K data items, present only 4 occurrences of collection while Dalvik imposes 228 occurrences for the same sequence. This difference is profound as Dalvik would pause processing of data items for, on average, 100 milliseconds, once every second. The garbage collector in Dalvik could prove to be more efficient for window sizes that surpass 50K. Overall pause times are approximately 24% lower than for ART during the same time span, but inflicts pauses that last up to 400 milliseconds.

We mention in chapter 4.2.4 that threads would, for most part, attempt to run their collection procedures concurrently, and only impose stop-the-world pauses upon verification, or when no sizeable slot is available in the heap. Dalvik imposes, almost exclusively; only GC_FOR_ALLOC operations opposed to ART, and struggles to maintain sizable windows. Queries could be limited from presenting answers in a timely manner upon high bursts of incoming data. ART is less conflicted about this manner because it retains a separate heap for large allocations. Stop-the-world pauses are for windows that retain a size of up to 50K, only 5.1 milliseconds, and for 500K, only 25.4 milliseconds on average. It implies that other threads can run will less disturbance.

Discussions that regard jumping windows, are presented in chapter 8.3.4.

8.2.3 Event Format

The choice of a data format that represents data item instances is profoundly influenced by the implementation of our Generator component, presented in chapter 7.3.3. How performance is affected by the choice of a format, is however, of interest to study as we operate on a resource-constrained device. We state, in chapter 6.2, that we alter the way plain Java objects are interpreted by the engine. Utilization of reflection, opposed to indexes, should have an impact on the amount of data items an engine processes as each attribute lookup is performed by costly name comparisons.

We utilize the projection query from task 2, and configure it to utilize three distinct attributes. The Generator component is itself, instructed to generate data items that present an object-array, plain object or a string based key-value map. Worker components are instructed to not clone data items as this is proven problematic for plain objects (see chapter 7.3.3)

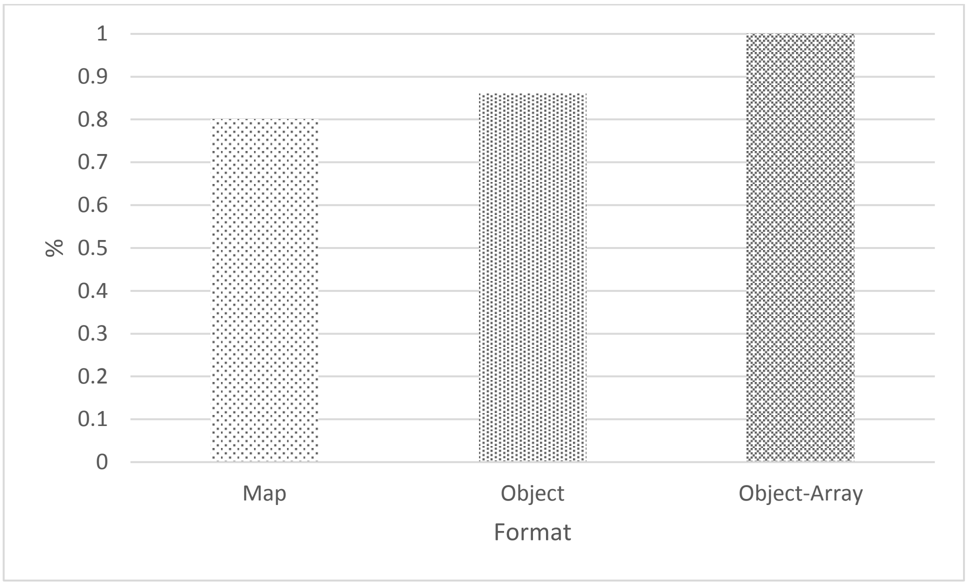


Figure 8.2.3-i Performance implications caused by the choice of a data format

Figure 8.2.3-i presents a single, 60 second measurement, on a single Worker, with ART as the runtime environment. Our measurements render an improvement in the amount of data items processed per second, and present a difference of 19.6 % between a key-value map and an object-array and 13.4 % between a plain object and an object-array. An improvement between the use of plain object and a map proves as an interesting result because it was initially expected that a Java based map embedded lookup algorithms that were more effective than attribute traversal imposed by our alteration of CGlib.

8.3 Measurements

This section presents measurements that concern the tasks in chapter 7.1. It begins by describing the general setup and follows by a presentation of each individual task.

8.3.1 Setup

All measurements are conducted with the benchmark application presented in chapter 7. Each task is implemented, configured and expressed through JSON, before being loaded onto the device. Parameters for each task are set individually and result in over 50 distinct tests that each exercise an unique factor. Each distinct test is conducted in three repetitions on both Dalvik and ART in order to render the difference between the runtime environments. Switching between runtime environments implies that the device recompiles every application that is present. No difference in terms of stability or performance is seen from this procedure.

8.3.1.1 Time and Rest

Warmup procedures and measurement procedures retain equal length.

Results presented in conjunction to thermal effects (see chapter 8.2.1) render that test durations can influence the performance over time. This will present higher deviations in the measurements and generate noise that could be difficult to distinguish from how Asper behaves.

We state that it is sufficient for a test that embeds only a single Worker, to perform a warmup and measurement procedure that lasts for 60 seconds (each). This is however different when the number of Workers increase or when temporal contexts are used. Separate measurement times are described in each task that operates in such context.

The device is instructed to rest for 120 seconds between each repetition in order to ensure that it is properly cooled down.

8.3.1.2 General Parameters

Task parameters are toned down because of the memory and processing capabilities that reside on a smartphone device.

In general, 100k unique data items are generated before each test sequence. This imposes an allocation of 46,2 MB. Patterns and joins between windows are highly dependent on random number generation. A fixed seed is present but twice the amount of random numbers should be picked to ensure correctness when matching that requires 100 % selectivity is performed as lower numbers present some fluctuations between tests. We observe that allocation of more than 100k affects test measurements because of considerably high deviations caused by allocation procedures[27]. This implies that window sizes must at most; retain a size of 50k.

8.3.1.3 Presentation

All measurements are calculated by the procedure described in chapter 7.3.6. $\overline{T}$ denotes the average throughput per second as expressed by Equation 7.3-ii. All results are presented as the average amount of data items processed per second, for all repetitions combined.

[27] Measured on Task 1, with 25 % selectivity on Dalvik.

Each task contains a comparison between distinct exercise factors, and represent these in the percent increase or decrease in the format $\Delta \pm \sigma$ where:

$$\Delta = \frac{(\overline{T}_b - \overline{T}_a)}{\overline{T}_a} x\ 100$$

$$\sigma = cv_a + cv_b, where\ cv_x = \frac{T\sigma_x}{\overline{T}_x} x\ 100$$

Equation 8.3-i

8.3.2 Task 1: Selectivity

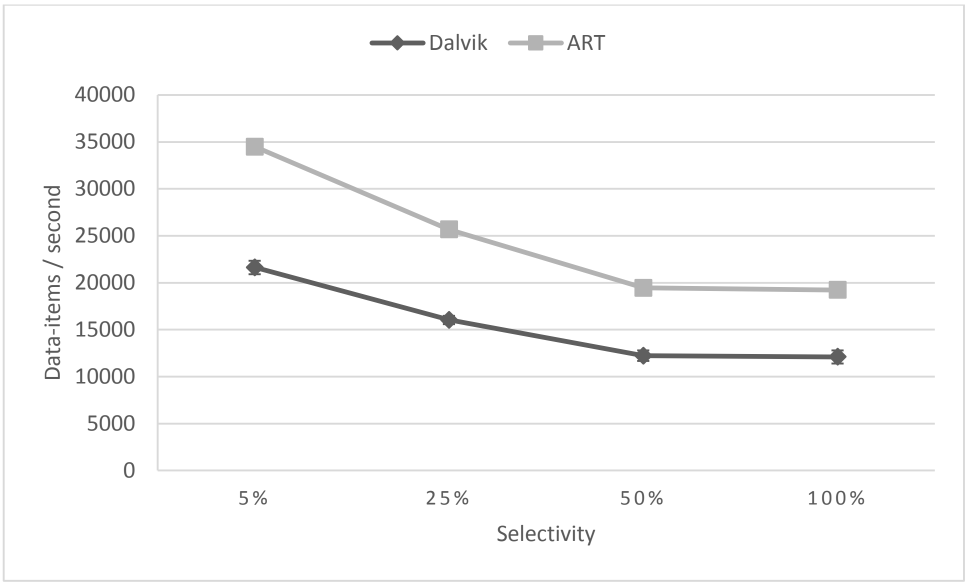


Figure 8.3.2-i Selectivity measurements

Figure 8.3.2-i represents Query 7.1.1-i and Query 7.1.1-ii. Event$_a$.id takes the range 1-100k. Predicate selector *K* takes the values 5k, 25k and 50k to render a selectivity of 5, 25 and 50%. Query 7.1.1-ii imposes no filtration and ensures 100% selectivity.

Δ Selectivity %	**Dalvik %**	**ART %**
5 – 25	-25.8 ± 6	-25.5 ±1.7
25 – 50	-23.7 ± 7.4	-24.2 ± 1.9
50 – 100	-1 ± 10.4	-1.1 ± 1.8

Table 8.3-i Change in throughput for selectivity

Table 8.3-i presents the negative effect an increase in selectivity has on the throughput. It is clearly indicated that there little or no difference between Dalvik and ART when windows or functions are absent as both retain a stable decrease in performance. Figure 8.3.2-i indicates that removing filtration imposes that processing will not be affected by the approximate drop of 25%. It is considerably cheaper to not impose any filtration, if the main interest lies within extraction of attributes from the arriving data items.

Selectivity %	***Δda* %**
5	59.3 ± 4.0
25	59.s9 ± 3.7
50	59 ± 5.6
100	58.8 ± 6.6

Table 8.3-ii Difference between Dalvik and ART for selectivity

Table 8.3-ii presents the difference in throughput between Dalvik and ART for each parameter. It identifies a baseline that renders how well ART outperforms Dalvik when only pure processing is needed. Low deviation between the measurements is indicated by the fact that the engine is not required to withhold any data in windows, and Dalvik manages to perform its garbage collection procedures without a profound amount of GC_FOR_ALLOC incidents.

8.3.3 Task 2: Projection

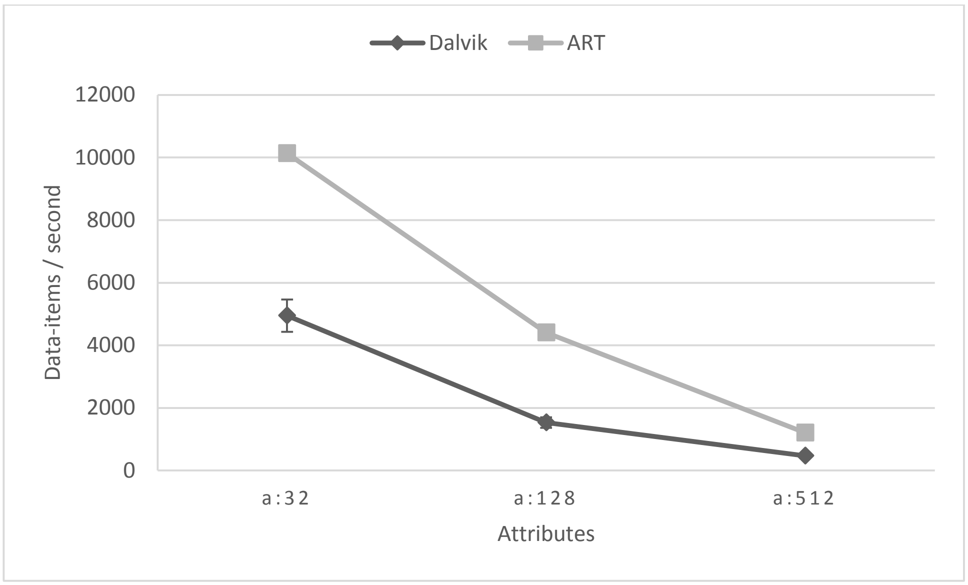


Figure 8.3.3-i Projection measurements

Figure 8.3.3-i represents Query 7.1.2-i. The amount of data items generated is lowered to 1k to accommodate the increased size that a high number of attributes impose. Cardinality of projected attributes are presented on the x-axis.

Δ Attributes	**Dalvik %**	**ART %**
32-128	-69 ± 21.7	-56.6 ± 4
128-512	-69.1 ± 23.9	-72.6 ± 8.2

Table 8.3-iii Change in throughput for projection

Table 8.3-iii presents how an increase in the cardinality of attributes affects the overall throughput. Projection is in this context, performed by array-traversal and indicates that more time is spent on looping and index lookup rather than processing of events. Garbage collection procedures come in affect for Dalvik when 32 attributes are projected as more data items must be collected at the output channels, and thereby cause deviations, but these collection procedures seize to exist.

Attributes	***Δda* %**
32	104.8 ± 12.2
128	187.7 ± 17.3
512	155.5 ± 18.6

Table 8.3-iv Difference between Dalvik and ART for projection

Table 8.3-iv presents the difference in throughput between Dalvik and ART for each attribute cardinality. The increase in performance between 32 and 128 attributes is difficult to explain, but we question the presence of JIT-optimization for Dalvik as ART presents a remarkable performance gain. Profiling method calls indicates that Asper performs more than 35 distinct calls to perform the primitive task of projection. The short time spent in each area could impose a toll that prevents Dalvik from constructing an efficient translation cache. We therefore believe that other areas (arithmetic operations, temporal comparisons) could be equally effected, rendering Dalvik less suitable overall.

8.3.4 Task 3: Aggregation and Window Type

This section is separated into three distinct sub-sections that each present an aggregation function, denoted by the sub-section name. Results include both sliding and jumping window types to indicate the difference on performance that they impose. All measurements utilize Query 7.1.3-i. Window size W_n takes the value 500, 5k and 50k and each and every data item that enters the stream is outputted to a single, awaiting listener that retrieves the calculated value of the aggregation function $f(a_x)$.

8.3.4.1 Sum

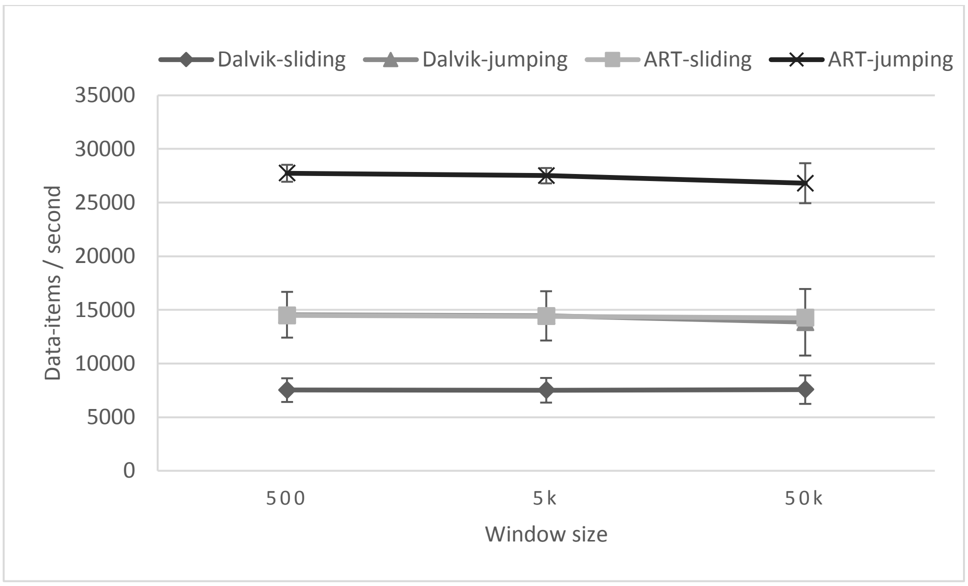


Figure 8.3.4-i sum(); aggregation measurements over different window types

Figure 8.3.4-i presents the SUM () function over a sliding and jumping window. Distinction between both window types is profound as a sliding window imposes eager evaluation and produces answers continuously as new data items enter the window. A jumping window will retain its data items until it expires and perform the same calculations in a batch operation, effectively creating an offset between sizable allocations of heap space, and continuous cleanup of processed data items.

Δ Window size / type	**Dalvik %**	**ART %**
500 / sliding-jumping	93.3 ± 29.7	91.4 ± 4.2
5k / sliding-jumping	92.2 ± 31.1	91 ± 6.1
50k / sliding-jumping	82.7 ± 39.8	88 ± 10.2

Table 8.3-v Change in throughput for sum() and distinct window types

Table 8.3-v presents the increase in throughput between a sliding and a jumping window with respect to each window size. We mention in chapter 7.1.3 that the SUM() function should compute its answers at fixed cost, independently of window length and policy. However, it becomes clear that window sizes over 5k are subjected to garbage collections that impose throughput variability. The main distinction lies however between Dalvik and ART, where Dalvik presents that throughput may deviate by over 30% for jumping windows.

Window size / type	***Δda* %**
500 / sliding	92.7 ± 12.4
5k / sliding	91.7 ± 18.7

50k sliding	87.9 ± 20.7
500 / jumping	90.7 ± 18.5
5k / jumping	90.4 ± 18.4
50k / jumping	93.4 ± 29.3

Table 8.3-vi Difference between Dalvik and ART for sum()

Table 8.3-vi presents the increase in throughput between Dalvik and ART for each window type and size and renders that ART imposes near twice as high throughput for each combination. Discussions regarding jumping windows and garbage collections are present in chapter 8.3.4.2 below.

8.3.4.2 Median

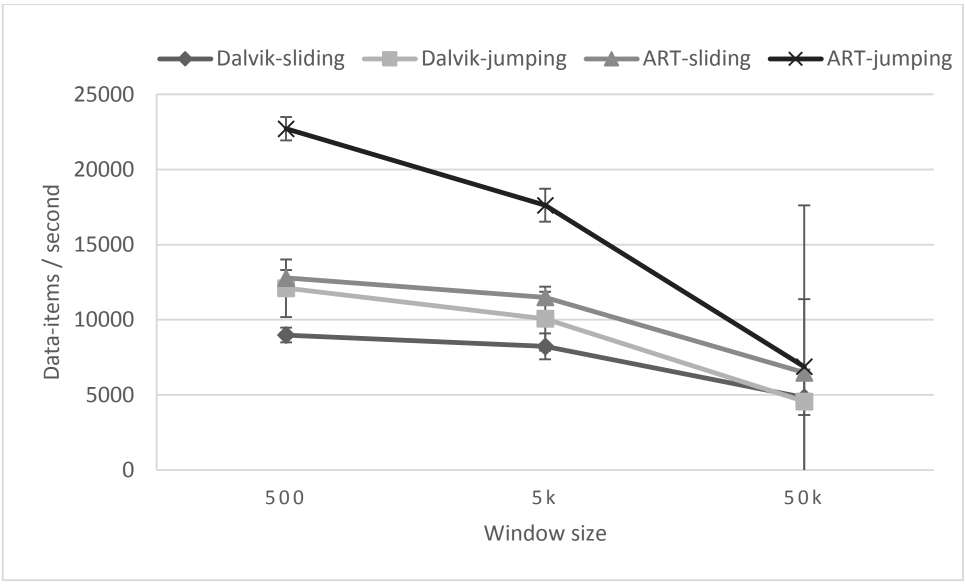


Figure 8.3.4-ii median(), aggregation measurements over different window types

Figure 8.3.4-ii presents the MEDIAN() function over a sliding and jumping window. We state that calculation of the median is cannot be performed at fixed cost for either window type, and this exercise renders how profoundly throughput can drop as the window size increases.

Δ Window size / type	**Dalvik %**	**ART %**
500-5k / sliding	-8.4 ± 15.9	-10 ± 7.3
5k – 50k / sliding	-41.2 ± 34.8	-43.6 ± 6.4
500-5k / jumping	-16.8 ± 36.9	-22.4 ± 9.6
5k – 50k / jumping	-54.7 ± 170.7	-61 ± 162.5

Table 8.3-vii Change in throughput for median() and distinct window types

Table 8.3-vii presents how an increase in window size, effectively decreases the throughput. The profound distinction lies between a window size of 5k and 50k, where both window types drastically drop by over 40%. We initially believed that this effect is influenced by executional pauses from the garbage collection procedures that we mention in in chapter 8.2.2. This however, renders untrue when comparing the garbage collection messages.

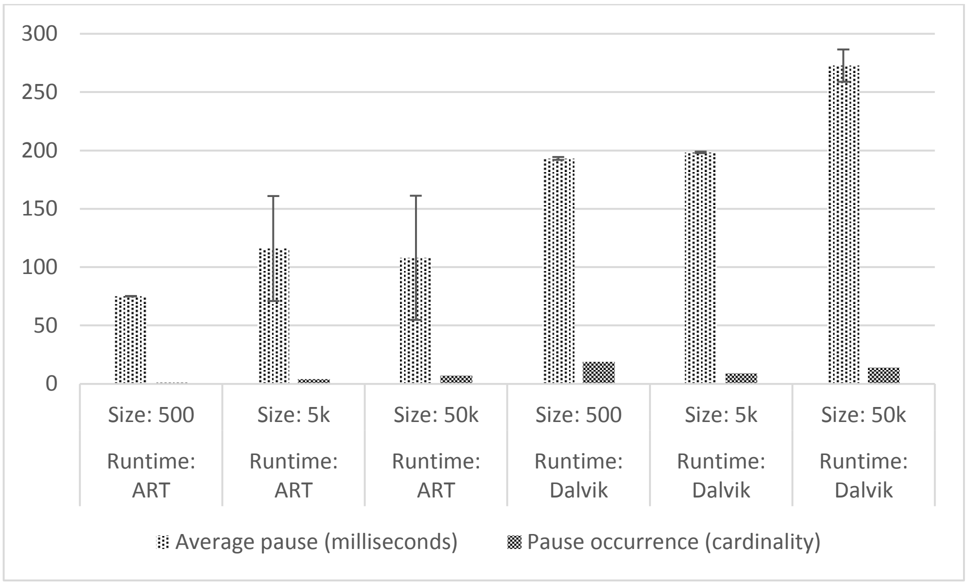


Figure 8.3.4-iii Garbage collection occurrences for median over a jumping window

Figure 8.3.4-iii presents garbage collection samples derived from 30 seconds of MEDIAN() on a jumping window of size 500, 5k and 50k. The y-axis presents, for the average pause columns, the amount of milliseconds a pause lasts, and the amount of such pauses for the occurrence columns. The cardinality of pauses is low opposed to a sliding window, and implies that our measurements should not be affected so profoundly. Visualization of the measured throughput rendered no distinct deviation until we lowered the level of granularity that our Monitor component operated on (see chapter 7.3.5) from 1 sample per second, to one sample every 100 milliseconds.

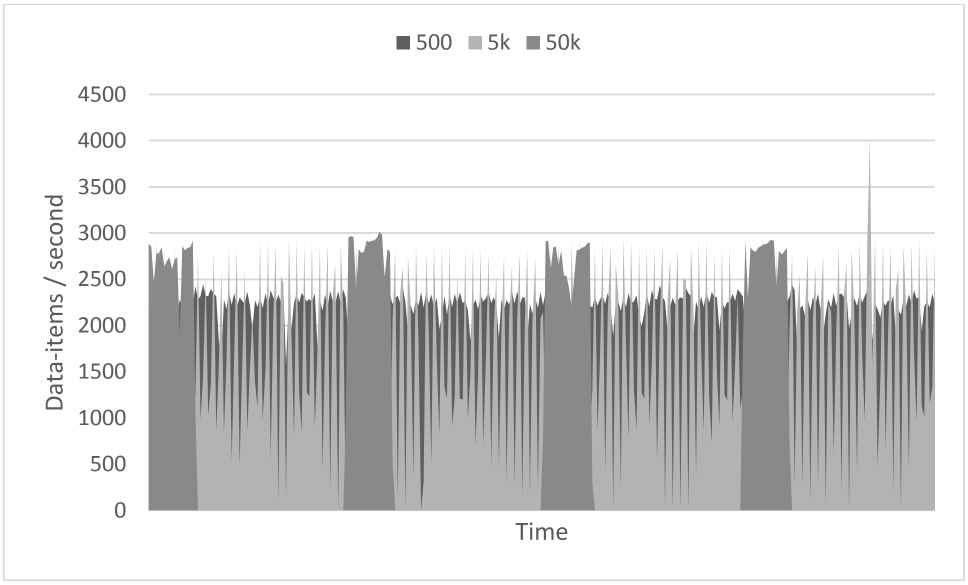


Figure 8.3.4-iv Throughput over time for median over a jumping window

Figure 8.3.4-iv presents the throughput every 100 milliseconds for the same 30 second run, and each shade of gray represents a distinct window size. This implies that the high deviations from Figure 8.3.4-ii are caused by computational pauses imposed by the engine for windows with higher size than 5k. Data item feeding is blocked for long periods of time until computations are performed and answers are given. Espers documentation state that only aggregate values are kept and updated as new data items enter the stream[28]. This could explain why the garbage collection procedures are not affecting our measurements, but implies that that the computational complexity of locating the median value of a set, is distinctively different from locating the sum of all values. This could imply that custom aggregate functions could suffer from the same endeavor, and that blocking, even for smaller windows, could prove as the main cause for data items repository overflows.

Δ Window size / type	**Dalvik %**	**ART %**
500 / sliding-jumping	34.7 ± 21.3	77.6 ± 7.4
5k / sliding-jumping	22.4 ± 31.6	53.3 ± 9.4
50k / sliding-jumping	-5.6 ± 173.2	6.1 ± 159.5

Table 8.3-viii Difference between window type and throughput for median()

Table 8.3-viii presents the increase or decrease in throughput between a sliding and jumping window with respect to each window size. The increase in throughput is near half of what the SUM() function presented on the same setup, and the stability of a 50k sized window makes this comparison near impossible to render.

Window size / type	***Δda*** **%**
500 / sliding	42.3 ± 15.8
5k / sliding	39.7 ± 19.4
50k sliding	33.9 ± 23.8
500 / jumping	87.6 ± 19.2
5k / jumping	75 ± 27.3
50k / jumping	50.6 ± 305.9

Table 8.3-ix Difference between Dalvik and ART for median()

Table 8.3-ix presents the difference between Dalvik and ART for each window size and type and renders that ART is over 35% more performant on sliding windows and near to 80% more performant when jumping windows are utilized.

[28] http://esper.codehaus.org/tutorials/faq_esper/faq.html#keep_in_memory

8.3.5 Task 4: Window to Table Join

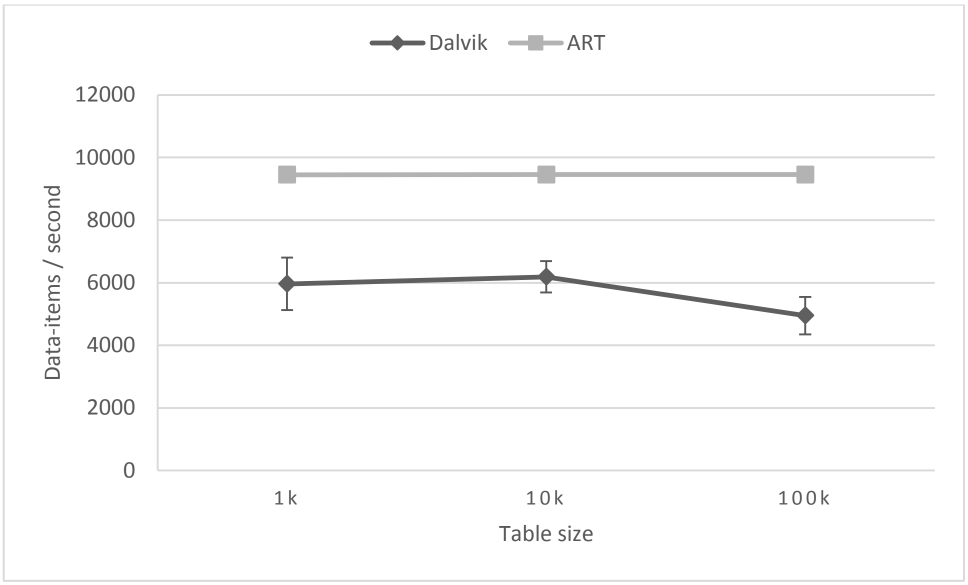


Figure 8.3.5-i Table join measurements

Figure 8.3.5-i presents measurements that concern joins between a pre-populated window of size 1k, 10k, and 100k. Query 7.1.4-i ensures that a match is found for every arriving data item that enters the stream and implies that listeners retrieve information continuously.

Δ Table size	**Dalvik %**	**ART %**
1k – 10k	3.7 ± 22.1	0.1 ± 2.4
10k – 100k	-20 ± 20.1	0.1 ± 2.2

Table 8.3-x Change in throughput for table joins

Table 8.3-x presents the decrease in throughput for each table size and immediately implies that Dalvik suffers from continuous generation of output. Attempting to lower the chance of finding a match in the table (down to 10%) will result in profoundly more stable throughput rates, and match what we see for ART. We believe that Dalvik must traverse the whole chain of preloaded data items in order to find whether or not they are considered dead as projection of joined data enforces small and continuous allocations and this could indicate that Dalvik is not directly able to efficiently annotate recently swept heap areas. Joining information from a static window can be performed at fixed cost. Attempting to increase the number of attributes that should be joined to 10, will not present any profound distinction, and could indicate that Esper / Asper performs some attribute indexing in order to optimize matching.

Table size	***Δda* %**
1k	58 ± 15.3
10k	52.7 ± 9.2
100k	91.1 ± 13.1

Table 8.3-xi Difference between Dalvik and ART for table joins.

Table 8.3-xi presents the difference in throughput between ART and Dalvik for each table size. ART is, as seen, not affected by the issues that Dalvik imposes, and present a stable environment of operation. Effectively rendering a throughput of more than 50%.

8.3.6 Task 5: Window to Window Join

This section presents two distinct tests that exercise window to window joins from different perspectives. The first test varies the window size, but retains probability of locating matching events at 100%, and the second test retains a fixed window size and exercises how matching and output generation affects the overall performance.

8.3.6.1 A. Variable Window Size and Fixed Selectivity

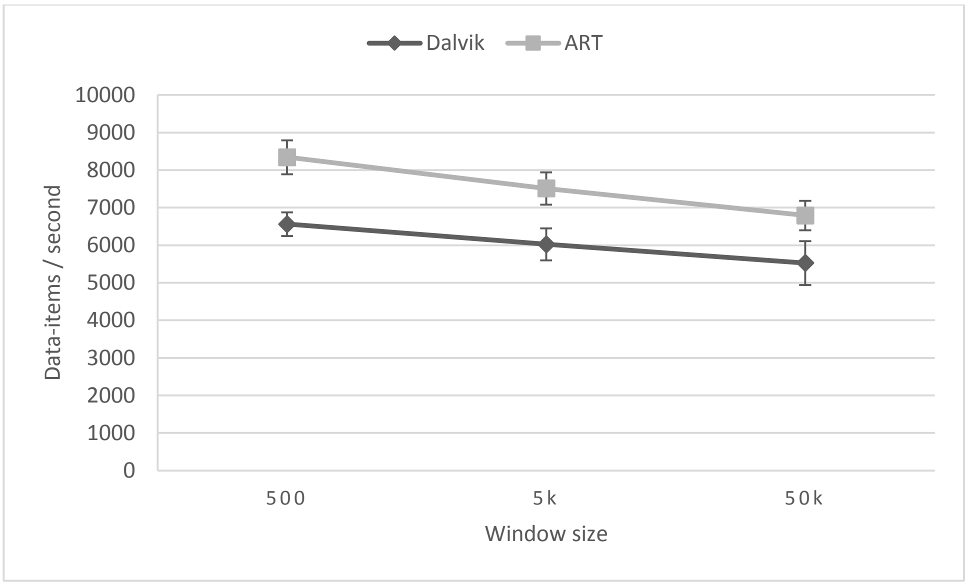


Figure 8.3.6-i Measurements concerning window to window joins with fixed selectivity

Figure 8.3.6-i represents Query 7.1.5-i over a sliding window where the size W_n and the identifier range I_n of Event$_a$ and Event$_b$ takes the value 500, 5k and 50k in order to ensure that each instance of Event$_a$ will find a matching presence of Event$_b$ in the other window (and vice versa).

Δ Window size	**Dalvik %**	**ART %**
500-5k	-8.1 ± 11.8	-9.9 ± 11.1
5k-50k	-8.3 ± 17.6	-9.5 ± 11.5

Table 8.3-xii Change in throughput for fixed window joins

Table 8.3-xii presents the decrease in throughput between each increase in window size and renders a stable and low drop for both environments. Dalvik seems unprecedented by the issues that arose on windows of size 50k in measurements presented table joins in chapter 8.3.5, and enforces the belif that continuous clean up of data items that leave the bounds of the sliding window, aids in the overall stability of the operating environment.

Window size	***Δda* %**
500	27.1 ± 10.1
5k	24.6 ± 12.7
50k	22.9 ± 16.3

Table 8.3-xiii Difference between Dalvik and ART for fixed window joins

Table 8.3-xiii presents the difference between Dalvik and ART for each window size and renders that the actual distinction between both engines becomes lower when tasks include a combination of projection, comparisons and windows. Effectively representing a different truth than the tasks who exercise a primitive factor in isolation.

8.3.6.2 B. Fixed Window Size and Variable Selectivity

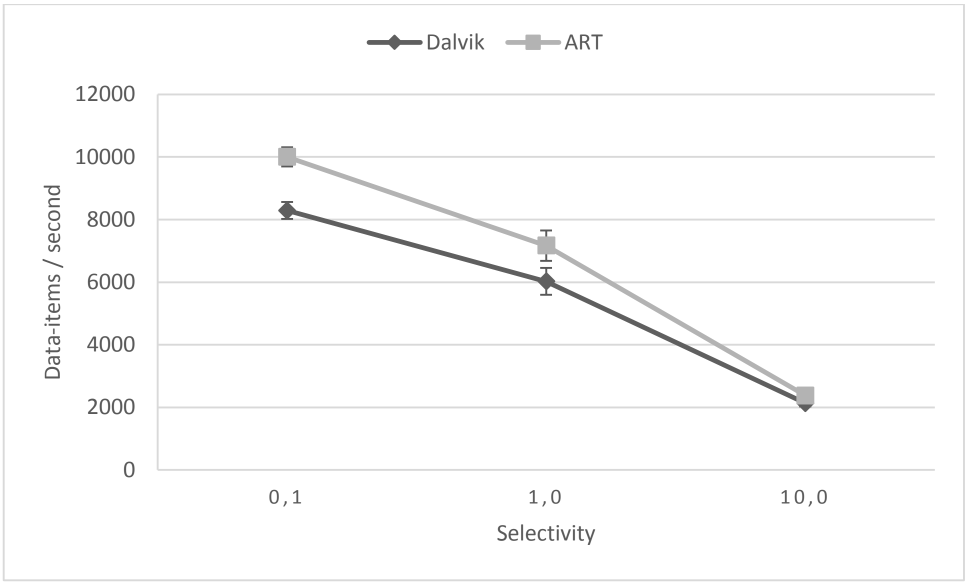


Figure 8.3.6-ii Measurements concerning window to window joins with variable selectivity

Figure 8.3.6-ii represents Query 7.1.5-i over a sliding window of size 5k. The identifier range I_n for $Event_a$ and $Event_b$ takes the values 50k, 5k and 500 in order to ensure that each instance will find 0.1, 1 and 10 presences in the other window.

Δ Selectivity	**Dalvik %**	**ART %**
0.1 – 1.0	-27.3 ± 10.3	-28.3 ± 9.8
1.0 – 10.0	-64.6 ± 11.8	-66.8 ± 12.9

Table 8.3-xiv Change in throughput for variable window joins

Table 8.3-xiv presents a profound drop in throughput when the cardinality of matches increase. We know that Esper / Asper is forced to construct previously unseen information and implies that listeners will retrieve an array of information that represents each match. It becomes apparent that this operation is profoundly costly regardless of runtime environment. The cardinality of garbage collection procedures increases by approximately 12% for Dalvik and 5% for ART, and indicate that this might not be the deciding factor for why throughput is so severely decreased. Previous tasks fail to exercise this factor as they only produce a single answer and this effect is again seen in chapter 8.3.7.3, where patterns are enforced to output every event that has previously been seen.

Selectivity	***Δda* %**
0.1	20.6 ± 6.3
1.0	18.9 ± 13.8
10.0	11.5 ± 10.9

Table 8.3-xv Difference between Dalvik and ART for variable window joins

Table 8.3-xv presents the difference between Dalvik and ART for each increase in selectivity and indicates that the decrease is closely related to the engine, rather than the operating environment.

8.3.7 Task 6: Patterns

This section presents three distinct tests that each exercise various aspects of pattern detection. The first test is concerned about temporal intervals, while the second test exercises the amount of unique attributes that the pattern engine must keep notion of, and the third test exercises selectivity.

8.3.7.1 Time

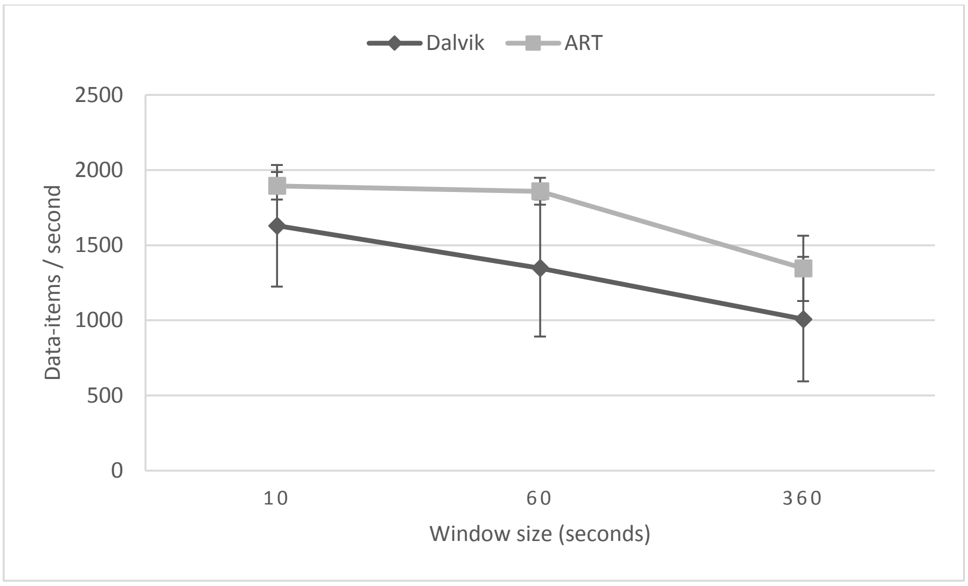


Figure 8.3.7-i Change in throughput for temporal patterns

Figure 8.3.7-i presents Query 7.1.6-i over a temporal context that lasts for 10, 60 and 360 seconds. The warmup procedure retains equal length for the context that lasts 360 seconds. Selectivity is fixed at 1% and the identifier cardinality is set to 1k. The high deviations imposed by Dalvik are from our perspective, not exclusive to patterns. Standard deviation is stable at approximately 400 data items per second and relate to deviations seen in task 1 and 5. Lowering the monitoring granularity to 100 milliseconds present no distinct processing pauses.

Δ Window size (seconds)	**Dalvik %**	**ART %**
10-60	-17.3 ± 58.6	-1.8 ± 9.6
60-360	-25.1 ± 75	-27.5 ± 20.9

Table 8.3-xvi Change in throughput for temporal patterns

Table 8.3-xvi presents the decrease in throughput for each increase in window length. ART retains stability when the temporal context increases from 10 to 60 seconds as the cardinality of garbage collection procedures is, on average, unchanged.

Window size (seconds)	***Δda* %**
10	16.2 ± 29.7
60	38 ± 38.6
360	33.5 ± 57.2

Table 8.3-xvii Difference between Dalvik and ART for temporal patterns

Table 8.3-xvii renders that the difference between Dalvik and ART is faded because of the high deviations caused by Dalvik. The main distinction between both environments lies in the garbage collector and enforces stability for ART.

8.3.7.2 Identifier Range

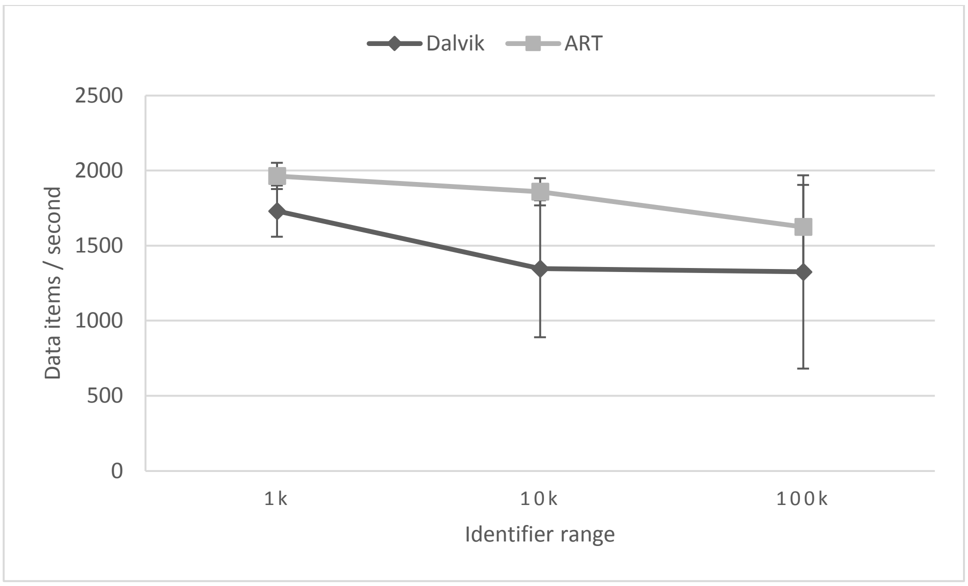


Figure 8.3.7-ii Change in throughput for variable identifier ranges

Figure 8.3.7-ii presents Query 7.1.6-i over a temporal context that lasts 60 seconds and retains a fixed selectivity of 1%. The range of unique event identifiers takes is set between 1k and 100k, and imposes that the pattern engine must accommodate to retaining an increased amount of different states that continuously expire. 1% selectivity cannot be assured for a range of 100k as it would impose pre-allocation of more than 100k data-items to ensure that the appropriate random numbers are drawn. Thereby imposing a distinct setup that cannot compare to the other measurements.

Δ Identifier range	**Dalvik %**	**ART %**
1k-10k	-22.1 ± 43.7	-5.3 ± 9.3
10k-100k	-1.6 ± 82.4	-12.5 ± 22

Table 8.3-xviii Change in throughput for variable identifier ranges

Table 8.3-xviii presents the decrease in throughput for each increase in identifier range. The decrease between 1k and 100k is surprisingly low given the amount of state that the engine must retain, but imposes profound garbage collection procedures that blocks processing and introduces spikes of executional pauses. ART experiences a stable decrease in throughput and increase in deviation that we have not explored further.

Identifier range	***Δ da* %**
1k	13.5 ± 14.3
10k	38 ± 38.6
100k	22.6 ± 65.8

Table 8.3-xix Difference between Dalvik and ART for variable identifier ranges

A clear distinction between Dalvik and ART is again faded because of the high deviations but imposes that the main distinction is operational stability over time.

8.3.7.3 Selectivity

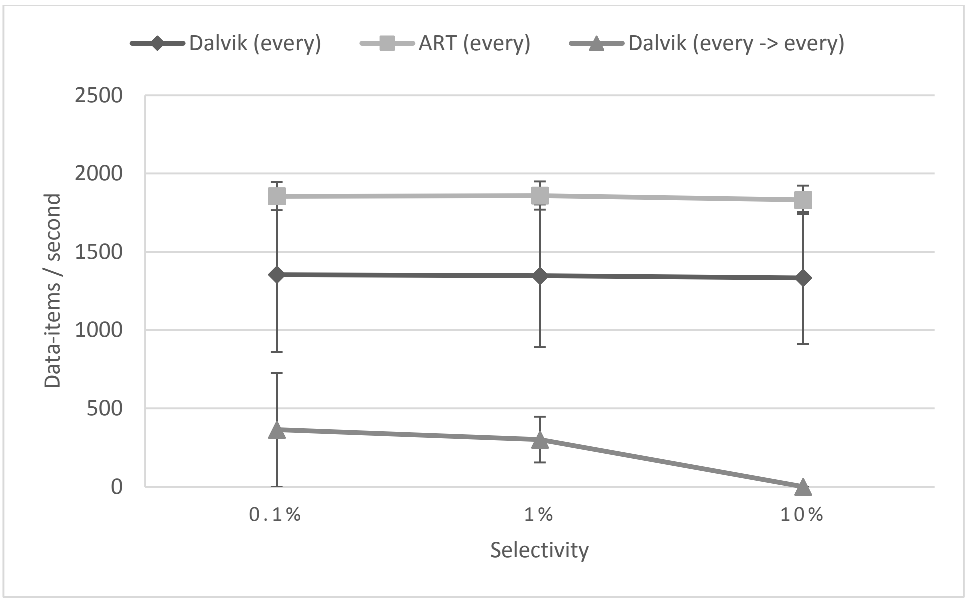


Figure 8.3.7-iii Change in throughput for pattern selectivity

Figure 8.3.7-iii presents Query 7.1.6-i over a temporal context that lasts 60 seconds and retains a variable selectivity from 0.1, to 10%. The range of unique event identifiers takes is set to 1k and the main intention of this task is to render the distinction between the use of the *every Event$_a$ -> Event$_b$* and *every Event$_a$ -> every Event$_b$*, where the latter places pressure on the cardinality and evaluation policies of a pattern.

The utilization of a single *every* operator presents little or no distinction between the measurements as the low selectivity rate enforces little or no output projection. We experience however, a different story when utilizing the *every -> every* operator as both Dalvik and ART suffers profoundly from the increase in evaluation and retainment of previously seen pattern matches. ART was under no circumstance able to finish a single test sequence with this setup, as memory allocations accumulate to a degree that forces the application to terminate, and Dalvik experiences the same problem with 10% selectivity.

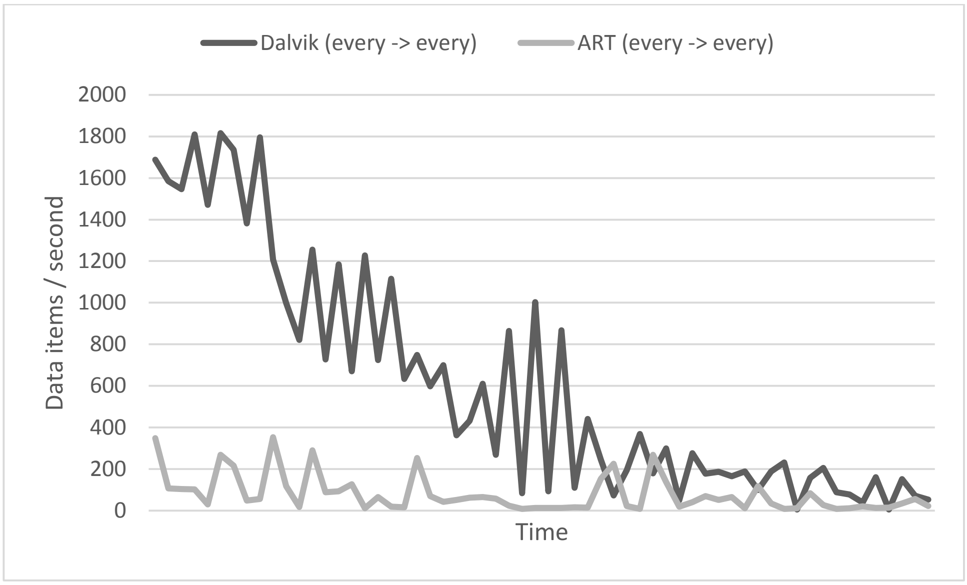


Figure 8.3.7-iv Difference between Dalvik and Art for repetitive evaluation policies

Figure 8.3.7-iv presents a single measurement that utilizes the *every->every* operator over a 10 second temporal context at 0.1 %, where no warmup time is present and the runtime is set to 60 seconds. Distinction between lower identifier ranges prove no distinct difference and imply that output projection could be the deciding factor that renders Dalvik and especially ART, nearly useless in this scenario. ART will, almost immediately suffer from continuous garbage collection procedures that do not stop long after the test is ended. We believe that this could be caused by an operational fault in the design of the garbage collector as we fear that sizable arrays of data items are stored in a separate heap intended for large objects. If the cleanup policies for this heap are of such nature that they are conducted on low intervals, then it could explain the terminations that we experience. Dalvik retains operational stability as it constantly performs GC_FOR_ALLOC operations and ensures that the heap is maintained. Attempts to utilize a logical pattern expiration policy rather than a temporal, rendered no difference for this matter.

8.3.8 Task 7: Database

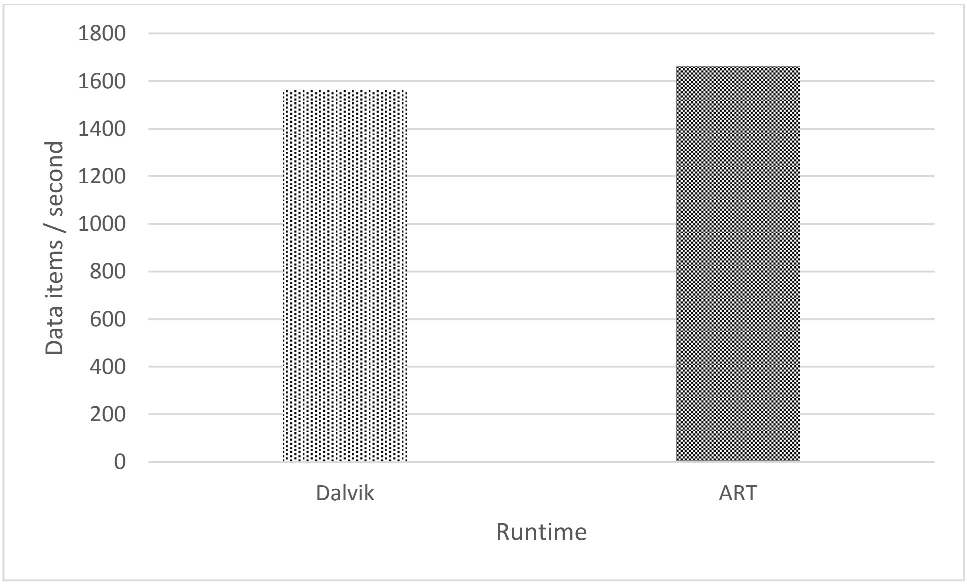


Figure 8.3.8-i Measurements concerning static database access

Figure 8.3.8-i presents Query 7.1.7-i, where each en every data item that enters the stream is joined and enriched with information from a prepopulated SQLite database that contains 10k entries. It is, without doubt presented that higher throughput rates are unachievable as the main bottleneck lies in between the layer of communication between Asper and the database [30] . We know that Asper will allocate a thread pool that establishes a stable connection to the database and perform query executions upon request[29]. Resizing this thread pool to include 10, 20 and 30 threads yielded in no difference with respect to throughput. We started to question our implementation of the driver that enables Asper to communicate with SQLite, and performed therefore a simple experiment where we disabled the driver and performed the same joins through a custom function[30] that utilized Androids embedded API for database communication. This function would simply retrieve a numeric index of the primary key that we wanted to locate, and returned a key-value map of the projected table columns that the database returned. This experiment yielded the same results as seen in Figure 8.3.8-i, and indicate, to some extend, that our driver implementation is as efficient as the one embedded in Android.

***Δda* %**
6.4 ± 10.5

Table 8.3-xx Difference between Dalvik and ART for static database access

Table 8.3-xx presents the difference in throughput between Dalvik and ART, and render that there is little or no distinction between the runtime environments. Higher throughputs can be achieved with result caching, but will impose tradeoffs between increased memory consumption and real time requirements.

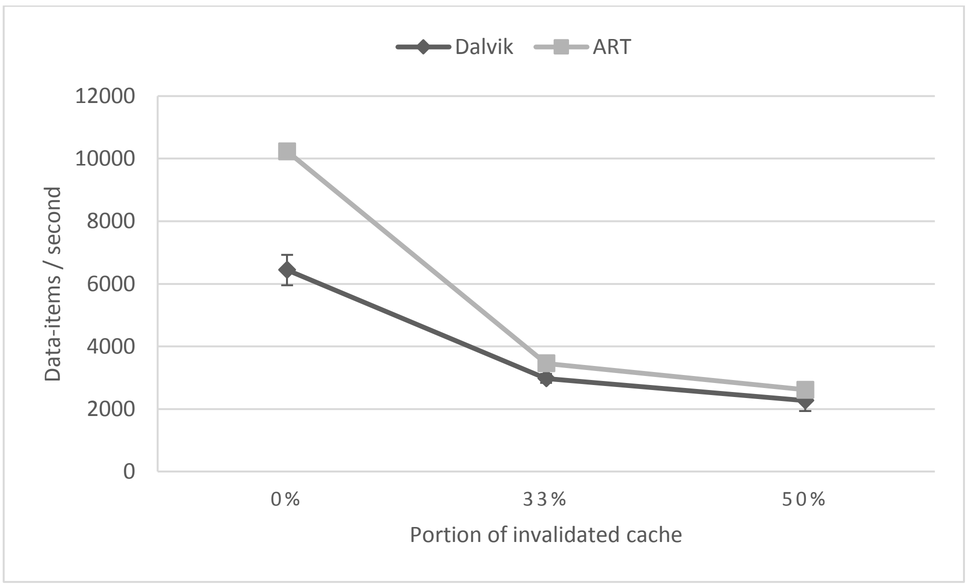


Figure 8.3.8-ii Measurements concerning cached database access

Figure 8.3.8-ii presents the same experiment as above, with the distinction of including an LRU result cache (see Chapter 3.6). The cache retains a fixed size of 1k and cache miss probability is imposed by alternating the event identifier range. An identifier range of 1 - 1k, 1.5k and 2k renders that the result cache will, on average, experience that 0, 33 and 50% of its repository is invalidated.

[29] See [19] , section 15.4.9.1

[30] See [19], section 9.3

Δ Invalidated portion %	Dalvik %	ART %
0 – 33	-53.7 ± 12.2	-66.2 ± 5.2
33 – 50	-23.7 ± 19.2	-24.4 ± 8.6

Table 8.3-xxi Change in throughput for invalidated cache portions

Table 8.3-xxi presents the difference between each increase of portion invalidation and implies that the cost of invalidation can be high, but still profoundly better than without the use of any caching.

Δ Invalidated portion / no cache %	Dalvik %	ART %
0 / 100	312.8 ± 14.7	516.4 ± 4.5
33 / 100	90.8 ± 11.8	108.1 ± 7.4
50 / 100	45.5 ± 21.7	57.3 ± 7.9

Table 8.3-xxii Change in throughput between cached and un-cached access

Table 8.3-xxii presents the increase in throughput between the use of a partly invalidated cached and the use of no cache. A cache with no invalidated portions operates with a throughput that is nearly on pair with regular window to window joins. Attempting to increase the portion of invalidated cache to 100 and 200 % yield in little or no difference when comparing it to an un-cached join and implies that lookup procedures are cheap as they index on query parameters.

8.3.9 Task 8: Multiple Events

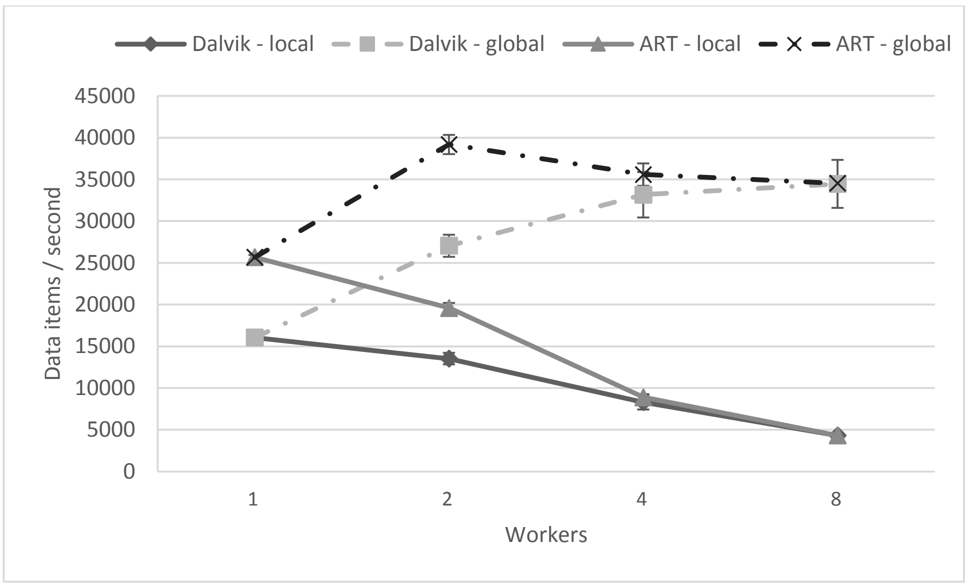


Figure 8.3.9-i Change in throughput for multiple Worker instances

Figure 8.3.9-i presents Task 8, where 1-8 distinct Workers operate on 1-8 distinct queries and event types. The range of unique and random event identifiers takes an initial value of 1-100k, and is then divided by the amount of Workers that currently operate. Selectivity predicate *K* is lowered by half for every increase in Workers and ensures that the engine only outputs 25% of the events seen across all installed queries. Each Worker retains only a single event type in its repository.

Local throughput presents the average throughput achieved by each separate Worker, while the *global* throughput is denoted as the average sum of data items processed by all Worker instances during the test duration.

Δ Workers / type	Dalvik %	ART %
1-2 / local	-15.7 ± 7.8	-23.6 ± 4.2

2-4 / local	-38.6 ± 15.2	-54.6 ± 7.5
4-8 / local	-48 ± 18.9	-51.4 ± 7.3
1-2 / global	68.6 ± 7.6	52.6 ± 4
2-4 / global	22.6 ± 13.1	-9.2 ± 6.7
4-8 / global	3.8 ± 16.5	-2.9 ± 5.5

Table 8.3-xxiii Change in throughput for multiple Workers

Table 8.3-xxiii presents the difference between each increase of Workers both locally and globally. We initially hoped that the average local throughput between 1-4 Workers would present lower drops in performance as the queries share no state or references between them. Attempting to set explicit context partitions[31] for each separate query render no difference from our state of view, and indicate that locking could occur on the input or output channels. Attempting to profile[32] the benchmark application yield in no direct conclusion, as we cannot see any distinct increase in the use of locks. The profoundly high amount of method calls that Asper imposes for its operations makes it difficult to identify distinct segments.

The higher global throughput rates render that the Workers are able to process more work, but it should be noted that it is only exclusive to distinct queries and events. Attempting to utilize multiple Workers for fewer queries and events will present an unchanged, or lower global throughput as the engine must retain correctness and lock segments to avoid issues that relate to concurrency.

Workers	***Δda* %**
1	59.9 ± 3.8
2	44.9 ± 8.2
4	7.2 ± 14.5
8	0.2 ± 11.5

Table 8.3-xxiv Difference between Dalvik and ART for multiple Workers

Table 8.3-xxiv Presents the difference between Dalvik and ART. We cannot find any references that state if ART embeds a differentiated threading or scheduling model, but some indications on Figure 8.3.9-i prove that this could be true.

8.4 Discussion

Utilizing Esper on an Android based smartphone device proves to be a feasible solution. A direct comparison between these measurements and the results presented by Mendes et.al [30] renders that this solution achieves approximately 1/5 of the throughput opposed to a powerful workstation machine. This decrease is expected as ARM processors embed reduced instruction sets and architectures that favor size, thermal abilities and energy consumption over processing power.

The main intention of this benchmark is to identify the limitations of Asper in order to verify whether its presence on a mobile device is of such level that it could prove feasible to use, thereby acting as a guideline. Measurements are primitive and lack the notion of multiple queries that correlate and depend on each other. It should not be unexpected that actual performance is in realistic scenarios, comfortably lower. A true system would likely have to allocate distinct threads that handle incoming data items from multiple disparate channels. Concurrent access to data repositories, additional locking and waiting and additional garbage collection procedures imposed by the system could all render a different truth about the actual throughput of the engine. Our enablement of Esper has

[31] See [19], section 4.1
[32] With utilization of TraceView [41]. Method based profiling with 1000 microsecond resolution.

never before been present in the environment that we tested. Starting at lower levels and measuring primitive aspects of the engine proved necessary as we identified executional errors that we imposed by our enablement in Chapter 6.

We can without doubt state that the presence of ART enables Esper to operate with increasingly higher throughput and stability, and future revisions of Android will be better suited for operations that require real time constrains. We expect that the official release of ART will be embedded as the default runtime environment on newer smartphone devices during the third or fourth quarter this year. Statements from [25] indicate that ART will embed new and optimized memory allocation procedures (Rosalloc) that enable concurrent threads to operate with less distraction. We question the presence of JIT-optimization in context of Dalvik as even primitive filtering operations yielded profoundly lower results than what we see for ART. We believe however, that the main distinction comes from the improved garbage collection algorithms that ART embed. Heap and runtime environment settings could be tweaked for each distinct device in order to enable Asper to operate optimally for certain scenarios. Asper could be better suited with settings that enable the system to allocate up to 1024 MB of heap space and impose low and frequent collection procedures. Our own experimentation with these settings rendered this concept achievable, but garbage collection must eventually happen. Executional pauses are for Dalvik identified as a major bottleneck as our observations indicate that the garbage collector is unable to annotate and discard collections without imposing executional pauses that will affect all threads within the process. It imposes itself as a dangerous concept because small buffers could overflow upon pauses that last for more than 3-400 milliseconds.

The combination of temporal or logical contexts and output generation appears, from the perspective of task 5 and task 6 to be of profound cost as the environment is forced to allocate, project, discard and reflect over the data that arrives. Both tasks could be rendered distinctively important as they perform operations that we actually expect to locate in true systems. Projecting large collections of data proves to be poorly handled by both runtime environments as we see in Task 2 and Task 6. Esper and Asper is designed to produce a distinct data set that in our measurements is presented as an "EventBean"-object that contains a key value set of each projected attribute. The cost of producing this object could be omitted if queries utilize the "*" operator to reference the underlying data item. We also investigated the use of a Subscriber[33] instead of a Listener, where the Subscriber retains a fixed method signature that retrieves the outputted events as references. We found however that throughput was lowered as invocation of these method signatures are performed by our implementation of CGLib and involves the use of reflection every time the engine outputs information.

Caches will, in context of database communication impose improved overall performance, but must impose timeout procedures to ensure that inconsistent information is not presented to the query that performs the join. The most important aspect of database communication is not necessarily the throughput, but rather the ability of embedding persistent, historic information of occurrences that prove of no value to retain in memory.

We state that it is misfortunate to see that the individual performance of each Worker in task 8 was to such degree lowered as the number of concurrent queries grew. High priority information that requires real time constrains could be subjected to the presence of other threads and queries. It could however prove sensible to a design a system that pre-allocates pool of at most two threads

[33] See [19], section 12.3.2

that process any event that enters the application domain. Care must however be taken in order to prevent that thermal constrains take effect.

The rate and cardinality of data that arrive from one or more sensors will likely not be as profound as the setup that we present, and render that Asper is presents itself as a feasible system for processing of information within the domain of pervasive healthcare.

9 Conclusions

Esper was originally developed with the intention of processing intensive amounts of financial information, but its design and size indicates that it could be feasible to utilize it as a central aggregator in context of medical monitoring and pervasive healthcare. We cannot directly imply that it would be sensible to utilize Asper for the purpose of transforming and aggregating high rates of raw data streams. Such transformations should be conducted on the sensor itself and Asper should only retain a notion about the higher-level event that occurs. The introduction of ART resolves many performance related issues that was previously seen with Dalvik [9], and proves that future revisions of Android could be suitable for this enablement.

The true power of this enablement comes from the fact that CEP assisted data stream processing becomes ubiquitous. Effectively abstracting away complexities of data stream management by embedding expressive queries instead of application specific code. We envision that Asper could be implemented as a central service that handles streams from multiple disparate sensors and information sources. Android embeds a set of primitive building blocks that could be formed into a truly heterogeneous and scalable system, where a single instance of Asper retains notion of every event that occurs in the externally observed world.

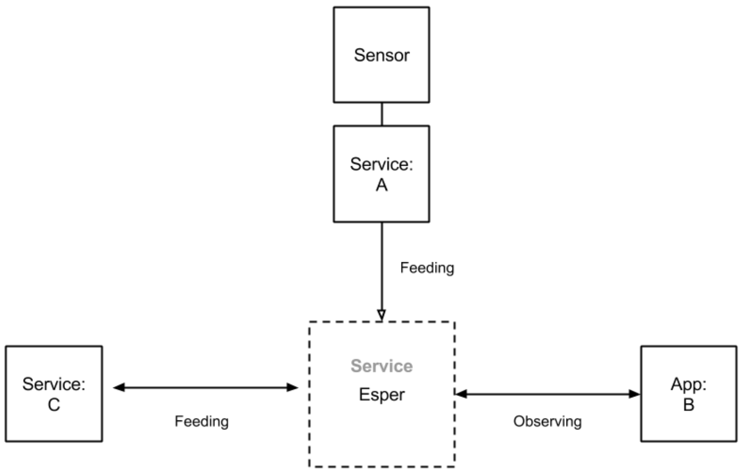


Figure 9-8.3.9-i Placement of Asper in an Android based system

Communication could be formed through the use of IPC-messages and simple API endpoints that enable any service to register event types and queries of interest. Listeners could broadcast event occurrences or route projected events back to subscribing services by the sole use of existing framework components. Some sense of state could be retained as Esper enables declaration and updating of variables through one time queries. Moving installed queries and variables between a static aggregator at home and to Asper could prove possible through the use of *Modules*[34] that contain summaries of the current engine configuration. We state that an enablement of Esper on Android is sensible, as it could prove useful for other purposes than pervasive healthcare. Two distinct organizations placed interest in Asper during our implementation and evaluation cycles and expressed scenarios where Asper could be utilized to offload central servers by moving logic and

[34] See [19], section 12.2

processing into the smartphone domain as communication is seen as a profound cost in terms of energy consumption.

9.1 Summary of Contribution

Our contributions can be summarized in the following note:

- We identified the main factors that prevented an enablement of Esper on Android true. Each factor was accommodated and resolved to ensure that Esper would retain functionality without the need of profound alternation of the existing source code. We thereby state that this groundwork will ease future implementations of Esper, and enable continuity and use of what we believe, is the only fully functional CEP engine that can be utilized on smartphone devices.

- We located, compiled and verified a specific JDBC driver that enabled Esper to communicate with the embedded relational database on Android. Thereby effectively presenting a standardize and maintainable mean of performing structured joins between persisted, historic information that we believe is important for an enablement of contextual, pervasive healthcare monitoring.

- Our enablement of Esper was benchmarked with a set of primitive tasks that covered fundamental principles of both DSM and CEP to present a general guideline for this system and to identify possible implications that could render this enablement problematic. The benchmark application retains a generic structure that could prove useful for others that want to evaluate Asper for tasks that this thesis did not cover.

- Both Dalvik and ART was individually studied and where we uncovered differences that could render useful outside the scope of this thesis and relate directly to processing intensive amount of data on an Android based smartphone device. ART is, from our knowledge at this point still sparsely documented and external measurements that concerns this runtime environment is not present.

9.2 Critical Review

Initial measurements that concerned Asper where conducted wrongfully as we believed that Esper would, by itself clone any data items that entered named window in order to maintain immutability. This however, proved false after a discussion with an EsperTech representative and existing measurements had to be discarded as the garbage collection procedures were unaffected by the fact that no data items were ever reclaimed. This thesis contained additional tasks that exercised the use of multiple Workers under scenarios where joins between multiple windows were conducted concurrently, but was not included due to the lack of time imposed by measurement repetitions. The exclusion of garbage collection procedures should have been identified in earlier stages as a simple observation would have indicated that something was wrong. We also believe that our focus should have been retained on complex tasks that utilize several principles form both CEP and DSM concurrently. The value of a simple filtration or projection could be stated limited.

Information about the internal workings of ART were not available until the third quarter of 2014 [25]. We misinterpreted how the internal garbage collector operated and relied to a large extend on literature that was present for Java and not Android. We state that literature concerning Dalvik and ART is limited, but this should not have prevented us from conducting the experiments from chapter 8.2.2 earlier, as they would have given us much needed information about the distinction between ART and Dalvik.

9.3 Future Work

We believe that our implementation of CGLib could be performed differently by the use of Dexmaker [40] to perform dynamic code generation and to disable the use of reflection. Our investigations on this matter where at the time of this thesis sparse and it could be stated that this would be an optimal solution.

The benchmark application that we present should include the possibility of retaining data items from external sources in order to render how the inclusion of communication and unmarshaling of information can effect the overall performance of the engine. The benchmark application itself should support the use of predefined arrays of data that could render it useful for scenarios that require queries to reason about the content that reside in the data items.

Bibliography

[1] "Novelda. UWB-IR overview." [Online]. Available: https://www.xethru.com/content/technology-0. [Accessed: 01-Aug-2014].

[2] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *Computer (Long. Beach. Calif).*, vol. 38, no. 11, pp. 23–31, Nov. 2005.

[3] H. C. Powell, A. T. Barth, K. Ringgenberg, and B. H. Calhoun…, "Body area sensor networks: Challenges and opportunities," *Computer (Long. Beach. Calif).*, 2009.

[4] U. Varshney, *Pervasive healthcare computing: EMR/EHR, wireless and health monitoring*. 2009, pp. 1–282.

[5] S. Støa, M. Lindeberg, and V. Goebel, "Online analysis of myocardial ischemia from medical sensor data streams with Esper," *Appl. Sci. …*, 2008.

[6] J. Soberg, V. Goebel, and T. Plagemann, "Commonsens: Personalisation of complex event processing in automated homecare," in *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2010 Sixth International Conference on*, 2010, pp. 275–280.

[7] S. Hong, R. P. Sahu, M. R. Srikanth, S. Mandal, K.-G. Woo, and I.-P. Park, "Real-Time analysis of ECG data using mobile data stream management system," in *Database Systems for Advanced Applications*, 2012, pp. 224–233.

[8] J. Dunkel, R. Bruns, and S. Stipkovic, "Event-based smartphone sensor processing for ambient assisted living," in *2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS)*, 2013, pp. 1–6.

[9] K. Jaein, K. Nacwoo, Y. Simkwon, and L. Byungtak, "A study on CEP performance in mobile embedded system," in *2012 International Conference on ICT Convergence (ICTC)*, 2012, pp. 49–50.

[10] D. Bade, "Esper-Android / JESPA-Android," 2010. [Online]. Available: http://vsis-www.informatik.uni-hamburg.de/oldServer/teaching//projects/esper-android/. [Accessed: 12-May-2013].

[11] G. Cugola and A. Margara, "Processing flows of information," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–62, Jun. 2012.

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '02*, 2002, p. 1.

[13] D. Luckham, *The power of events*, vol. 204. Addison-Wesley Reading, 2002.

[14] O. Etzion and P. Niblett, *Event Processing in Action*. Manning, 2010, pp. 1–325.

[15] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM Sigmod Rec.*, vol. 32, no. 2, pp. 5–14, 2003.

[16] “Stream and Complex Event Processing. PhD Course: On benchmarking Information Flow Processing Systems,” *Politecnico di Milano*, 2013. [Online]. Available: http://streamreasoning.org/schep-phd-course-2013. [Accessed: 01-Aug-2014].

[17] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[18] “Esper. Open source, project page.” [Online]. Available: http://esper.codehaus.org/. [Accessed: 01-May-2014].

[19] “Esper, v.4.8.0 documentation.” [Online]. Available: http://esper.codehaus.org/esper-4.8.0/doc/reference/en-US/html/index.html. [Accessed: 01-Aug-2014].

[20] D. Luckham, “Rapide: A language and toolset for causal event modelling of distributed system architectures,” in *Worldwide Computing and Its Applications — WWCA’98 SE - 8*, vol. 1368, Y. Masunaga, T. Katayama, and M. Tsukamoto, Eds. Springer Berlin Heidelberg, 1998, pp. 88–96.

[21] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing*. 2013.

[22] “Apache Harmony Project. Open source, Java distribution.,” 2010. [Online]. Available: http://harmony.apache.org/.

[23] D. Bornstein, “Google I/O 2008 - Dalvik Virtual Machine Internals,” 2008. [Online]. Available: https://www.youtube.com/watch?v=ptjedOZEXPM. [Accessed: 20-Oct-2013].

[24] B. Cheng and B. Buzbee, “Google I/O 2010 - A JIT Compiler for Android’s Dalvik VM,” 2010. [Online]. Available: https://www.youtube.com/watch?v=Ls0tM-c4Vfo. [Accessed: 10-Apr-2014].

[25] A. Ghuloum, B. Carlstrom, and I. Rogers, “Google I/O 2014 - The ART runtime,” 2014. [Online]. Available: https://www.youtube.com/watch?v=EBlTzQsUoOw. [Accessed: 26-Jul-2014].

[26] P. Dubroy, “Google I/O 2011: Memory management for Android Apps.” [Online]. Available: https://www.youtube.com/watch?v=_CruQY55HOk.

[27] R. Meier, *Professional Android Application Development*. 2012, p. 432.

[28] “Discussion thread. Implications that regard porting of Esper to Android.” [Online]. Available: http://esper.13850.n7.nabble.com/Esper-on-Android-td8669.html. [Accessed: 10-Oct-2013].

[29] R. Friedman, A. Kogan, and Y. Krivolapov, “On Power and Throughput Tradeoffs of WiFi and Bluetooth in Smartphones,” *IEEE Trans. Mob. Comput.*, vol. 12, no. 7, pp. 1363–1376, Jul. 2013.

[30] M. R. N. Mendes, P. Bizarro, and P. Marques, “A performance study of event processing systems,” in *Performance Evaluation and Benchmarking*, Springer, 2009, pp. 221–236.

[31] M. R. N. Mendes, P. Bizarro, and P. Marques, “A framework for performance evaluation of complex event processing systems,” *Proc. Second Int. Conf. Distrib. eventbased Syst. DEBS 08*, p. 313, 2008.

[32] M. Mendes, P. Bizarro, and P. Marques, "FINCoS: benchmark tools for event processing systems," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, 2013, pp. 431–432.

[33] "Dalvik Wiki. Inclusion of Javax packages." [Online]. Available: https://code.google.com/p/dalvik/wiki/JavaxPackages.

[34] "OpenJDK - JDK6 Project page." [Online]. Available: http://openjdk.java.net/projects/jdk6/. [Accessed: 17-Apr-2014].

[35] "SLF4J - Android." [Online]. Available: http://www.slf4j.org/android/. [Accessed: 01-Aug-2014].

[36] "Xerces for Android." [Online]. Available: http://code.google.com/p/xerces-for-android/. [Accessed: 01-Aug-2014].

[37] "SQLDroid. SQLite JDBC driver for Android." [Online]. Available: https://github.com/SQLDroid/SQLDroid. [Accessed: 05-Jul-2014].

[38] "Oracle Berkeley Database." [Online]. Available: http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/. [Accessed: 01-Aug-2014].

[39] "Building the Android JDBC Driver / Building Berkeley DB for Android." [Online]. Available: http://docs.oracle.com/cd/E17076_02/html/installation/build_android_jdbc.html. [Accessed: 01-Aug-2014].

[40] "Dexmaker. Programatic code generation for Android." [Online]. Available: https://code.google.com/p/dexmaker/. [Accessed: 10-Aug-2014].

[41] "Android. Profiling with Traceview and dmtracedump." [Online]. Available: http://developer.android.com/tools/debugging/debugging-tracing.html. [Accessed: 01-Aug-2014].

Appendices

Appendix A

The intention of this section is to describe and discuss the problems that arose and prevented us from including the UWB-IR presented in the preface of this thesis. We believed that locating the maximum, achievable throughput renders only an image of fictive workloads and isolated measurements that miscorrelate with our perception of ubiquitous monitoring. A realistic scenario would render how well Asper works over time and project if the overhead of utilizing Asper is of such kind that it would deplete the smartphone device of its resources.

A working prototype was constructed and intended to present a proof-of-concept. The intention of this prototype was to ensure that the surrounding application logic was as minimal, only embedding a simple producer/consumer model, and that Asper should handle all processing and classification by utilizing principles from the domain of DSM and CEP. A separate requirements analysis was conducted where we identified that we under no circumstances intended to develop a fully featured, heart-rate and respiration monitoring system as this would require expertise in a domain that outranges the purpose of this thesis. We could however, place a set of factors, presented below, under consideration and effectively render a baseline cost of operation.

Factor	**Description**
Processor	How much toll is placed, on average on the processor? If a high percentage of processor utilization is demanded – then it will impose implications for other applications and services that reside on the device.
Energy consumption	Information about processor utilization is of little value if presented without a notion how long such computations can prevail. Pervasive vital-sign monitoring is of limited value if consumption levels are of such kind that the device will cease of function after a quarter day.
Response time	Delay between the presence of a vital sign and Aspers detection of it could be caused by a combination of the runtime environment and Asper itself. Any delays would prove useful to identify as real-time vital-sign monitoring requires a notion of timeliness.

Sampling of data was conducted by monitoring standing, young male with on-body antennas that were placed on bare skin towards solar plexus in order to render a clearer separation between the heart and the lungs.

A single, 60 second sample includes a resolution of approximately 150 data items per second and results in several million individual data items that must be filtered out as they include information and noise that is useless in our context. Filtration and cleaning operations were conducted offline, by utilizing a simple, 5-point Triangular Weighted Smooth algorithm that conducted 50 iterations on the sample in order to present a stream of data that Asper could process without wrongfully classifying noise as actual vital signs.

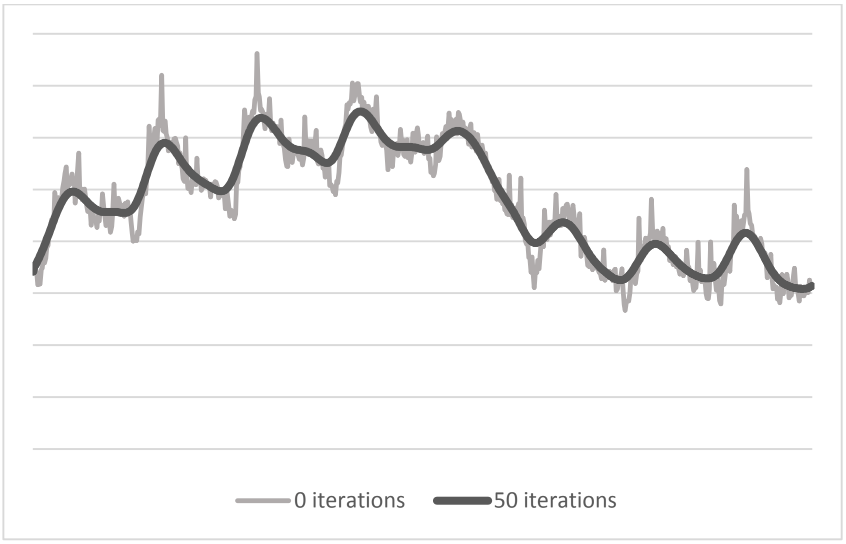


Figure A i

Figure A i presents motions that regard ventricular filling and ejection of the heart. Similar motions are detected from the lungs and represent inhalation and exhalation patterns. Asper itself should be responsible for reasoning about the stream of information coming from the sample. This implies that installed queries should perform selection, transformation and identification of heartbeats by detecting changes in motion. The point to point intervals of this mechanism is aggregated into a higher level representation of the heartbeat, and any lower level data-items that represented this instance, are discarded as high accumulation of objects in the heap could case unnecessary executional pauses. A higher level representation of the heartbeats would render the average *Beats Per Minute (BPM)* interval by embedding a set of aggregation functions. Respiration detection can be detected as a separate stream and reasoned upon in isolation. Identification of inhalation and exhalation and classification of a respiration cycle will render possible breathing patterns that could be utilized for a set of contextual purposes.

All operations are performed in-memory, and the transformed sample is identified as a *Radarsignal* event that embeds the following schema:

Timestamp	**Heart**	**Respiration**
Unix timestamp representing the occurrence of this sample	Decimal / Double	Decimal / Double

The sample itself is manually annotated with special values that highlights zero derivatives and identifies turning points for each heartbeat (value: 1001) and respiration (value: 1002) cycle. The intention of this was to identify, extract and timestamp these values in order to classify their occurrence and directly compare the applications notion of a cycle, and Aspers notion of a cycle.

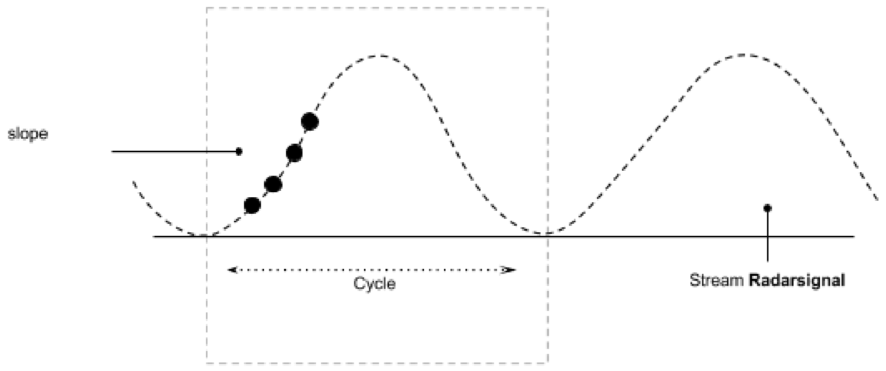


Figure A ii

Figure A ii presents how the raw sample from Figure A ii is interpreted by Query A i. The main intention of this query is to identify the current slope of the stream by utilizing a jumping window of size *n* that denotes how many data-items we require in order to properly classify the slope without introducing false positives. We know that a single, sampled heartbeat occurs approximately every 0.6-0.7 seconds and *n* can for this sample be set to ± 25 to effectively evaluate the slope 6 times per second. *n* will however vary if the heart rate increases, and would for such purposes require a higher classification interval of the slope [35]. A respiration cycle occurs, on average every 4-6 seconds, and implies that we can set *n* to 100 as re-evaluation of the stream is not needed as often.

```
INSERT INTO Heartstream
SELECT slope
FROM Radarsignal.win:length_batch(n).stat:linest  (signal,  time)
```

Query A i

Query A i utilizes an embedded, statistical function from Esper / Asper and takes two parameters. *Signal* denotes either a single heart or respiration sample that represents a fictional y-axis, and *time* denotes the explicit timestamp that denotes a fictional x-axis. The output of Query A i is present to Query A ii, which sole purpose is to look for a simple pattern that identifies whether a cycle has occurred by classifying the occurrence of positive and negative slopes.

```
INSERT INTO Heartbeats
SELECT c.timestamp as occurrence, (c.timestamp - a.timestamp) as duration
FROM PATTERN
[
        every
        a = Heart(slope > 0)
        → b = Heart(slope < 0)
        → c = Heart(slope > 0)
]
```

Query A ii

[35] Could utilize explicit query variables to dynamically re-calculate *n* as the heart or respiration rate increases or decreases.

Query A ii presents how heart cycles can be classified. The extraction of timestamps from the underlying stream allows us to project a higher-level event that includes information about when ventricular filling occurred (denoted by *a*), when it stopped (denoted by *b*) and when ventricular ejection completed (denoted by *c*). This event is then outputted into a separate stream named Heartbeats (presented by Figure A iii), and allows us to discard any remaining, lower level information and the same procedure applies for respiration cycles.

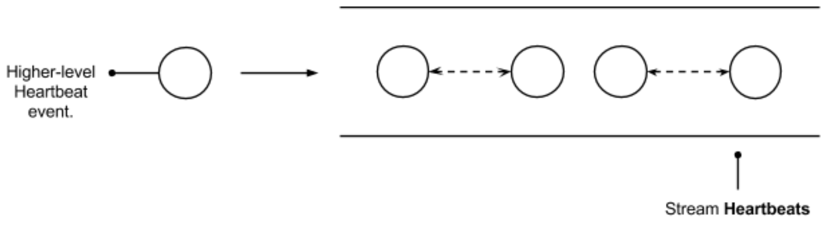


Figure A iii

Classification of the Beat-Per-Minute (BPM) and whether inhalation or exhalation is occurring can now simply be performed on the higher level events as presented in Query A iii.

```
# Classification of BPM
SELECT (count(*) * (60 / n)) as bpm
FROM Heartbeats.win:time(n seconds)
OUTPUT LAST EVERY n seconds
```

```
# Classification of inhalation or exhalation
SELECT slope
FROM Respirationstream
OUTPUT SNAPSHOT EVERY n seconds
```

Query A iii

We managed to verify that our queries performed the correct classifications, but measurements that concerned processor and energy consumption were in no order presentable. Profiling of this application was conducted by utilizing Trepn[36], but the high overhead that this profiling utility imposed, was not distinguishable from the overhead that Asper imposed. Trepn utilized approximately 35-40% of a single processor core when sampling at a rate of 10 samples per second. Our implementation of Asper and this vital sign detection application was running on a separate thread, but utilized the same, single processor core throughout all samples that we conducted. We initially attempted to profile the application when no queries were running, but could not directly compare the average consumption of a non-working application to a working one based on 10 consecutive runs. Attempting to run the Top-command[37] on the smartphone device rendered that our prototype application consumed, on average 10 ± 3 % of available processor time, but we could not locate the same numbers when analyzing profiles that Trepn conducted. This simple exercise rendered doubt in the utility, and we could no longer trust that measurements concerning energy consumption rendered true.

Our idea of calculating the average response time by comparing time stamped annotations from the sample resulted in uncorrelated timestamps that deviated profoundly (± 500 milliseconds). We

[36] A profiling utility from Qualcomm, capable of extracting information about the current processor, memory and energy consumption. See https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler

[37] A Linux specific shell command that presents consumption statistics about distinct processes that reside on the system.

believe that the cause of this deviation was directly related to poor annotation schemes and to the use of un-synchronized timers. Query A i utilizes a jumping window that expires differently for each application run. Attempting to replace the logical window with a temporal window, yielded in the same issues as we believe that the internal notion of time in Asper should have been synchronized by an external clock that both our application, and Asper utilized.

We could not construct a specialized application for this purpose as this thesis was beyond its dead-line, and impose that these measurements are therefore omitted. We are however concerned about the high consumption levels that the Top-command presented, and state that Asper might not be directly usable in context of continuous, mid/high-rate data arrival on an energy constrained device. It does however, not imply that Asper should be utilized as a tool for abstracting complex tasks that would require a profound amount of application logic. All queries in this section present how easy it can be to perform simple tasks without the need of domain specific knowledge about the framework and programming environment that reside on a smartphone device.