# Design and Evaluation of an FPGA-based Query Accelerator for Data Streams

by
Yasin Oge

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering

Graduate School of Information Systems
The University of Electro-Communications
Tokyo, Japan

March 2016

# Design and Evaluation of an FPGA-based Query Accelerator for Data Streams

by
Yasin Oge

Supervisory Committee:

Prof. Tsutomu Yoshinaga, Chair

Prof. Hiroshi Nagaoka

Prof. Hiroyoshi Morita

Prof. Satoshi Kurihara

Prof. Tadashi Ohmori

# 概要

近年，データストリームに対するリアルタイム処理の重要性が高まってきている．特にストリーム処理に特化したデータストリーム管理システム（Data Stream Management System，DSMS）は，理論的に無限のデータストリームに対して SQL ライクな継続的クエリを実行することでリアルタイム処理を実現する．このような背景の中で，増加し続けるデータ量に対してリアルタイムなレスポンスを提供する手段として FPGA（field-programmable gate array）に代表されるプログラマブル・デバイスをクエリ処理専用のアクセラレータとして活用する取り組みが注目されている．本論文は，データストリーム処理におけるウィンドウ集約クエリの高速化を目的として，FPGA を用いたクエリ・アクセラレータの設計・評価に関する研究について記述したものである．

本論文では，まず関連研究の説明を通して本研究の立ち位置を明確化した後，３つの研究課題とそれぞれの解決方法を示している．1 つ目の研究課題として，従来研究のデータストリーム処理専用ハードウェアでは入力データ（タプル）の順序の乱れへの対応が考慮されていなかった問題が挙げられる．そこで本研究は，順序の乱れた入力タプルを許容する効率的なハードウェア実装方法を提案している．2 つ目の研究課題は，オーバラップするスライディング・ウィンドウの増加に対して性能及び面積の観点からスケールしない問題を解決するハードウェア設計方法を示すことである．そこで，オーバラップするスライディング・ウィンドウをペイン（pane）と呼ばれるサブ・ウィンドウに分割し，アグリゲーション（集約）処理を 2 段階に分けてパイプライン実装する．この際 FPGA 内部のメモリ・ブロック（Block RAM, BRAM）を有効活用することで，従来手法と比較して，性能と面積の両観点において優れたスケーラビリティを実現する．3 つ目の研究課題は，クエリ処理内容の変更に伴うオーバーヘッドによりクエリの実行時（ランタイム時）のコンフィギュレーションが困難な点の解決である．本研究では，この問題を解決する動的再構成可能なクエリ・アクセラレータ「Configurable Query Processing Hardware (CQPH)」のアーキテクチャを提案している．提案アーキテクチャの実現可能性と有効性を示すことを目的として，Xilinx の FPGA 開発ボードを用いて CQPH のプロトタイプを実装・評価した．実機を用いた実験の結果，ギガビット・イーサネットの実効速度で受信したパケットを取りこぼすことなく連続して処理できることを確認した．また，より高速な入力インターフェイスとして DRAM を用いた実機テストでは，10Gbps 以上の入力ストリームに対して，複数のクエリを並列処理できることを示している．

# Abstract

An important and growing class of applications requires to process online data streams on the fly in order to identify emerging trends in a timely manner. Data Stream Management Systems (DSMSs) deal with potentially infinite streams of data that should be processed for real-time applications, executing SQL-like continuous queries over data streams. In order to deliver real-time response for high-volume applications, there is currently a great deal of interest in the potential of using field-programmable gate arrays (FPGAs) as custom accelerators for continuous query processing over data streams.

One of the previous studies focuses on sliding-window aggregate queries and shows how these queries can be implemented on an FPGA. Nevertheless, there still remain three practical issues related to the implementation of sliding-window aggregation. The first issue is that it is necessary to consider out-of-order arrival of tuples at a windowing operator. To address the issue, this work presents an order-agnostic implementation of a sliding-window aggregate query on an FPGA. The second issue is that a large number of overlapping sliding-windows cause severe scalability problems in terms of both performance and area. Instead of replicating a large number of aggregation modules, each sliding window is divided into non-overlapping sub-windows called panes. Results obtained in this work indicate that the pane-based approach can provide significant benefits in terms of performance (i.e., the maximum allowable clock frequency), area (i.e., the hardware resource usage), and scalability. Finally, the third issue is that there is a lack of run-time configurability, which severely limits the practical use in a wide range of applications. To address the problem, the present study proposes a novel query accelerator, namely Configurable Query Processing Hardware (CQPH). CQPH is an FPGA-based query processor that contains a collection of configurable hardware modules, especially designed for sliding-window aggregate queries. As a proof of concept, a prototype of CQPH is implemented on an FPGA platform for a case study. Evaluation results indicate that the prototype implementation of CQPH with a Gigabit Ethernet interface can process a packet stream at wire-speed without packet loss. Since the Gigabit Ethernet is not sufficient to saturate the CQPH, a DDR3 SDRAM module is used as a high-speed data source. Results indicate that the prototype of CQPH can execute multiple queries simultaneously without sacrificing the performance (i.e., throughput) even if the data rate reached more than 10 Gbps.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

An important and growing class of applications requires to process online data streams on the fly in order to identify emerging trends in a timely manner. Many data processing tasks, such as financial application [23] and traffic monitoring [48], are required to process high-rate data sources with certain time restrictions. To address the issue, database researchers have expanded the data processing paradigm from the traditional "store-and-process" model toward the "stream-oriented processing" model [2, 4, 8].

*Data Stream Management Systems* (DSMSs) [3] deal with potentially infinite streams of data that should be processed for real-time applications, executing SQL-like *continuous queries* [6] over data streams. It is essential for DSMSs that incoming data be processed in real time, or at least near real-time, depending on the applications' requirements. In particular, low-latency and high-throughput processing are key requirements of systems that process unbounded and high-rate data streams rather than fixed-size stored data sets.

One of the key challenges for DSMSs is an efficient support for *sliding-window queries* [5] over unbounded streams. Indeed, it is considered one of the eight key requirements in [41] that a stream processing system must have a highly-optimized and minimal-overhead execution engine to deliver real-time response for high-volume applications. In order to meet the above-mentioned requirement, there is currently a great deal of interest in the potential of using field-programmable gate arrays (FPGAs) as custom accelerators for continuous query processing over data streams [24, 26, 25, 44, 38, 40, 39, 29].

## 1.1 Motivating Issues

Mueller *et al.* consider the use of FPGAs for data stream processing as co-processors [25]. In particular, they propose an implementation method for sliding-window aggregate queries on an FPGA. In fact, it is a common approach that subsequence of data stream elements (hereafter *tuples*) is defined as a *window*. In other words, windows decompose a data stream into possibly overlapping subsets of tuples (*i.e.,* each

tuple belongs to multiple windows). After that, according to a given query, window-aggregate operators repeatedly calculate aggregate functions such as COUNT, SUM, AVERAGE, MIN, and MAX for each window. Nevertheless, there still remain three practical issues related to the implementation of sliding-window aggregation:

1. The first issue is that it is necessary to consider out-of-order arrival of tuples at a windowing operator.

2. The second issue is that a large number of overlapping sliding-windows cause severe scalability problems in terms of both performance and area.

3. The third issue is that there is a lack of run-time configurability, which severely limits the practical use in a wide range of applications.

Unfortunately, the above issues are neither discussed nor addressed in the previous work [25], and to the best of our knowledge, each of them is still an open question for hardware-based approaches. In the following subsections, a brief explanation is given for each problem.

### 1.1.1 Out-of-order arrival of tuples

The implementation technique adopted in [25] relies on an implicit assumption about the physical order of incoming tuples, that is to say, tuples arrive in correct order at a windowing operator. Obviously, this assumption simplifies the definition and implementation of sliding windows; however, it does not always fit into a realistic setting where some degree of disorder (*i.e.,* out-of-order arrival of tuples) might be expected.

It is mentioned in [21] that previous works on data streams commonly model a data stream as an unbounded sequence of tuples arriving in order of some timestamp-like attribute; however, disorder naturally occurs in real-world stream systems. This means that, in reality, we cannot always assume all tuples to be ordered by their timestamp values when they arrive to a DSMS. For example, input tuples arriving over a network from remote sources may take different paths with different delays. As a result, some tuples may arrive out of sequence according to their timestamp values.

### 1.1.2 Scalability issue for overlapping sliding-windows

It is stated in [19] that sliding-window aggregate queries allow users to aggregate input streams at a user-specified granularity and interval, and thereby provide the users a flexible way to monitor streaming data. In other words, user-defined queries can determine the number of overlapping sliding-windows. Nevertheless, the previous approach adopted in [25] relies on a simple replication strategy of aggregation

units for overlapping sliding-windows. As a result, a large number of aggregation circuits are instantiated on an FPGA, and this leads to serious scalability problems in terms of clock frequency (*i.e.,* performance) and hardware resource usage (*i.e.,* area).

### 1.1.3 Lack of run-time configurability

A common limitation from which the most FPGA-based approaches suffer is that the existing approaches impose significant overhead on run-time query registration/modification. It is mentioned in [29] that while supporting query modification at run time is almost trivial for software-based techniques, they are highly uncommon for custom hardware-based approaches such as FPGAs. Moreover, as stated in [15], given the dynamic environment of data streams, queries can join and leave a streaming system at any time. It is therefore imperative for a query processing accelerator to support on-the-fly configurability for easy adaptation to the dynamic environment.

## 1.2 Objectives

There are three main objectives in this dissertation:

1. The first objective is to address the problem of out-of-order arrival of tuples and propose an alternative approach to implement a sliding-window aggregate query on an FPGA.

2. The second objective is to address the scalability problem and propose another approach to implement a sliding-window aggregate query on an FPGA in an *efficient* and *scalable* manner.

3. The third objective is to address the problem of the lack of run-time configurability and propose a novel query accelerator, namely Configurable Query Processing Hardware (CQPH).

### 1.2.1 Order-agnostic implementation technique

First, the dissertation presents an order-agnostic implementation of a sliding-window aggregate query on an FPGA, based on a one-pass query evaluation strategy called the *W*indow-*ID* (WID) [20]. With the proposed method, we can process out-of-order tuples at wire speed due to the one-pass query evaluation strategy and simultaneous evaluations of overlapping sliding-windows by taking advantage of the hardware parallelism. The proposed implementation can handle disorder by utilizing punctuations [47]. It is stated in [20] that WID does not require a specific type of assumption about the physical order of tuples in a data stream and can process out-of-order tuples as they arrive without sorting them into the "correct" order. Since the proposed implementation is based on WID approach, it can also process input tuples on the fly without reordering them into the correct order.

### 1.2.2 Efficient and scalable hardware design

Secondly, the dissertation presents a scalable hardware design of sliding-window aggregation and its implementation on an FPGA, by examining and integrating two key concepts: Pane [19] and *W*indow-*ID* (WID) [20]. Instead of replicating a large number of aggregation modules for overlapping sliding-windows, we divide each sliding window into non-overlapping sub-windows called *panes*. For each sub-window, or pane, we first calculate a sub-aggregate (*i.e.,* pane-aggregate), which is then shared by the aggregation of the multiple windows (*i.e.,* overlapping sliding-windows). The pane-based approach is originally proposed for software-based implementation to reduce the required buffer size and the computation cost [19]. In this work, however, we show that the same idea can provide significant benefits for hardware-based implementation, especially in terms of performance (*i.e.,* the maximum allowable clock frequency), area (*i.e.,* the hardware resource usage), and scalability.

### 1.2.3 CQPH: Configurable Query Processing Hardware

Finally, the dissertation presents the design and evaluation of Configurable Query Processing Hardware (CQPH), a highly-optimized and minimal-overhead query processing engine, especially designed for sliding-window aggregate queries. CQPH is an FPGA-based query processor that contains a collection of configurable hardware modules, each of which supports selection (*i.e.,* filtering), group-by operation (*i.e.,* partitioning), and sliding-window aggregation. CQPH is highly optimized for performance with a fully pipelined implementation to exploit the increasing degree of parallelism that modern FPGAs support. In addition, the proposed design imposes minimal overhead on query configuration. More specifically, CQPH can support registration of new queries as well as modification of existing queries, without a time-consuming compilation process which is a common drawback of the previous approaches.

## 1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 provides necessary background and briefly reviews related work. Chapter 3 presents an order-agnostic implementation of a sliding-window aggregate query on an FPGA. Chapter 4 presents a scalable hardware design of sliding-window aggregation and its implementation on an FPGA. Chapter 5 presents the design and evaluation of CQPH. Finally, Chapter 6 concludes the dissertation by summarizing the results and identifying future work.

# Chapter 2

# Background and Related Work

Chapter 2 briefly reviews related work and provides some background information.

## 2.1 Data Processing on FPGAs

### 2.1.1 Acceleration of Data-intensive Operations using FPGAs

As stated in [45], FPGAs are an increasingly attractive alternative to overcome the architectural limitations of commodity hardware. Mueller *et al.* show the potential of FPGAs as an accelerator for data-intensive operations [24]. It is demonstrated in [24] that FPGAs can achieve competitive performance compared to modern general-purpose CPUs while providing remarkable advantages in terms of power consumption and parallel stream evaluation. Indeed, recently a number of research efforts have used FPGAs to target acceleration of data-intensive workloads. For instance, Putnam *et al.* propose an FPGA-based reconfigurable fabric, called *Catapult*, to accelerate datacenter workloads [37]. They introduce the Catapult fabric into 1,632 servers, and demonstrate its efficacy in accelerating the Bing web search engine. Some other works focus on the idea of using custom hardware to accelerate database queries [49, 9, 10, 53, 52]. Woods *et al.* propose an intelligent storage engine, called *Ibex*, which supports advanced SQL off-loading [49]. Dennl *et al.* propose partial reconfiguration-based approach to accelerate a subset of SQL queries for traditional database systems [9, 10]. Yoshimi *et al.* present acceleration of OLAP workload on interconnected FPGAs with flash storage [53, 52].

### 2.1.2 Continuous Query Processing on FPGAs

FPGAs are used to process data streams due to their low-latency and high-throughput processing advantages. Most of the previous FPGA-based approaches require full circuit compilation to implement dedicated hardware for different kinds of query workloads. In general, these query-tailored circuits are

inherently very efficient in terms of performance, area and power consumption; however, static compilation process is highly CPU-intensive and imposes significant overhead on dynamic workload changes. For example, Sadoghi *et al.* propose an efficient event-processing platform called *fpga-ToPSS* [38], and demonstrate high-frequency and low-latency algorithmic trading solutions [40]. These projects mainly focus on queries with selection operator.

### Window Joins

How to implement stream joins on FPGAs is indeed a challenging task. It is mentioned in [44] that the M3Join proposed by Qian *et al.* [7] implements the join step as a single parallel lookup; however, this approach causes the significant performance drop for larger join windows. Alternatively, another study [39] concentrates on the execution of SPJ (Select-Project-Join) queries with multi-query optimization. Terada *et al.* suggest another approach to implement window join operator on an FPGA [43]. Nevertheless, only two join processes are concurrently executed since their approach is based on sequential execution.

To address the issue, Teubner and Mueller propose a new join algorithm called *handshake join* [44]. Data flow model of handshake join does not suffer from the limitation mentioned above and the previous study such as [30, 31, 32, 33] focus on the acceleration of handshake join on FPGA. Finally, Kung and Leiserson propose the idea of *systolic array* that is a structure composed of an array of processors for VLSI implementation [18]. It is stated in [18] that processing units of a systolic array rhythmically compute and pass data through the system. The data processing and communication model of *join cores* [30, 31, 32, 33] are consistent with the properties of systolic arrays. In fact, the data flow model of the handshake join is very similar to the join arrays proposed for relational databases [17].

### Glacier: A query-to-hardware compiler

An important study closely related to this work is *Glacier* [25, 26]. Mueller *et al.* propose a query-to-hardware compiler, called Glacier, for continuous queries [25]. The compiler takes a query plan as its input and produces VHDL description of a logic circuit that implements the input plan. In particular, it provides a library of components for basic operators such as selection, aggregation, grouping, and windowing operators. Glacier can generate logic circuits by composing the library components on an operator-level basis, and thereby can support a wide range of continuous queries involving the basic operators (*i.e.,* selection, aggregation, grouping, and windowing operators). The windowing operator provides sliding-window functionality and aggregate operator includes four distributive (*i.e.,* COUNT, SUM, MIN, and MAX) and an algebraic (*i.e.,* AVERAGE) aggregate functions [14]. As windowing and aggregation operators are provided by the library, Glacier can compile sliding-window aggregate queries into hardware circuits to be implemented on an FPGA.

As described in Chapter 1, there are three issues regarding the sliding-window aggregate queries implemented by Glacier:

- out-of-order arrival of tuples,
- scalability issue for overlapping sliding-windows, and
- lack of run-time configurability.

In the following chapters, three alternative approaches are presented in detail to address each issue.

### 2.1.3 Run-time Configuration of FPGAs

Most of the previous FPGA-based approaches suffer from a common limitation, namely, lack of flexibility to adapt to dynamic workload changes. One possible solution is to exploit partial reconfiguration technology of FPGAs along with prebuilt libraries of custom-designed components. For example, Dennl *et al.* [9, 10] propose partial reconfiguration-based approach to accelerate a subset of SQL queries for traditional database systems. It is stated, however, in [46] that the partial reconfiguration causes another level of complexity which severely limits its use in real-world systems.

Another promising solution is the approach adopted in *skeleton automata* [45, 46] or *Flexible Query Processor* (FQP) [29, 27, 28]. The main idea is to implement a generic template circuit along with a number of configuration registers/memories. With this approach, the template circuit can be easily configured for a specific type of workloads by changing the values of the configuration registers/memories within the template design. For instance, the skeleton automata and FQP can support XML projections and sliding-window join queries, respectively. The major advantage of both works is that they can offer run-time configurability without long running static compilation or the partial reconfiguration. CQPH shares similar motivation with skeleton automata and FQP; however, target workloads are quite different. The main focus of the dissertation differs from these works as we are primarily concerned with sliding-window aggregate queries which are not in the scope of [45, 46, 29].

## 2.2 Sliding-Window Aggregation

### 2.2.1 Example of a sliding-window aggregate query

Consider the following online auction example taken from Li *et al.* [19]. In this example, an online auction system monitors bids on auction items. We assume an input stream that contains information about each bid, the schema of which is defined as ⟨*item-id, bid-price, timestamp*⟩. In addition, assume that the online auction system runs over the Internet, and each bid is streamed into a central auction

```
SELECT max(bid-price), timestamp
  FROM bids [RANGE 4 minutes
             SLIDE 1 minute
             WATTR timestamp]
```

Figure 2.2.1: $Q_1$: "Find the maximum bid-price for the past 4 minutes and update the result every 1 minute."

server where a DSMS is running. Query $Q_1$ (cited from [19]) shows an example of a sliding-window aggregate query (see Fig. 2.2.1). This query can be used to find the maximum bid-price for the past 4 minutes and update the result every 1 minute.

#### Window specification

It is stated in [5] that since data streams are infinite, queries that execute over a data stream need to define a region of interest (termed a window). Following the definition of window semantics [20], Query $Q_1$ introduces a window specification which consists of a window type and a set of parameters that define a window. In Query $Q_1$, *sliding windows* have three parameters: *RANGE*, *SLIDE*, and *WATTR*. *RANGE* indicates the size of the windows; *SLIDE* indicates the step by which the windows move; *WATTR* indicates the windowing attribute—the attribute over which *RANGE* and *SLIDE* are specified [20].

It is important to note that the window specification can be user-defined values; therefore, as stated in [19], sliding-window aggregate queries allow users to aggregate the stream at a user-specified granularity (*i.e., RANGE*) and interval (*i.e., SLIDE*). Given the specification above, for example, the bid stream is divided into overlapping 4-minute windows starting every minute, based on the timestamp attribute of each tuple. Fig. 2.2.2 illustrates overlapping sliding-windows (only the first four windows) for Query $Q_1$. Notice that all windows in Fig. 2.2.2 have *RANGE* = 4 and *SLIDE* = 1, respectively.

#### Output generation

Window-aggregate operators are generally classified as a blocking operator in DSMSs; therefore, it is necessary to unblock them at the end of each window. One of the common approaches for DSMSs to unblock aggregate operators is to use timestamp attribute of each tuple. For example, in Query $Q_1$, a MAX aggregate operator can keep track of timestamp values of input tuples (*i.e.,* bids) to identify the end of each sliding-window. In this case, the maximum value of bid-price can be calculated at the end of each window and Query $Q_1$ produces an output tuple with schema ⟨*max, timestamp*⟩ where the timestamp attribute specifies the end of the window. For interested readers, some key applications of

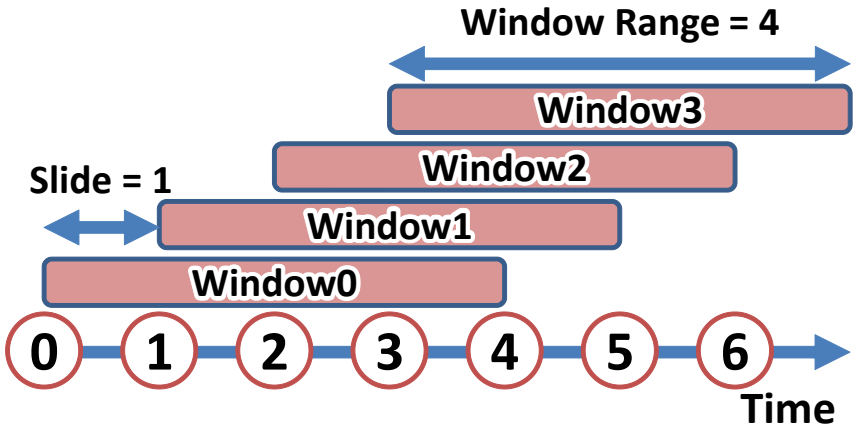



Figure 2.2.2: Overlapping Sliding-Windows for Query $Q_1$ (*RANGE* = 4 minutes and *SLIDE* = 1 minute).

sliding-window queries are discussed in [5].

### 2.2.2 Handling out-of-order arrival of tuples

Abadi *et al.* [2] classifies types of operators as *order-agnostic* or *order-sensitive*. Order-agnostic operators can always process tuples in the order in which they arrive whereas order-sensitive operators can only be guaranteed to execute with finite buffer space and in finite time if they can assume some ordering (potentially with bounded disorder) over their input streams [2].

#### *W*indow-*ID* (WID)

Li *et al.* presents a software-based implementation of order-agnostic window aggregation [20]. Their approach is called *W*indow-*ID* (WID), and it is stated in [21] that WID provides a method to implement window aggregate queries in an order-agnostic way, by using special annotations called *punctuations* [47]. Informally, a punctuation can be regarded as a special tuple that contains some meta-data about a given stream. For instance, punctuations can be used to indicate that no more tuples having certain timestamp values will be seen in the stream [20]. In practice, punctuation tuples are embedded into a data stream, and those tuples can be used to unblock some blocking operators such as group-by and aggregation.

#### Slack Specification

Another approach to handle disorder is to use *slack* specification. Slack [2] defines an upper bound on the degree of disorder which can be handled by an operator. As mentioned before, we cannot expect that

all tuples always arrive in order of some timestamp-like attribute in a realistic setting. Instead, Aurora [2] assumes some ordering (potentially with bounded disorder) over input streams. Any tuple arriving after its corresponding period specified by a slack parameter is discarded. In Aurora, the slack parameter is used to specify the number of tuples to be stored and sorted before an order-sensitive operator processes input tuples. Aurora classifies window aggregation as an order-sensitive operation. In Aurora, therefore, aggregate operators require buffering and reordering of tuples before computation in order to handle disorder.

# Chapter 3

# Sliding-window Aggregate Operator over Out-of-order Data Streams

## 3.1 Abstract

This chapter presents the design and evaluation of an FPGA-based accelerator for sliding-window aggregation over data streams with out-of-order data arrival. We propose an order-agnostic hardware implementation technique for windowing operators based on a one-pass query evaluation strategy called Window-ID, which is originally proposed for software implementation. The proposed implementation succeeds to process out-of-order data items, or tuples, at wire speed due to the simultaneous evaluations of overlapping sliding-windows. In order to verify the effectiveness of the proposed approach, we have also implemented an experimental system as a case study. Our experiments demonstrate that the proposed accelerator with a network interface achieves an effective throughput around 760 Mbps or equivalently nearly 6 million tuples per second, by fully utilizing the available bandwidth of the network interface.

## 3.2 Design Concept

Although Glacie [25] is capable of compiling sliding-window aggregate queries into logic circuits, it implements a windowing operator as an order-sensitive operator. In other words, its implementation relies on an implicit assumption about the physical order of incoming tuples; that is to say, tuples arrive in correct order at the windowing operator. Obviously, this assumption simplifies the definition and implementation of sliding windows; however, it does not always fit into a realistic setting where some degree of disorder can be expected. Glacier does not discuss the issue regarding out-of-order arrival of tuples.

In order to address the problem, this dissertation proposes an alternative implementation technique for sliding-window aggregate queries. The proposed implementation follows the same approach as WID [20] to handle disorder. In other words, aggregation operation is order-agnostic, and punctuations are used to unblock window-aggregate operators. On the other hand, WID is proposed for a software-based implementation. The main interests of WID [20] are to calculate window aggregates with the one-pass evaluation strategy and to handle disorder by using punctuations. However, hardware-based implementation of order-agnostic window aggregation is neither provided nor discussed in [20].

Contrary to the software-based implementation proposed in [20], this dissertation presents hardware-based implementation which handles multiple windows with a single clock cycle. The proposed implementation instantiates multiple window-aggregation modules by taking advantage of hardware parallelism. Upon arrival of a new tuple, each of the window-aggregation modules can simultaneously evaluate the tuple within the same clock cycle. This is the main difference between software-based WID [20] and our proposed approach.

The number of the window-aggregation modules is determined by using window parameters (*RANGE* and *SLIDE*) and a slack [2] specification. As mentioned in Chapter 2, Aurora [2] uses a slack parameter to determine the number of tuples to be buffered and reordered before aggregation. The proposed approach, however, relies on punctuations to handle disorder, and the slack parameter is used to calculate the number of the window-aggregation modules required to be instantiated. This is a significant difference between the approach adopted in this work and that of Aurora.

## 3.3 Motivating Application

Glacier [25] demonstrates how to implement a window aggregate query on an FPGA. For example, the implementation of the third query of [25] includes a windowing operator which implicitly relies on the arrival sequence of input tuples. Contrary to Glacier, the proposed approach permits windowing on any attribute, allowing a bounded disorder of the tuples. This work focuses on the same query (*i.e.,* the third query of [25]) as a case study and shows how to implement the query in an order-agnostic manner. It should be noted that the proposed approach is general enough to apply a wide range of window aggregate queries which include the primitive aggregate functions such as COUNT, SUM, AVERAGE, MIN, and MAX (*i.e.,* the same aggregate functions supported by Glacier).

This work assumes the same financial application as that of the previous work [25]. Our approach, however, requires an explicit timestamp attribute to define windows over an input stream. Instead of a simple sequence number considered in [25], a timestamp attribute has been added to the schema of the input stream. The schema of the stream, called *Trades*, is defined in Fig. 3.3.1. This schema describes the structure of an input tuple. Each tuple of the stream consists of four attributes (*i.e.,* Symbol, Price,

```
CREATE INPUT STREAM Trades(
  Symbol char[4],   -- valor symbol
  Price  int,       -- stock price
  Volume int,       -- trade volume
  Time   int)       -- timestamp
```

Figure 3.3.1: The schema of the input stream. Each tuple of the stream consists of four attributes: Symbol, Price, Volume, and Time.

```
SELECT Time, count(*) AS Number
  FROM Trades [RANGE 600 seconds
               SLIDE 60 seconds
               WATTR Time]
 WHERE Symbol = "UBSN"
```

Figure 3.3.2: $Q_2$: "Count the number of trades of UBS (Union Bank of Switzerland) shares for the past 10 minutes (600 seconds) and update the result every 1 minute (60 seconds)."

Volume, and Time) each of which is represented as a 32-bit value. Based on the definition of window semantics [20], the example query can be written as in Fig. 3.3.2.

In Query $Q_2$, *WATTR* indicates the windowing attribute (*i.e.,* Time) over which *RANGE* and *SLIDE* are specified. Given the window specification of Query $Q_2$, the input stream is divided into overlapping 10-minute windows starting every minute, based on the *Time* attribute of each tuple. For each window, $Q_2$ counts the number of tuples which satisfy the condition (*i.e.,* WHERE Symbol = "UBSN"), and it generates an output stream with schema ⟨*Time, Number*⟩ where the *Time* attribute indicates the end of the window. The details of the implementation of Query $Q_2$ are provided in the following section.

## 3.4 Implementation Details

### 3.4.1 Wiring Interface

In this work, the same wiring interface as that of the previous work [25] is adopted to represent data flow. That is to say, each *n*-bit-wide data is regarded as a set of *n* parallel wires. Furthermore, additional *punctuation flag* and *data valid* lines each of which is a one-bit signal indicate the presence of a punctuation and a tuple, respectively. For example, datum on the multiple lines (*i.e., n* parallel wires) is considered

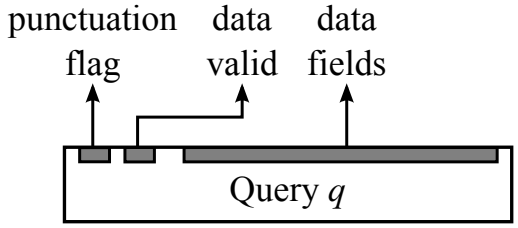



Figure 3.4.1: Wiring Interface for a query $q$.

as a punctuation when their *punctuation flag* is asserted (*i.e.,* set to logic "1"). Similarly, the data lines are regarded as a tuple when their *data valid* field is asserted.

Following the notation of [25], Fig. 3.4.1 shows a black box view of the hardware implementation for a given query $q$. In Fig. 3.4.1, the gray-shaded boxes represent flip-flop registers. It is mentioned in [25] that their circuits are all clock-driven (*i.e., synchronized*), and each operator writes its output into a flip-flop register after processing. The present work follows the same approach and implements a fully pipelined hardware for the sliding-window aggregate query.

### 3.4.2 Hardware Execution Plan

Hardware execution plan for Query $Q_2$ is illustrated in Fig. 3.4.2. As shown in Fig. 3.4.2, the proposed implementation adopts a 4-stage pipeline architecture. Intermediate results are stored into flip-flop registers (*i.e.,* the gray-shaded boxes) at the end of the each stage. In other words, these registers can be regarded as pipeline registers, and each stage of the pipeline can use the result of the previous stage. It should be noted that each stage requires only one clock cycle to complete. The arrows indicate the connections between the pipeline stages, as well as between hardware components. According to the notation adopted in the previous work [25], name of a specific field in the *data fields* is explicitly identified with its label wherever appropriate. It should be also mentioned that the label "$*$" means "all of the remaining fields" in the data bus.

#### Selection Operation

The beginning two stages of Fig. 3.4.2 correspond to a selection operation. In **Stage 1**, *Symbol* field of the data bus is compared to a constant value ("UBSN") which is specified in the WHERE expression of Query $Q_2$ (indicated as $\boxed{=}$ in Fig. 3.4.2). At the same time, the result of the comparison is labeled as a one-bit *is_equal* flag and added to the data bus. In **Stage 2**, a logical AND gate (indicated as $\boxed{\&}$ in Fig. 3.4.2) computes whether an input tuple is valid or not. If the tuple should be discarded, the *data valid* flag is negated (*i.e.,* set to logic "0") for the next pipeline stage. Actually, these two stages can be

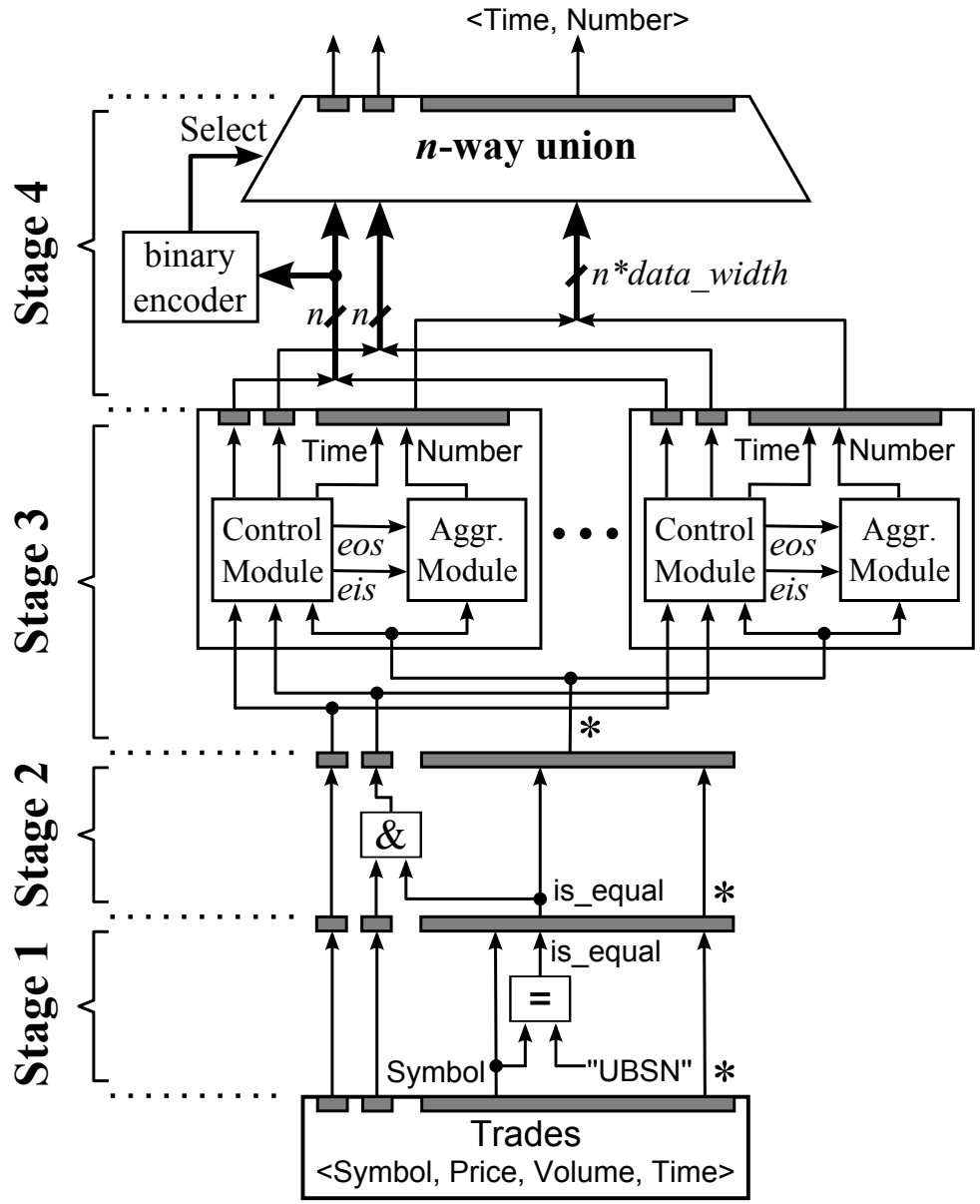



Figure 3.4.2: Hardware execution plan for Query $Q_2$.

implemented in a straightforward way based on the approach proposed in the previous work [25]. The main difference, however, is the presence of the *punctuation flag* field which is required for **Stage 3**. It is stated in [20] that some operators, such as selection, simply pass punctuations through to the next operator in a query plan. **Stage 1** and **Stage 2** meet the above requirement since *punctuation flag* field is directly connected to the next stage of the pipeline as shown in Fig. 3.4.2.

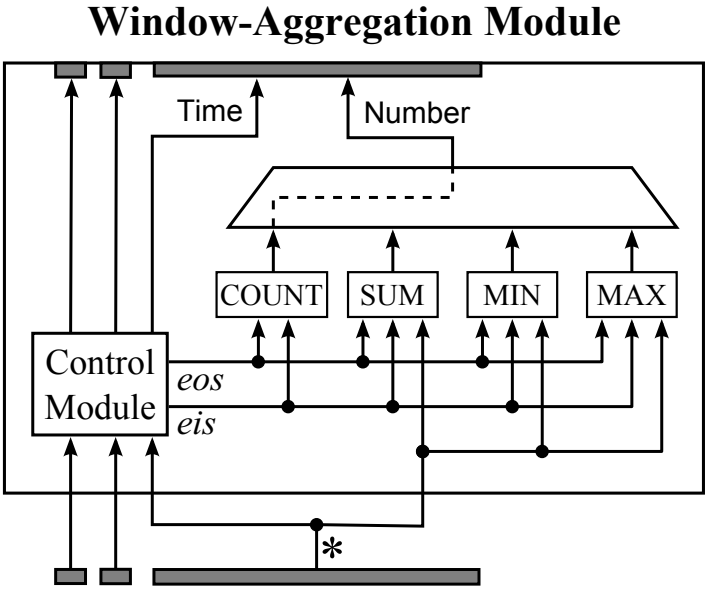



Figure 3.4.3: Block diagram of a window-aggregation module.

**Windowing and Aggregation**

The next step is **Stage 3** of the pipeline which corresponds to windowing and aggregation operators. In **Stage 3**, a number of window-aggregation modules are instantiated as shown in Fig. 3.4.2. They provide sliding-window functionality and can concurrently compute aggregate functions. The number of window-aggregation modules to be instantiated is calculated by using *RANGE*, *SLIDE*, and *SLACK* parameters (see the following Equation 3.4.1 and 3.4.2).

$$N_{\text{WIN}} = \left\lceil \frac{RANGE}{SLIDE} \right\rceil + x, \text{ where } x \in \mathbb{Z}^{+} \tag{3.4.1}$$

$$x \geq \frac{SLACK + RANGE}{SLIDE} - \left\lceil \frac{RANGE}{SLIDE} \right\rceil \tag{3.4.2}$$

We should determine the number of the window-aggregation modules required to handle disorder specified by a slack parameter. For example, assume that *SLACK* value is set to 60 seconds for Query $Q_2$. Recall from Query $Q_2$ that window parameters are specified as $RANGE = 600$ seconds and $SLIDE = 60$ seconds, respectively. Applying Equation 3.4.2, we obtain $x \geq \frac{60+600}{60} - \left\lceil \frac{600}{60} \right\rceil \Rightarrow x \geq 1$. After that, using Equation 3.4.1 with $x = 1$, we can calculate $N_{\text{WIN}} = \left\lceil \frac{600}{60} \right\rceil + 1 \Rightarrow N_{\text{WIN}} = 11$.

**Aggregation Module.** Each instance of window-aggregation module consists of *aggregation module* and *control module* as shown in **Stage 3** of Fig. 3.4.2. A more detailed block diagram of a single window-

aggregation module is depicted in Fig. 3.4.3. It is stated in [25] that primitive aggregate functions such as COUNT, SUM, AVERAGE, MIN, and MAX can be implemented in a straightforward fashion on an FPGA. Glacier [25] supports the above five aggregate functions, and in this work, we also focus on the same aggregate operators. In fact, AVERAGE can be obtained with the combination of two aggregate values: SUM and COUNT. It is therefore necessary to implement the other four aggregate operators as shown in Fig. 3.4.3. When it comes to the implementation of Query $Q_2$, a standard counter component can be used inside the aggregation module. Since Query $Q_2$ requires count($*$) function, the result of the COUNT operator is selected as the output value (indicated as the broken line in Fig. 3.4.3).

Aggregation module maintains partial aggregates and only stores the current (partial) result of aggregation instead of buffering all input tuples belonging to the current window. In other words, it incrementally computes the aggregate result as new tuples arrive and always keeps the latest result. In order to maintain the aggregate value, it requires two control signals: enable input stream (*eis*) and end of stream (*eos*). These signals are provided by the control module as illustrated in Fig. 3.4.3. Both *eis* and *eos* are asynchronous signals each of which is a one-bit flag. The *eis* signal indicates whether or not datum on the *data fields* should be considered as a valid tuple for the current window. Whenever *eis* is asserted, the aggregation operator accepts input tuple and records its contribution to the partial result. If *eis* is negated, the aggregation operator simply ignores the input data and waits for the next tuple to arrive. The other signal, *eos*, indicates whether an input stream reaches the end of the current window. When *eos* is asserted, it means that the current window is no longer active, and the aggregate operator should reset its internal state to be ready for input tuples belonging to the next window.

**Control Module.** Each control module maintains its own window states and provides two control signals (*i.e., eis* and *eos*) to the aggregation module. The control module maintains window states by updating its internal registers called $win_{\text{begin}}$ and $win_{\text{end}}$. These registers represent the beginning and the end of the current window, respectively. The control module uses $win_{\text{begin}}$ and $win_{\text{end}}$ registers to generate the two control signals, *eis* and *eos*. Details about how to maintain these registers and to generate the control signals are provided in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 describes how to initialize $win_{\text{begin}}$ and $win_{\text{end}}$ registers. Recall from Query $Q_2$ that windowing attribute (*i.e., WATTR*) is specified as *TIME*. Therefore, $WATTR_{\text{start}}$ can be considered as $TIME_{\text{start}}$ which means the start time of the execution of the query. It can be determined in several ways. For example, if one knows when to start the query (*e.g.,* market opening times for Query $Q_2$), this information can be used to determine $TIME_{\text{start}}$ value. Another option is to use the most recent value of the punctuation. One can also use the timestamp value of the first tuple with the *SLACK* parameter (*e.g.,* $TIME_{\text{start}} = TIME_{\text{first tuple}} - SLACK$). The other parameters (*i.e.,* $N_{\text{WIN}}$, *RANGE*, *SLIDE*) can be determined before the execution of the query; thus, they are regarded as constant parameters.

**Algorithm 1** Maintain window states ($win_{\text{begin}}$ and $win_{\text{end}}$)

**State Registers:**

$win_{\text{begin}}(i)$: the beginning of the $i$-th window instance

$win_{\text{end}}(i)$: the end of the $i$-th window instance

**Initialization:**

**for all** $i$ such that $1 \leq i \leq N_{\text{WIN}}$ **do**

$win_{\text{begin}}(i) \leftarrow WATTR_{\text{start}} + (i-1) \times SLIDE$

$win_{\text{end}}(i) \leftarrow WATTR_{\text{start}} + (i-1) \times SLIDE + RANGE$

**end for**

**Synchronous Update:**

**for all** $i$ such that $1 \leq i \leq N_{\text{WIN}}$ **do**

**for each** clock cycle **do**

**if** *punctuation flag* is asserted **and**

$WATTR \geq win_{\text{end}}(i)$ **then**

$win_{\text{begin}}(i) \leftarrow win_{\text{begin}}(i) + N_{\text{WIN}} \times SLIDE$

$win_{\text{end}}(i) \leftarrow win_{\text{end}}(i) + N_{\text{WIN}} \times SLIDE$

**end if**

**end for**

**end for**

In addition to initialization, Algorithm 1 shows the update operation of the registers. Initialization or update operation described in Algorithm 1 can be completed in one clock cycle. It should be emphasize that the control module included in each window instance has its own window states. Notice that $win_{\text{begin}}(i)$ and $win_{\text{end}}(i)$ are correspond to the $i$-th window instance. All window instances concurrently perform the same operation on each cycle in a synchronous manner.

The generation of the asynchronous control signals is described in Algorithm 2. Notice that $eis(i)$ and $eos(i)$ are correspond to the $i$-th window instance. The control module included in each window instance generates these signals, using its own window states as well as input signals. It is important to notice that the implementation of $eis(i)$ and $eos(i)$ signals is fully asynchronous. The aggregation module can use the control signals as soon as they are generated within the same clock cycle. In other words, all of the operations performed in both the control module and the aggregation module can be completed in a single clock cycle.

**Algorithm 2** Generate asynchronous signals (*eis* and *eos*)

**Asynchronous Signals:**

*eis*($i$): input enable signal for the $i$-th window instance

*eos*($i$): output enable signal for the $i$-th window instance

**Asynchronous Update:**

**for all** $i$ such that $1 \leq i \leq N_{\text{WIN}}$ **asynchronously do**
  **if** *punctuation flag* is negated **then**
    negate *eos*($i$) signal
    **if** *data valid* is asserted **and**
    $win_{\text{begin}}(i) \leq WATTR < win_{\text{end}}(i)$ **then**
      assert *eis*($i$) signal
    **else**
      negate *eis*($i$) signal
    **end if**
  **else** {*punctuation flag* is asserted}
    negate *eis*($i$) signal
    **if** $WATTR \geq win_{\text{end}}(i)$ **then**
      assert *eos*($i$) signal
    **else**
      negate *eos*($i$) signal
    **end if**
  **end if**
**end for**

### Union Operation

It is stated in [25] that, from a data flow point of view, the task of an algebraic union operator is to merge the outputs of several source streams into a single output stream. As shown in Fig. 3.4.2, the $n$-way union operator merges the outputs of $n$ window-aggregation modules and generates a single result stream.

The implementation of the union operator is based on a multiplexer component. According to a select signal, the multiplexer component transfers the result of $i$-th window-aggregation module to the output registers of **Stage 4**. As illustrated in Fig. 3.4.2, the select signal is provided by a binary encoder component. It should be also mentioned that **Stage 4** requires only one clock cycle to complete its operation.

Table 3.1: Specifications of the Virtex®-6 FPGA (XC6VLX240T).

| | |
|---|---|
| # of Slice Registers | 301,440 |
| # of Slice LUTs | 150,720 |
| # of Slices | 37,680 |
| # of BRAM (36Kbit) | 416 |

Glacier [25] evaluates the complexity and performance of the resulting circuits in terms of *latency* and *issue rates*. Issue rate is defined as the number of tuples that can be processed per clock cycle. The overall latency and the issue rate of the proposed implementation are 4 cycles and 1 tuple/cycle, respectively.

## 3.5 Evaluation

The proposed design is implemented on a Virtex®-6 FPGA (XC6VLX240T) included in the Xilinx ML605 Evaluation Kit [51]. The specification of the FPGA used to implement the design is given in Table 3.1. Xilinx ISE 14.7 is used as an FPGA development environment during the implementation process (*e.g.,* synthesis, map, and place & route).

### 3.5.1 Resource Utilization and Performance

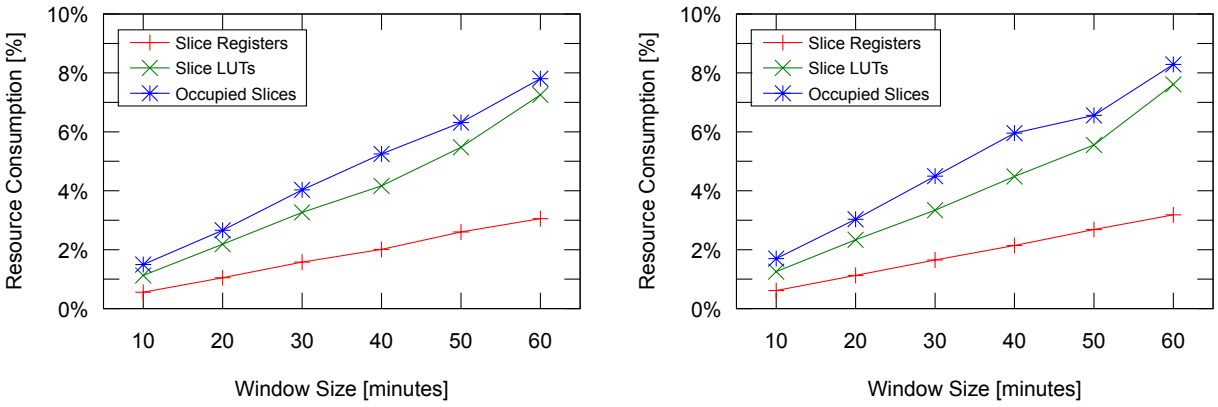
In order to evaluate the resource utilization and performance of the proposed design, Query $Q_2$ is implemented with different sizes of sliding windows. The *RANGE* of a window is increased from 10 minutes to 60 minutes, by increments of 10 (*i.e.,* a total of six different configurations). It should be also noted that all of the implemented queries have the same *SLIDE* parameter as Query $Q_2$ (*i.e.*, 60 seconds). A baseline design is implemented with $SLACK = 0$ as a reference point and the proposed design is implemented with $SLACK = 60$, respectively. Each design is synthesized with a timing constraint of 6.37 ns, which yields the target clock frequency of 157 MHz.

#### Resource Utilization

The circuit size of each design is measured in terms of the number of slice registers, the number of slice LUTs (Look-Up Tables), and the number of occupied slices. The resource usage is expected to increase with respect to the number of the window-aggregation modules instantiated on the target device. The number of these modules can be easily calculated by Equation 3.4.1 and Equation 3.4.2, using *RANGE*,

Table 3.2: Number of the window-aggregation modules with respect to the window size.

| | Size of the Time-based Sliding Window (*i.e.*, *RANGE*) | | | | | |
|---|---|---|---|---|---|---|
| | 10 min | 20 min | 30 min | 40 min | 50 min | 60 min |
| Baseline (*SLACK* = 0) : | 10 | 20 | 30 | 40 | 50 | 60 |
| Proposed (*SLACK* = 60) : | 11 | 21 | 31 | 41 | 51 | 61 |



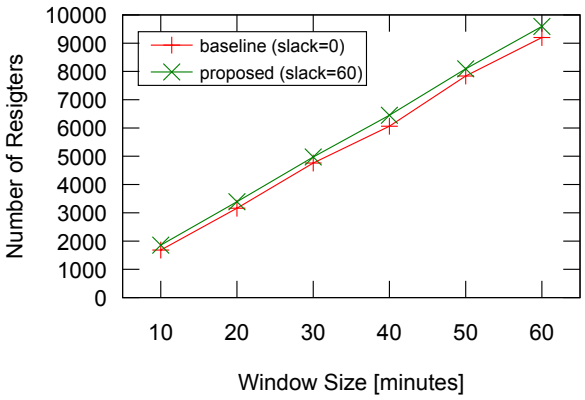

(a) Baseline implementation with *SLACK* = 0.

(b) Proposed implementation with *SLACK* = 60.

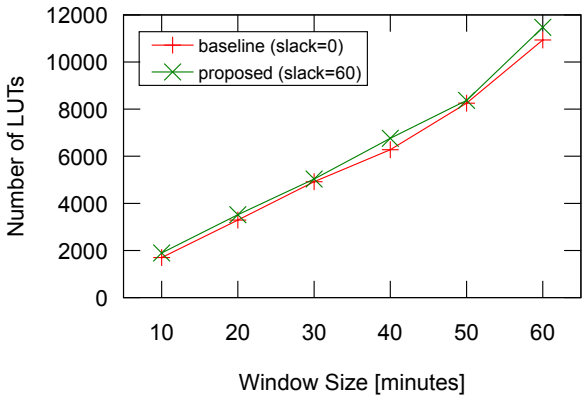
Figure 3.5.1: Overall resource consumption as a percentage of the total available resources on the target FPGA (Xilinx XC6VLX240T) with respect to the window size (*i.e.*, *RANGE*).

*SLIDE*, and *SLACK* parameters. For a reference, Table 3.2 summarizes the number of the window-aggregation modules for both *baseline* (*SLACK* = 0) and *proposed* (*SLACK* = 60) implementations.

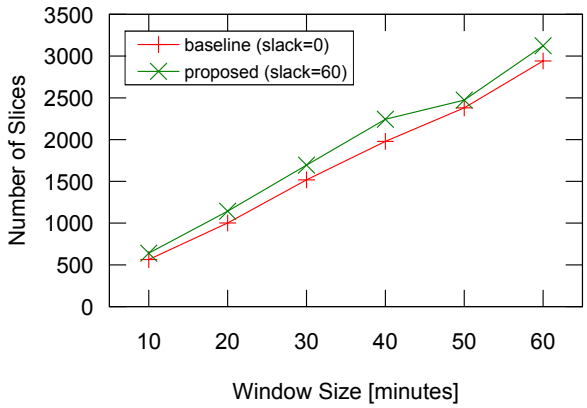
Overall resource consumption is plotted in Fig. 3.5.1. The x-axes of Fig. 3.5.1a and Fig. 3.5.1b represent the size of the time-based sliding window (*i.e.*, *RANGE*) from 10 to 60 minutes. The y-axes of the same figures indicate the resource consumption as a percentage of the total available resources on a Xilinx XC6VLX240T FPGA device. As shown in Fig. 3.5.1a or Fig. 3.5.1b, all three graphs (*i.e.*, Slice Registers, Slice LUTs, and Occupied Slices) are almost linearly increased with increasing window size, as expected. The increase in window size results in a higher $\frac{RANGE}{SLIDE}$ ratio. This implies an increase in the number of the window-aggregation modules (see Table 3.2). This is the main reason for the increased resource utilization. It should be also emphasized that a relatively small percentage of the overall FPGA resources is required to implement the query. For example, when the size of window is 10 minutes, slice usage is particularly low (less than 2%) for both baseline (*SLACK* = 0) and proposed (*SLACK* = 60) implementations. Even if the size of window is increased up to 60 minutes, overall slice utilization is




(a) Comparison of the number of registers.




(b) Comparison of the number of LUTs.

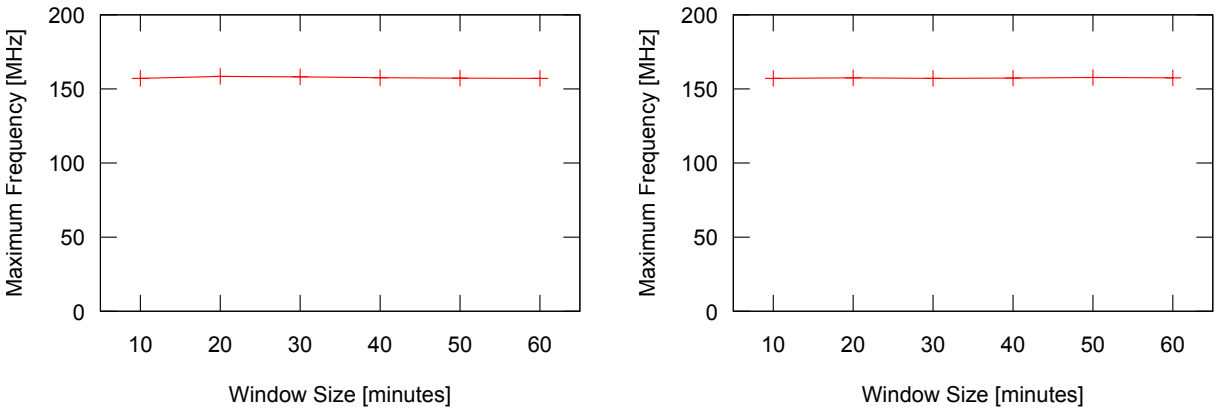



(c) Comparison of the number of slices.

Figure 3.5.2: Comparison of the resource usage between the baseline implementation ($SLACK = 0$) and the proposed implementation ($SLACK = 60$).

still less than 9%.

In order to evaluate the resource overhead of the proposed design, Fig. 3.5.2 summarizes the results of the comparison of the hardware-resource usage between the baseline implementation and the proposed implementation. The x-axes of Fig. 3.5.2a (registers), Fig. 3.5.2b (LUTs), and Fig. 3.5.2 (slices) represent the size of the time-based sliding window (*i.e.*, *RANGE*) from 10 to 60 minutes. The y-axes of the same figures indicate the actual number of each resource (*i.e.*, registers, LUTs, or slices) required to implement the design on the target FPGA device. Results indicate that the proposed implementation does not impose significant overhead on the resource usage over the baseline implementation. More specifically, the overhead is only a few percent of the total available resources on the target FPGA device. This means that the proposed implementation can handle a disordered stream ($SLACK \leq 60$ seconds) with a very reasonable hardware cost.




(a) Baseline implementation with $SLACK = 0$. (b) Proposed implementation with $SLACK = 60$.

Figure 3.5.3: Maximum clock frequencies of the implemented design on the target FPGA (Xilinx XC6VLX240T) with respect to the window size (*i.e.*, *RANGE*).

**Performance Evaluation**

The performance of the implemented design is evaluated in terms of the maximum clock frequency of the circuit for each window size. The clock frequency can be obtained from post-place & route static timing report, which is provided by Xilinx's Timing Analyzer tool. The obtained results are summarized in Fig. 3.5.3 for baseline ($SLACK = 0$) and proposed ($SLACK = 60$) implementations, respectively. The x-axes of Fig. 3.5.3a and Fig. 3.5.3b represent the size of the time-based sliding window (*i.e.*, *RANGE*) from 10 to 60 minutes. The y-axes of the same figures indicate the maximum clock frequencies of the implemented design on the target FPGA device.

As shown in Fig. 3.5.3a and Fig. 3.5.3b, each implementation achieves the target clock frequency of 157 MHz. Equivalently, this means that all implementations meet the timing constraint of 6.37 ns. Since the issue rate of the implemented queries is equal to 1 tuple/cycle, the proposed implementation can process up to 157 million tuples per second for different sizes of windows. It is also important to note that we obtained almost the same results for both baseline and proposed implementations as indicated in Fig. 3.5.3a and Fig. 3.5.3b. This means that an additional window-aggregation module of the proposed implementation does not impose any overhead on the performance of the overall circuit.

We can calculate the peak throughput by multiplying the data width of an input tuple by the clock frequency. Recall that the data width of a tuple is 128 bits; therefore, multiplying 157 million tuples/s by 128 bits/tuple yields 20,096 Mbps. Thus, the peak throughput can be estimated at 20 Gbps. As for latency, recall that the latency of the implemented queries is equal to 4 cycles, and the clock period is 6.37 ns if we assume a clock rate of 157 MHz. Hence, multiplying 4 by 6.37 ns yields 25.48 ns. These data lead us to the conclusion that the proposed approach can accomplish both high throughput (over 150

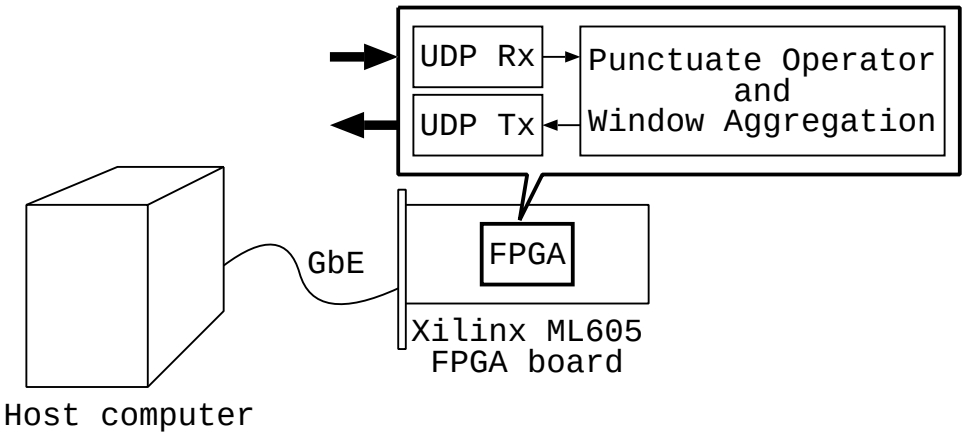



Figure 3.5.4: Overview of the Experimental System.

million tuples per second) and low latency (the order of a few tens of nanoseconds) which are essential for stream processing systems to handle a huge volume of data for real-time applications.

### 3.5.2 Experimental Measurement

An overview of the experimental system is depicted in Fig. 3.5.4. The experimental system consists of a Xilinx ML605 FPGA board and a host computer which are directly connected by a dedicated Gigabit Ethernet cable (indicated as "GbE" in Fig. 3.5.4). To simulate a disordered input stream, we implement a synthetic data generator to produce an input stream with bounded disorder. The schema of the input stream is defined as follows: $\mathit{Trades} = \langle \mathit{Symbol} : \text{char}[4], \mathit{Price} : \text{int}, \mathit{Volume} : \text{int}, \mathit{Time} : \text{int} \rangle$. The *Symbol* attribute contains either a constant string "UBSN" or other randomly generated strings of 4 characters (4 bytes). The *Price* and *Volume* attributes contain uniformly distributed random integers from the interval 1–100 and 1–1000, respectively. The *Time* attribute contains sequential numbers starting from 0 with an incremental interval of 1. The data generator on the host computer first generates 147,200 input tuples in non-decreasing order with respect to their *Time* attribute. After that, the positions of the tuples are randomized in such a way that no tuples are to be late or out-of-order more than 60 seconds (*i.e.,* a predefined *SLACK* value) in the stream.

We measured the number of clock cycles elapsed from when the first tuple arrived at the UDP Rx module until the last result was transferred from the UDP Tx module. For each configuration (*i.e.,* $\mathit{RANGE} = 10 \sim 60$ minutes), we calculated the maximum throughput achieved by the experimental system, using the measured values. It should be noted that all results generated by the query circuit have been verified by the host computer. This has been confirmed by comparing expected results with those sent from the UDP Tx module. In our experiments, we obtained exactly the same results as those

expected. This means that the proposed implementation can properly handle out-of-order tuples.

Results of the experiments show that the proposed implementation achieves an effective throughput up to around 760 Mbps, which is the upper bound of the available bandwidth that the network interface (*i.e.,* the UDP Rx module) could handle. This is equivalent to nearly 6 million tuples per second, which means that the proposed implementation can process significantly high tuple rates at wire speed. Furthermore, we have also conducted experiments on other aggregation functions, such as SUM, MIN, and MAX instead of COUNT, and obtained almost the same performance as Query $Q_2$ (*i.e.,* around 760 Mbps and nearly 6 million tuples/s). In other words, the proposed design provides the same constant performance regardless of the choice of the aggregate function, which is another favorable property of the proposed design.

# Chapter 4

# Scalable Implementation of Sliding-window Aggregate Operator

## 4.1 Abstract

This chapter presents an efficient and scalable implementation of an FPGA-based accelerator for sliding-window aggregates over disordered data streams. With an increasing number of overlapping sliding-windows, the window aggregates have a serious scalability issue, especially when it comes to implementing them in parallel processing hardware (*e.g.,* FPGAs). To address the issue, we propose a resource-efficient, scalable, and order-agnostic hardware design and its implementation by examining and integrating two key concepts, called Window-ID and Pane, which are originally proposed for software implementation, respectively. Evaluation results show that the proposed implementation scales well compared to the previous FPGA implementation in terms of both resource consumption and performance. The proposed design is fully pipelined and our implementation can process out-of-order data items, or tuples, at wire speed up to 200 million tuples per second.

## 4.2 Scalability Issue

### 4.2.1 Glacier

One of the most important design issue is the scalability in terms of both resource consumption and performance. Glacier relies on simultaneous evaluations of overlapping sliding-windows by instantiating a number of aggregation modules on an FPGA. Given a sliding-window aggregate query, the number of the aggregation modules ($N_{\text{Glacier}}$) required to be instantiated by Glacier is calculated based on two parameters: *RANGE* and *SLIDE* (see the following Equation 4.2.1).

$$N_{\text{Glacier}} = \left\lceil \frac{RANGE}{SLIDE} \right\rceil + 1 \tag{4.2.1}$$

Although this approach may be considered as a possible solution for a relatively small $\frac{RANGE}{SLIDE}$ ratio (*e.g.,* a few tens of the aggregation modules), the approach suffers from the scalability issues for a large $\frac{RANGE}{SLIDE}$ ratio (*e.g.,* several hundreds or thousands of the aggregation modules). Due to the replication strategy of the aggregation modules for overlapping sliding-windows, the number of replicas linearly increases with increasing $\frac{RANGE}{SLIDE}$ ratio. As a result, a large number of aggregation circuits should be instantiated on an FPGA to implement only a single query. This leads to serious scalability problems especially for large *RANGE* and/or small *SLIDE* values. In fact, even if a small $\frac{RANGE}{SLIDE}$ ratio is considered, the replication strategy discussed above leads to extremely inefficient resource utilization.

### 4.2.2 WID-based Implementation

In order to implement WID-based approach on an FPGA, we need an upper bound for the required hardware resources. For the purpose of determining the upper bound, we introduce a new parameter, called *slack* [2], in Chapter 3. Slack defines an upper bound on the degree of disorder and any tuple arriving after its corresponding period is discarded. It is stated in [2] that some aggregate operators, such as COUNT, SUM, AVERAGE, MIN, and MAX, can simply delay closing windows according to the slack specification. This approach enables us to handle disordered streams appropriately for sliding-window aggregation.

The WID-based implementation (presented in Chapter 3) basically relies on punctuations to handle disorder, and the slack parameter is used to calculate the number of the window-aggregation modules required to be instantiated. Based on Equation 3.4.1 and Equation 3.4.2 from Chapter 3, the minimum number of the aggregation modules ($N_{\text{WID}}$) is determined by using window parameters (*i.e., RANGE* and *SLIDE*) and a slack specification (see the following Equation 4.2.2).

$$N_{\text{WID}} = \left\lceil \frac{RANGE + SLACK}{SLIDE} \right\rceil \tag{4.2.2}$$

Equation 4.2.2 suggests that the required number of the aggregation modules ($N_{\text{WID}}$) linearly increase with increasing $\frac{RANGE}{SLIDE}$ ratio, assuming a constant *SLACK* value. Thus, the WID-based implementation also suffers from the same scalability problem as Glacier. This is because both Glacier and the WID-based approach rely on simultaneous evaluations of overlapping sliding-windows by simply replicating

the window-aggregation modules. Since this approach causes the scalability issues in terms of the maximum clock frequency and the hardware resource usage, it is crucial to design and implement a scalable hardware accelerator for sliding-window aggregate operator that can handle large $\frac{RANGE}{SLIDE}$ ratios. In this chapter, we adopt a two-step aggregation method using panes [19] and address the scalability problem of the previous implementations even if a large $\frac{RANGE}{SLIDE}$ ratio is considered.

## 4.3 Design Concept

### 4.3.1 Sliding Windows and Panes

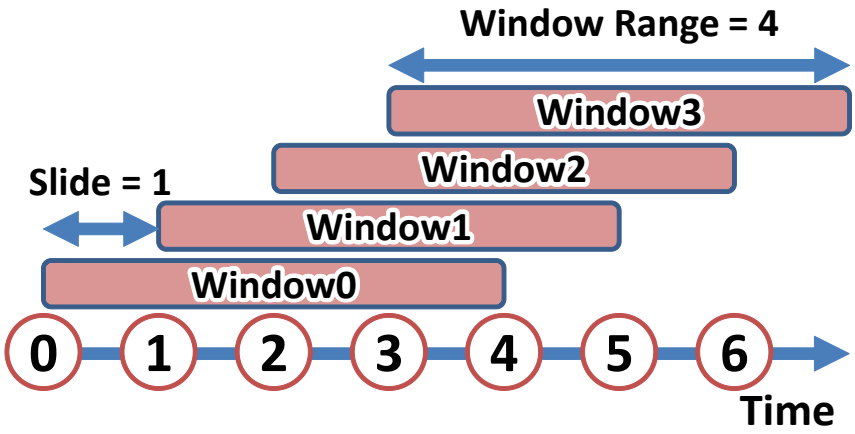
In this chapter, we first divide each sliding-window into disjoint sub-windows, called panes, instead of replicating the window-aggregation modules for overlapping windows. For example, Fig. 4.3.1a illustrates overlapping sliding-windows (only the first four windows) for Query $Q_1$ from Chapter 2. Recall from Query $Q_1$ that the bid stream is divided into overlapping 4-minute windows, each of which starts every 1 minute. Accordingly, in Fig. 4.3.1a, all windows have $RANGE = 4$ and $SLIDE = 1$, respectively.

How we divide these four sliding-windows into panes is illustrated in Fig. 4.3.1b. It is stated in [19] that *RANGE* and *SLIDE* of panes equal to the same value (*i.e.,* $RANGE_{\text{Pane}} = SLIDE_{\text{Pane}}$) and, given a sliding-window aggregate query, they are calculated as the greatest common divisor (GCD) of the *RANGE* and *SLIDE* of the original query. Since the original query (*i.e.,* Query $Q_1$) has $RANGE = 4$ and $SLIDE = 1$, we obtain $RANGE_{\text{Pane}} = SLIDE_{\text{Pane}} = \text{GCD}(4, 1) = 1$ minute. Thus, the number of panes per window is $RANGE_{Q_1}/RANGE_{\text{Pane}} = 4$. This can be easily noticed that each 4-minute window is composed of four consecutive panes as shown in Fig. 4.3.1b.

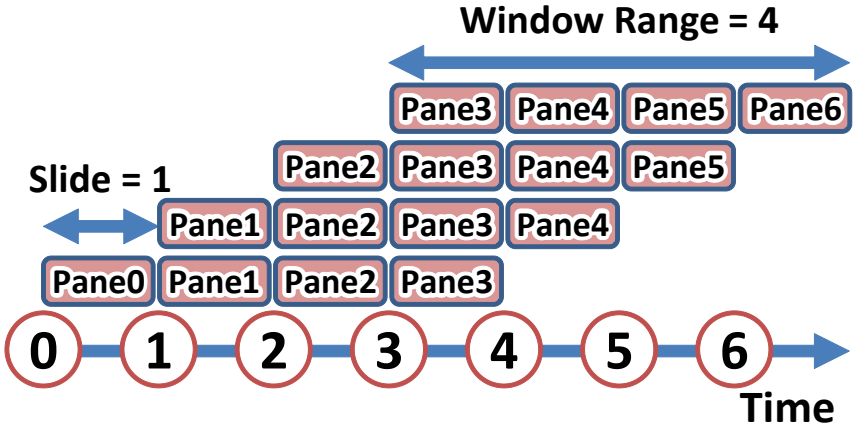
### 4.3.2 Two-Step Aggregation: PLQ and WLQ

In addition to dividing each sliding-window into multiple panes, the original aggregate query is decomposed into two sub-queries: a *pane-level sub-query* (PLQ) and a *window-level sub-query* (WLQ) [19]. We adopt this two-step aggregation approach and our hardware implementation of sliding-window aggregation is based on these two sub-queries. It should be also mentioned that Li *et al.* [19] use the term *pane-aggregates* and *window-aggregates* for the results of the PLQ and the WLQ, respectively. In the dissertation, the same terms are used to signify the difference between the results of the sub-queries (PLQ and WLQ).

In order to evaluate a sliding-window aggregate query by using panes, we need to determine window specifications and aggregate functions of the PLQ and WLQ, respectively. We can use the same windowing attribute (*i.e., WATTR*) as that of the original query for both sub-queries. In addition, we have already discussed how to determine the *RANGE* and *SLIDE* of panes in Section 4.3.1. As for the WLQ,




(a) Overlapping Sliding-Windows for Query $Q_1$ cited from Chapter 2 (*RANGE* = 4 minutes and *SLIDE* = 1 minute).




(b) Each sliding window of Query $Q_1$ is divided into four non-overlapping sub-windows (*i.e.,* panes), each of which has *RANGE* = 1 minute and *SLIDE* = 1 minute, respectively.

Figure 4.3.1: Relationship between overlapping sliding-windows and non-overlapping panes.

we use the same *RANGE* and *SLIDE* values as those of the original query.

Li *et al.* point out that aggregate functions of the two sub-queries depend on the aggregate function of the original query [19]. In the dissertation, we focus on the same aggregate functions as those of Glacier to implement sliding-window aggregate queries on an FPGA. In fact, AVERAGE function can be obtained with the combination of two aggregate values: SUM and COUNT, by simply dividing SUM by COUNT for each window. Therefore, we should consider the remaining four distributive aggregate functions (*i.e.,* COUNT, SUM, MIN, and MAX). Table 4.1 summarizes the relation between the aggregate function of the original query and the corresponding aggregates of the PLQ and WLQ. It turns out that, except for COUNT, we use the same aggregate function as the original query for both of the PLQ

Table 4.1: Relation between Original Query, PLQ, and WLQ.

| | | Aggregate Functions | | | |
|---|---|---|---|---|---|
| Sliding-window Aggregate Query | : | COUNT | SUM | MIN | MAX |
| ↓ | | ↓ | ↓ | ↓ | ↓ |
| Pane-level Sub-query (PLQ) | : | COUNT | SUM | MIN | MAX |
| + | | + | + | + | + |
| Window-level Sub-query (WLQ) | : | SUM | SUM | MIN | MAX |

```
SELECT max(bid-price) as p-max, timestamp
  FROM bids [RANGE 1 minute
             SLIDE 1 minute
             WATTR timestamp]
```

Figure 4.3.2: $Q_3$: "Find the maximum ***bid-price*** as ***p-max*** for the past ***1 minute*** and update the result every ***1 minute***."

and WLQ. When the original query is COUNT, the PLQ is also COUNT and the WLQ should be SUM, respectively.

**Example of Pane-level Sub-query (PLQ)**

It is stated in [19] that PLQ is a simple *tumbling-window* aggregation, which can be regarded as a special case of a sliding-window aggregate query whose window size (*i.e., RANGE*) is equal to the hop size (*i.e., SLIDE*). For each non-overlapping sub-window (*i.e.,* pane), the PLQ calculates an aggregate value, which is an intermediate result for the original sliding-window aggregate query. For example, Fig. 4.3.2 shows the PLQ (Query $Q_3$) of the original sliding-window aggregate query (*i.e.,* Query $Q_1$ from Chapter 2). In Query $Q_3$, the bid stream is divided into non-overlapping 1-minute windows starting every 1 minute. For each window, the maximum value of bid-price is calculated and Query $Q_3$ generates pane-aggregates with schema ⟨*p-max, timestamp*⟩ where the timestamp attribute indicates the end of each pane.

```
SELECT max(p-max) as w-max, timestamp
  FROM panes [RANGE 4 minutes
              SLIDE 1 minute
              WATTR timestamp]
```

Figure 4.3.3: $Q_4$: "Find the maximum ***p-max*** value for the past ***4 minutes*** and update the result every ***1 minute***."

**Example of Window-level Sub-query (WLQ)**

The other sub-query, WLQ, is a sliding-window query over the intermediate results of the PLQ and produces the final result for each window. For example, Fig. 4.3.3 shows the WLQ (Query $Q_4$) of the original sliding-window aggregate query (*i.e.,* Query $Q_1$ from Chapter 2). Query $Q_4$ accepts the pane-aggregates as its input and runs over the output stream of Query $Q_3$ (*i.e.,* the PLQ). In the WLQ, each pane (except for the first three panes) contributes four windows. For example, as shown in Fig. 4.3.1b, Pane3 contributes to Window0 through Window3. Similarly, Pane4 contributes to Window1 through Window4 and so forth. For each window, the WLQ computes the max of p-maxes of the last four panes and generates the window-aggregate with schema ⟨*w-max, timestamp*⟩ where the timestamp indicates the end of the window.

### 4.3.3 Hardware Cost Model

**Number of the Aggregation Modules**

The hardware design of PLQ is based on the WID-based approach discussed in Section 4.2.2. The main difference, however, is that the *RANGE* of the PLQ is always equal to its *SLIDE* value. As a result, we can simplify Equation 4.2.2 to calculate the number of the PLQ aggregate modules ($N_{\text{PLQ}}$) required to be instantiated (see the following Equation 4.3.1).

$$N_{\text{PLQ}} = \left\lceil \frac{RANGE_{\text{Pane}} + SLACK}{SLIDE_{\text{Pane}}} \right\rceil \tag{4.3.1}$$

By substituting $RANGE_{\text{Pane}} = SLIDE_{\text{Pane}}$, we obtain Equation 4.3.2.

$$N_{\text{PLQ}} = \left\lceil \frac{SLACK}{SLIDE_{\text{Pane}}} \right\rceil + 1 \tag{4.3.2}$$

The hardware design of WLQ is based on a sequential evaluation of the pane-aggregates. That is to say, given a sufficient buffer space for the pane-aggregates, we merely require to instantiate a single aggregation module to implement the WLQ. Therefore, the number of the WLQ aggregate module ($N_{\text{WLQ}}$) is always equal to one (*i.e.,* $N_{\text{WLQ}} = 1$). Finally, by adding $N_{\text{PLQ}}$ and $N_{\text{WLQ}}$, one obtains the total number of the aggregation modules ($N_{\text{Total}}$) required to implement the proposed design (see the following Equation 4.3.3).

$$N_{\text{Total}} = N_{\text{PLQ}} + N_{\text{WLQ}} = \left\lceil \frac{SLACK}{SLIDE_{\text{Pane}}} \right\rceil + 2 \tag{4.3.3}$$

Equation 4.3.3 suggests that the number of the total aggregation modules of the proposed pane-based hardware design is not affected by $\frac{RANGE}{SLIDE}$ ratio. In other words, contrary to Glacier (Equation 4.2.1) and WID-based implementation (Equation 4.2.2), the proposed design can handle large $\frac{RANGE}{SLIDE}$ ratios on the order of, say, hundreds or even thousands.

It should be also mentioned that there is a trade-off between accuracy and latency. Recall from Section 4.2.2 that *SLACK* defines an upper bound on the degree of disorder in order to wait for late tuples to arrive before finishing aggregate calculations. It is however stated in [2] that, given the real-time requirements of many stream applications, it is essential that it be possible to "time out" aggregate computations, even at the expense of accuracy. Since larger *SLACK* values result in longer latencies, we should restrict ourselves to a relatively small *SLACK* value. For example, assuming a slack value of 60 seconds for Query $Q_1$ (*i.e.,* exactly the same duration as *SLIDE* of Query $Q_1$), one obtains $N_{\text{PLQ}} = 2$ and $N_{\text{WLQ}} = 1$, which yield $N_{\text{Total}} = 3$. This means that the required number of the aggregation modules (*i.e.,* $N_{\text{Total}}$) remains constant with increasing $\frac{RANGE}{SLIDE}$ ratio. Thus, the proposed pane-based hardware design does not suffer from the scalability problems observed in Glacier and the WID-based implementation, both of which rely on the simple replication of a large number of the aggregation modules.

#### Pane-Buffer

In addition to the PLQ and WLQ, we also need to consider the design of *pane-buffer* when it comes to implementing the two-step aggregation on an FPGA. The pane-buffer is a cyclic first-in first-out (FIFO) buffer with support for sequential read access. As its name suggests, it stores the intermediate results of the PLQ (*i.e.,* pane-aggregates). It should be also noted that, by using panes, a sliding-window aggregate query can be evaluated with constant buffer space in dependent of the number of input tuples. Given a

set of window specifications (*i.e., RANGE*, *SLIDE*, and *WATTR*), we can determine the size of the pane-buffer ($S_{\text{buffer}}$) in terms of the number of pane-aggregates. In fact, the required buffer space is equal to the number of panes per window; therefore, the following Equation 4.3.4 gives us the size of the pane-buffer.

$$S_{\text{buffer}} = \frac{RANGE}{\text{GCD}(RANGE, SLIDE)} \tag{4.3.4}$$

Given a set of window specification, we can easily calculate $S_{\text{buffer}}$ as a constant value. With the constant bound on the size of the pane-buffer, we can efficiently implement the buffer using on-chip Block RAMs (BRAMs) on an FPGA. This is a significant difference between the approach adopted in this work and that of Glacier. While Glacier and the WID-based implementation (presented in Chapter 3) are not able to utilize BRAMs and only use the limited logic resources on an FPGA, the proposed pane-based design balances logic and BRAM utilization. This results in a considerable area reduction and a higher maximum frequency.

## 4.4 Implementation Details

In this Section, we focus on the same financial trading application as that of Chapter 3 (described in Section 3.3). In particular, Query $Q_2$ is considered as a target query to describe an implementation of the pane-based hardware design. The details of the implementation of Query $Q_2$ are provided in the following subsections.

### 4.4.1 Hardware Execution Plan

An overview of a hardware execution plan for Query $Q_2$ is illustrated in Fig. 4.4.1. In this Section, we adopt the same wiring interface as that of Chapter 3 to represent data flow (recall from Fig. 3.4.1 in Chapter 3). In particular, the gray-shaded boxes in Fig. 4.4.1 represent flip-flop registers which can be regarded as pipeline registers. As indicated in Fig. 4.4.1, Query $Q_2$ is implemented as a synchronous 5-stage pipeline.

The pipeline stages share a common clock signal and they are inserted between each stage as shown in Fig. 4.4.1. These registers buffer intermediate results at the end of each stage and the successive stages can use the result of the previous stage. The arrows in Fig. 4.4.1 indicate the connections between the pipeline stages. The first three stages correspond to pane-level sub-query (PLQ) and the remaining two stages are related to window-level sub-query (WLQ). The implementation details of PLQ and WLQ are discussed in Section 4.4.2 and Section 4.4.3, respectively.

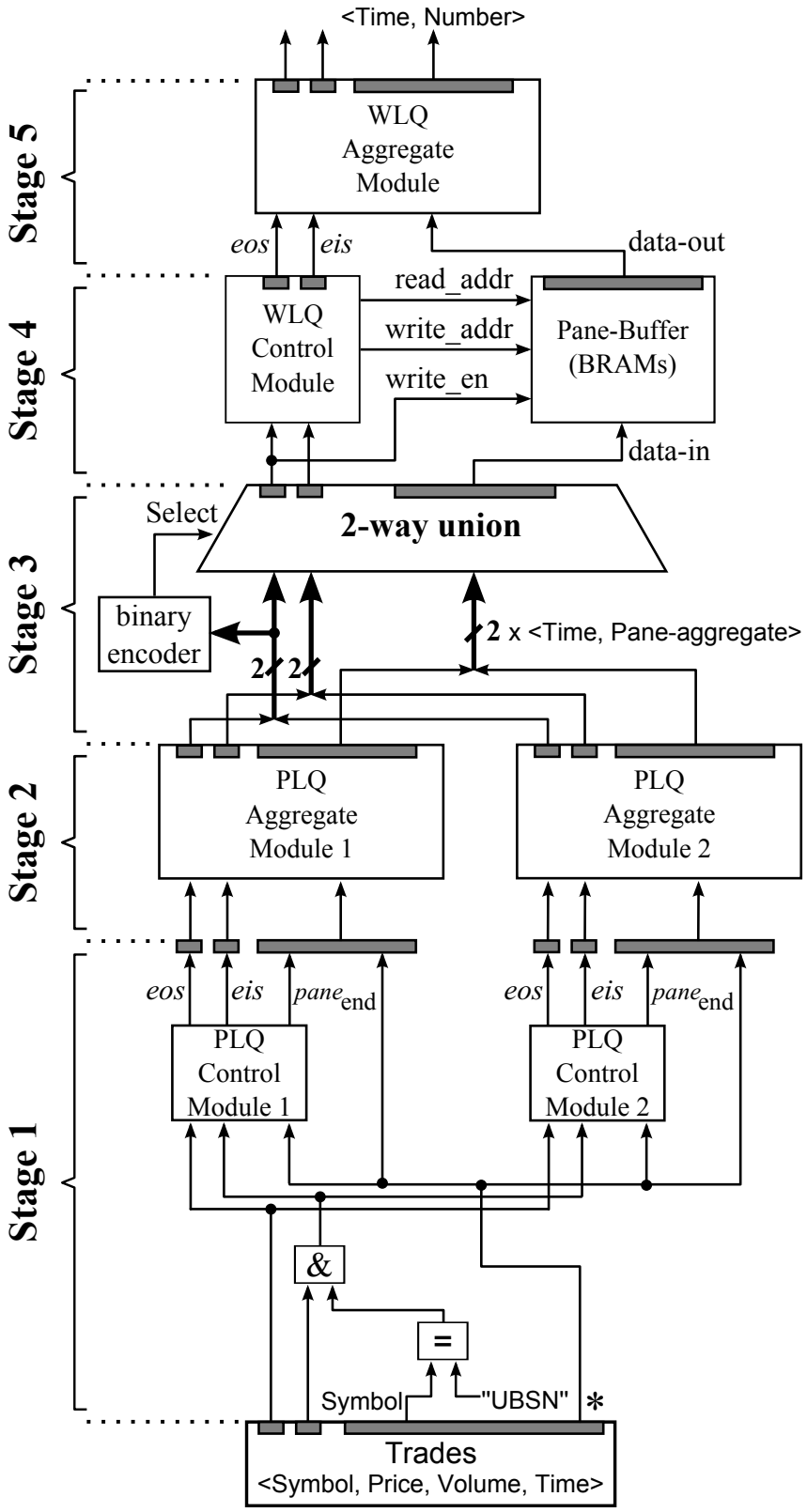



Figure 4.4.1: Hardware execution plan for Query $Q_2$.

### 4.4.2 Implementation of Pane-Level Sub-Query (PLQ)

The pane-level sub-query (*i.e.,* PLQ) of Query $Q_2$ is implemented as a 3-stage pipeline and the first three stages of Fig. 4.4.1 (*i.e.,* **Stage 1**, **Stage 2**, and **Stage 3**) show its data flow. In **Stage 1**, *Symbol* field of the data bus is compared to a constant string, "UBSN", which is specified in the WHERE expression of Query $Q_2$ (indicated as [=] in Fig. 4.4.1). At the same time, a logical AND gate (indicated as [&] in Fig. 4.4.1) computes whether an input tuple is valid for the query, using the result of the comparison. If the tuple should be discarded (*i.e.,* not satisfy the WHERE condition), the *data valid* flag is negated (*i.e.,* set to logic "0") for the next PLQ control modules. The PLQ control modules correspond to windowing operators that provide sliding-window functionality. Notice that each PLQ control module of **Stage 1** is paired with a PLQ aggregate module of **Stage 2** as shown in Fig. 4.4.1.

Each PLQ control module maintains pane states and provides two control signals, *eis* and *eos*, to the next stage of the pipeline (*i.e.,* **Stage 2**). The *eis* stands for *enable input stream* and it indicates whether or not data on the *data fields* should be considered as a valid tuple for the current pane. The other signal, *eos*, stands for *end of stream* and it indicates whether an input stream reaches the end of the current pane. The PLQ control module is responsible for its own states by updating its internal registers called $pane_{\text{begin}}$ and $pane_{\text{end}}$. These registers represent the beginning and the end of the current pane, respectively. The control module uses $pane_{\text{begin}}$ and $pane_{\text{end}}$ registers to generate the two control signals, *eis* and *eos*. Details about how to maintain these registers and to generate the control signals are provided in Algorithm 3 and Algorithm 4, respectively.

Algorithm 3 describes how to initialize and update the two registers, $pane_{\text{begin}}$ and $pane_{\text{end}}$. Since the windowing attribute (*i.e., WATTR*) of Query $Q_2$ is defined as *TIME*, $WATTR_{\text{start}}$ is equivalent to $TIME_{\text{start}}$ which indicates the start time of the execution of the query. Initialization or update operation described in Algorithm 3 can be completed in one clock cycle. All of the PLQ control modules concurrently perform the same operation on each cycle in a synchronous manner.

Algorithm 4 describes how to generate the two control signals, *eis* and *eos*. It is important to emphasize that the implementation of *eis* and *eos* signals is fully asynchronous. As shown in Fig. 4.4.1, *eis* and *eos* signals are connected to the data valid and punctuation flag registers, respectively. This means that these pipeline registers can be updated within the same clock cycle as soon as *eis* and *eos* signals are changed. In other words, all of the operations performed in **Stage 1** can be completed in a single clock cycle.

The next step is **Stage 2** of the pipeline which corresponds to aggregate operators for the PLQ. In **Stage 2**, two PLQ aggregate modules are instantiated as shown in Fig. 4.4.1. A more detailed block diagram of a single PLQ aggregate module is depicted in Fig. 4.4.2. The PLQ aggregate module includes four aggregate operators as shown in Fig. 4.4.2. It is stated in [25] that aggregate functions such as

**Algorithm 3** Maintain pane states for PLQ

**State Registers:**

$pane_{\text{begin}}(i)$: the beginning of the $i$-th pane instance

$pane_{\text{end}}(i)$: the end of the $i$-th pane instance

**Initialization:**

**for all** $i$ such that $1 \leq i \leq N_{\text{PLQ}}$ **do**

$pane_{\text{begin}}(i) \leftarrow WATTR_{\text{start}} + (i-1) \times SLIDE_{\text{PLQ}}$

$pane_{\text{end}}(i) \leftarrow WATTR_{\text{start}} + i \times SLIDE_{\text{PLQ}}$

**end for**

**Synchronous Update:**

**for all** $i$ such that $1 \leq i \leq N_{\text{PLQ}}$ **do**

**for each** clock cycle **do**

**if** *punctuation flag* is asserted **and**

$WATTR \geq pane_{\text{end}}(i)$ **then**

$pane_{\text{begin}}(i) \leftarrow pane_{\text{begin}}(i) + N_{\text{PLQ}} \times SLIDE_{\text{PLQ}}$

$pane_{\text{end}}(i) \leftarrow pane_{\text{end}}(i) + N_{\text{PLQ}} \times SLIDE_{\text{PLQ}}$

**end if**

**end for**

**end for**

COUNT, SUM, MIN, and MAX can be implemented in a straightforward fashion on an FPGA. Since the PLQ requires count($*$) function, the result of the COUNT operator is selected as the output value (indicated as the broken line in Fig. 4.4.2).

The aggregate operator incrementally computes aggregate value and only stores the current (partial) result of the aggregation. It requires two control signals (*i.e., eis* and *eos*) to maintain its aggregate value. Whenever *eis* is asserted, the aggregate operator accepts input tuple and records its contribution to the partial result. If *eis* is negated, the aggregate operator simply ignores the input data and waits for the next tuple to arrive. When *eos* is asserted, it means that the current pane is no longer active and the aggregate operator should reset its internal state. It is important to note that all operations performed in **Stage 2** can be completed in a single clock cycle just as in **Stage 1**.

We adopt a similar approach as that of the WID-based implementation (presented in Chapter 3) to implement a union operator, which is based on a multiplexer component. The main difference however is the required size of the multiplexer. In the WID-based implementation (Chapter 3), the size of the

**Algorithm 4** Generate asynchronous control signals for PLQ

**Asynchronous Signals:**

*eis*($i$): input enable signal for the $i$-th pane instance

*eos*($i$): output enable signal for the $i$-th pane instance

**Asynchronous Update:**

**for all** $i$ such that $1 \leq i \leq N_{\text{PLQ}}$ **asynchronously do**
  **if** *punctuation flag* is negated **then**
    negate *eos*($i$) signal
    **if** *data valid* is asserted **and**
    $pane_{\text{begin}}(i) \leq WATTR < pane_{\text{end}}(i)$ **then**
      assert *eis*($i$) signal
    **else**
      negate *eis*($i$) signal
    **end if**
  **else** {*punctuation flag* is asserted}
    negate *eis*($i$) signal
    **if** $WATTR \geq pane_{\text{end}}(i)$ **then**
      assert *eos*($i$) signal
    **else**
      negate *eos*($i$) signal
    **end if**
  **end if**
**end for**

multiplexer increases with increasing $\frac{RANGE}{SLIDE}$ ratio and hence leads to scalability problems. On the other hand, the pane-based approach is not affected by the $\frac{RANGE}{SLIDE}$ ratio; in other words, the proposed design requires a constant-size multiplexer. This is a significant difference between the proposed pane-based hardware design and the WID-based implementation (Chapter 3).

According to a select signal, the multiplexer component transfers the result of $i$-th PLQ aggregate module to the output registers of **Stage 3**. As illustrated in Fig. 4.4.1, the select signal is provided by a binary encoder component. It is stated in [25] that, from a data flow point of view, the task of an algebraic union operator is to merge the outputs of several source streams into a single output stream. As shown in Fig. 4.4.1, the 2-way union operator merges the outputs of two PLQ aggregate modules and generates a single result stream. It should be also mentioned that **Stage 3** requires only one clock cycle to complete

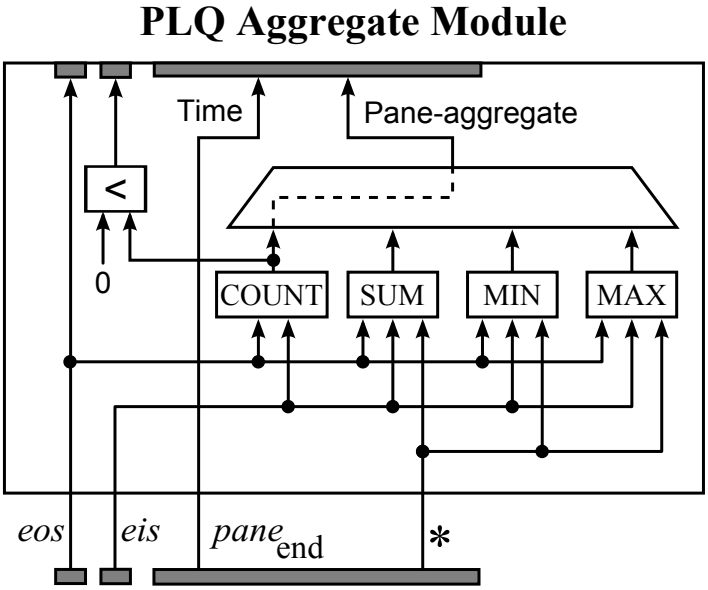



Figure 4.4.2: Block diagram of a PLQ aggregate module.

its operation.

### 4.4.3 Implementation of Window-Level Sub-Query (WLQ)

The window-level sub-query (*i.e.,* WLQ) of Query $Q_2$ is implemented as a 2-stage pipeline and the last two stages of Fig. 4.4.1 (*i.e.,* **Stage 4** and **Stage 5**) show its data flow. **Stage 4** includes a single WLQ control module and a pane-buffer. Similar to a PLQ control module, the WLQ control module maintains its internal states and provides two control signals, *eis* and *eos*, to the final stage of the pipeline (*i.e.,* **Stage 5**). In addition, the WLQ control module also provides read and write addresses to the pane-buffer.

Details about how to maintain the window states and to generate the control signals are provided in Algorithm 5 and Algorithm 6, respectively. Algorithm 5 describes how to initialize and update four internal registers: *wr_addr*, *rd_addr*, *rd_addr_prev*, and *pane_counter*. It should be mentioned that *wr_addr* and *rd_addr* registers are connected to the write_addr and read_addr ports of the pane-buffer (see **Stage 4** of Fig. 4.4.1), respectively. Algorithm 6 describes how to generate the two control signals, *eis* and *eos*, for the WLQ aggregate module.

As discussed in Section 4.3.3, the pane-buffer is a cyclic first-in first-out (FIFO) buffer and its implementation is based on on-chip Block RAMs (BRAMs). BRAMs support dual-ports and each port has its own data-in, data-out, and address bus. We use simple dual-port mode, that is to say, one port can only write and the other port can only read data. As illustrated in Fig. 4.4.1, the data field of **Stage 3** is connected to the data-in port (*i.e.,* write-only port) of the pane-buffer. Similarly, the data-out port (*i.e.,*

**Algorithm 5** Maintain window states for WLQ

**State Registers:**

*wr_addr*: write-address register of the pane-buffer

*rd_addr*: read-address register of the pane-buffer

*rd_addr_prev*: previous value of the read-address register

*pane_counter*: pane counts in the current window

**Initialization:**

*wr_addr* ← 1

*rd_addr* ← 0

*rd_addr_prev* ← 0

*pane_counter* ← 0

**Synchronous Update:**

**for each** clock cycle **do**

  *rd_addr_prev* ← *rd_addr*

  **if** *pane_counter* < PANES_PER_WINDOW **then**

    **if** *rd_addr* ≠ *wr_addr* **and** *rd_addr* + 1 ≠ *wr_addr* **then**

      *rd_addr* ← *rd_addr* + 1

      *pane_counter* ← *pane_counter* + 1

    **end if**

  **else**

    *rd_addr* ← *rd_addr* − PANES_PER_WINDOW + 2

    *pane_counter* ← 1

  **end if**

  **if** *punctuation flag* is asserted **then**

    *wr_addr* ← *wr_addr* + 1

  **end if**

**end for**

read-only port) of the pane-buffer is directly connected to the next stage (*i.e.,* **Stage 5**).

**Stage 4** requires a total of three clock cycles to complete its operation when the last pane-aggregate of each window arrives from the previous stage (*i.e.,* **Stage 3**). Specifically, when the punctuation flag is asserted, one clock cycle is required to update the *wr_addr* register. After that, another clock cycle is consumed to update the *rd_addr* and *pane_counter* registers. Finally, the third clock cycle is used to

**Algorithm 6** Generate asynchronous control signals for WLQ

**Asynchronous Signals:**

*eis*: input enable signal for the WLQ aggregate module

*eos*: output enable signal for the WLQ aggregate module

**Asynchronous Update:**

**if** *rd_addr* ≠ *rd_addr_prev* **then**

  assert *eis* signal

**else**

  negate *eis* signal

**end if**

**if** *pane_counter* < PANES_PER_WINDOW **then**

  negate *eos* signal

**else**

  assert *eos* signal

**end if**

retrieve data from the pane-buffer, which is implemented using BRAM primitives.

When it comes to the implementation of **Stage 5** of Fig. 4.4.1, the WLQ aggregate module is implemented almost in the same way as PLQ aggregate module (see Fig. 4.4.2). Hence, all operations performed in **Stage 5** can be completed in a single clock cycle. In Chapter 3, the performance characteristics of the implemented circuit is analyzed in terms of *latency* and *issue rate*. Recall from Chapter 3 that the issue rate is defined as the number of tuples that can be processed per clock cycle. The overall latency and the issue rate of the proposed pane-based implementation are 7 cycles and 1 tuple/cycle, respectively.

## 4.5 Evaluation

The pane-based hardware design is implemented on a Virtex®-6 FPGA (XC6VLX240T) included in the Xilinx ML605 Evaluation Kit [51] (*i.e.,* the same evaluation board as that of Chapter 3). The specification of the FPGA used to implement the design is given in Table 3.1 (Chapter 3). Xilinx ISE 14.4 is used as an FPGA development environment during the implementation process (*e.g.,* synthesis, map, and place & route).

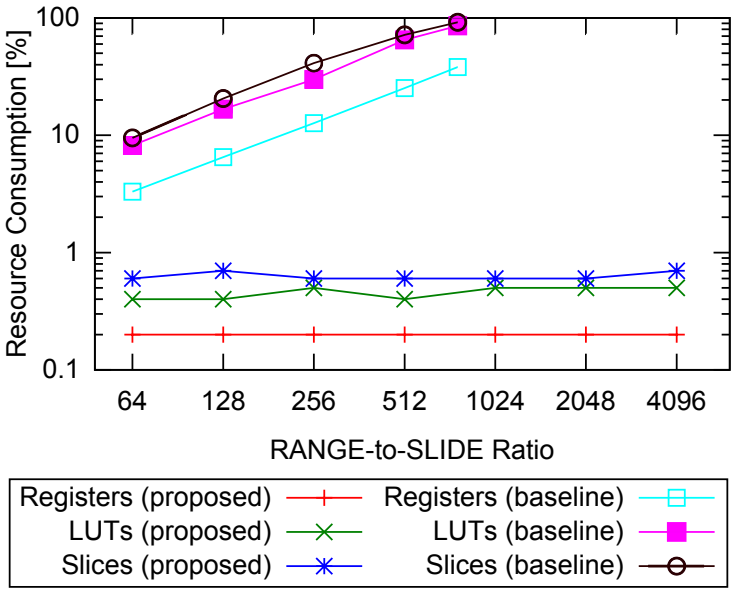



Figure 4.5.1: Comparison of the overall resource consumption between the pane-based implementation (labeled "proposed") and the WID-based implementation presented in Chapter 3 (labeled "baseline").

### 4.5.1 Resource Utilization and Performance

In order to evaluate the scalability of the proposed approach, the pane-based hardware design (*i.e.*, Fig. 4.4.1) and the WID-based approach (presented in Charpter 3) are implemented for the same target query (Query $Q_2$) with a wide range of different window parameters. We have modified the *RANGE* parameter of the query and the $\frac{RANGE}{SLIDE}$ ratio is increased by multiples of 2, beginning with 64 up to 4096 (*i.e.,* a total of seven different configurations). The proposed implementation is synthesized with a timing constraint of 5 ns for each configuration, which yields the target clock frequency of 200 MHz.

#### Resource Utilization

The comparison of the overall resource consumption is shown in Fig. 4.5.1. The x-axis represents $\frac{RANGE}{SLIDE}$ ratio of the time-based sliding window. The y-axis indicates the resource consumption (in log scale) in terms of percentages of the total available resources on the target FPGA device. In Fig. 4.5.1, the proposed implementation and the WID-based implementation (Chapter 3) are labeled "proposed" and "baseline", respectively.

Results of Fig. 4.5.1 suggest that the proposed implementation achieves significant area reduction compared to the baseline. For instance, when $\frac{RANGE}{SLIDE} = 512$, the baseline consumes over 70% of the available slices on the target FPGA whereas the proposed implementation only requires 0.6% of the

Table 4.2: Block RAM (BRAM) Utilization.

| | RANGE-to-SLIDE ratio (*i.e.*, $\frac{RANGE}{SLIDE}$) | | | | | | |
|---|---|---|---|---|---|---|---|
| | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 |
| Number of BRAMs : | 1 | 1 | 1 | 2 | 4 | 8 | 17 |
| Overall BRAM usage : | 0.2% | 0.2% | 0.2% | 0.5% | 1.0% | 1.9% | 4.1% |

available slices, by using only two BRAMs on the same FPGA (*i.e.,* about 0.5% of the total available BRAMs). The required number of BRAMs for each configuration is given in Table 4.2.

Moreover, as indicated in Fig. 4.5.1, all three graphs of the baseline almost linearly increases with increasing $\frac{RANGE}{SLIDE}$ ratio. As a result, when $\frac{RANGE}{SLIDE} = 768$, the baseline utilizes over 90% of the available slices and if we increase the ratio (*i.e.*, $\frac{RANGE}{SLIDE} \geq 1024$), the query cannot be implemented on the target device due to finite area of the FPGA. On the other hand, however, the proposed implementation does not suffer from this limitation as shown in Fig. 4.5.1. All three graphs of the proposed implementation are almost constant and do not increase with increasing $\frac{RANGE}{SLIDE}$ ratio; therefore, the proposed design provides better scalability compared to the baseline.

It is stated in [19] that one of the most important benefits of using panes is considerable reduction of required buffer space. Although this is true for software-based approach, it is not the case for hardware-based implementation. This is the main difference between the software- and hardware-based approaches. In fact, the baseline does not require any on-chip memory resource (i.e., BRAMs) as buffer space, whereas the proposed implementation requires a few BRAMs to implement pane-buffers (see Table 4.2). The results, however, indicate that the proposed implementation does not impose significant overhead on the BRAM usage. For example, when $\frac{RANGE}{SLIDE} = 4096$, the overhead is only 4.1 percent of the total available BRAMs on the target FPGA device as shown in Table 4.2. This means that when $SLIDE = 60$ seconds, the proposed implementation can handle large window sizes up to $RANGE = 60 \times 4096$ seconds with a very reasonable hardware cost.

**Performance Evaluation**

The comparison of the maximum clock frequency is shown in Fig. 4.5.2. The x-axis and the y-axis represent $\frac{RANGE}{SLIDE}$ ratio and the clock frequency, respectively. The clock frequency is obtained from post-place & route static timing report, which is provided by Xilinx's Timing Analyzer tool. As shown in Fig. 4.5.2, the clock frequency of the baseline drops sharply with increasing $\frac{RANGE}{SLIDE}$ ratio. In contrast, the

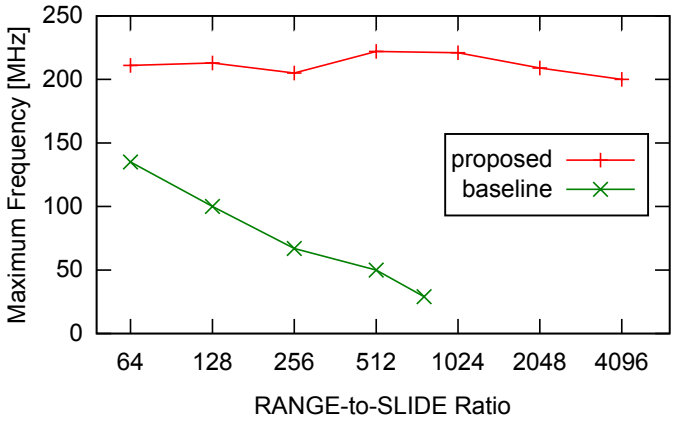



Figure 4.5.2: Comparison of the maximum clock frequency between the pane-based implementation (labeled "proposed") and the WID-based implementation presented in Chapter 3 (labeled "baseline").

clock frequency of the proposed design remains largely unaffected by the $\frac{RANGE}{SLIDE}$ ratio. Specifically, the proposed implementation meets the timing constraint of 5 ns and maintains the target clock frequency of 200 MHz for each configuration. The fact that the proposed design can still sustain high frequencies is a good indication for the scalability. For instance, when $\frac{RANGE}{SLIDE} = 512$, the baseline can operate at only up to 50 MHz on the target FPGA whereas the proposed implementation can operate at over 200 MHz. This means that the proposed approach achieves over 4× performance improvement (*i.e.,* higher clock frequency means better performance, especially in terms of throughput) with significantly reduced hardware cost.

Since the issue rate of the proposed design is equal to 1 tuple/cycle, the proposed implementation can process 200 million tuples per second. As for latency, recall that the latency of the implemented queries is equal to 7 cycles, and the clock period is 5 ns if we assume a clock rate of 200 MHz. Hence, multiplying 7 by 5 ns yields 35 ns. These data lead us to the conclusion that the proposed design is scalable against $\frac{RANGE}{SLIDE}$ ratio and can accomplish both high throughput (over 200 million tuples per second) and low latency (the order of a few tens of nanoseconds).

### 4.5.2 Experimental Measurement

It is stated in [25] that a key aspect of using an FPGA for data stream processing is its flexibility that enables us to insert custom hardware logic into an existing data path. For example, the proposed sliding-window aggregate circuit can be directly connected to the physical network interface. In order to verify the effectiveness of the proposed method, we implement the same experimental system as that of Chapter 3.

Our experiments are based on a Xilinx ML605 FPGA board, which includes the Virtex-6 FPGA and a Gigabit Ethernet interface. An overview of the experimental system is depicted in Fig. 3.5.4 (Chapter 3). The experimental system consists of the ML605 FPGA board and a host computer, which are directly connected by a dedicated Gigabit Ethernet cable (indicated as "GbE" in Fig. 3.5.4). To simulate a disordered input stream, we use the same data generator as that of Chapter 3 to produce an input stream with bounded disorder. The schema of the input stream is defined as follows: $Trades = \langle Symbol : \text{char}[4], Price : \text{int}, Volume : \text{int}, Time : \text{int} \rangle$. The *Symbol* attribute contains either a constant string "UBSN" or other randomly generated strings of 4 characters (4 bytes). The *Price* and *Volume* attributes contain uniformly distributed random integers from the interval 1–100 and 1–1000, respectively. The *Time* attribute contains sequential numbers starting from 0 with an incremental interval of 1. The data generator on the host computer first generates 753,664 input tuples in non-decreasing order with respect to their *Time* attribute. After that, the positions of the tuples are randomized in such a way that no tuples are to be late or out-of-order more than 60 seconds (*i.e.,* a predefined *SLACK* value) in the stream.

We have measured the effective throughput of the proposed pane-based implementation on the ML605 FPGA board. Results of the experiments show that the proposed implementation achieves an effective throughput up to around 760 Mbps for different $\frac{RANGE}{SLIDE}$ ratios. This is the upper bound of the available bandwidth that the network interface (*i.e.,* the UDP Rx module [11]) could handle. This is equivalent to nearly 6 million tuples per second, which means that the proposed implementation can process significantly high tuple rates at wire speed, even if large $\frac{RANGE}{SLIDE}$ ratios are considered (*e.g.,* $\frac{RANGE}{SLIDE} \geq 1024$). Furthermore, we have also conducted experiments on other aggregation functions, such as SUM, MIN, and MAX instead of COUNT in a similar way to Chapter 3, and obtained almost the same performance as Query $Q_2$ (*i.e.,* around 760 Mbps and nearly 6 million tuples/s). In other words, the pane-based design also provides the same constant performance regardless of the choice of the aggregate function, which is another favorable property of the proposed design.

# Chapter 5

# Configurable Query Processing Hardware for Data Streams

## 5.1 Abstract

This Chapter presents Configurable Query Processing Hardware (CQPH), an FPGA-based accelerator for continuous query processing over data streams. CQPH is a highly-optimized and minimal-overhead execution engine designed to deliver real-time response for high-volume data streams. Unlike most of the other FPGA-based approaches, CQPH provides on-the-fly configurability for multiple queries with its own dynamic configuration mechanism. With a dedicated query compiler, SQL-like queries can be easily configured into CQPH at run time. CQPH supports continuous queries including selection, group-by operation and sliding-window aggregation with a large number of overlapping sliding-windows. As a proof of concept, a prototype of CQPH is implemented on an FPGA platform for a case study. Evaluation results indicate that a given query can be configured within just a few microseconds, and the prototype implementation of CQPH can process over 150 million tuples per second with a latency of less than a microsecond. Results also indicate that CQPH provides linear scalability to increase its flexibility (*i.e.,* on-the-fly configurability) without sacrificing performance (*i.e.,* maximum allowable clock speed).

## 5.2 Hardware Design Issue

A common limitation from which the most FPGA-based approaches suffer is that the existing approaches impose significant overhead on run-time query registration/modification. It is mentioned in [29] that while supporting query modification at run time is almost trivial for software-based techniques, they are highly uncommon for custom hardware-based approaches such as FPGAs. In fact, both the WID-based implementation (presented in Chapter 3) and the pane-based implementation (presented in Chapter 4) can

be characterized as a *static* FPGA-based query processor, fully tailored for a specific query. Therefore, it has a disadvantage when it comes to reconfiguring for a new query because the compilation steps (*e.g.,* synthesis and place-and-route) can take on the order of minutes or even up to hours to complete.

On the other hand, given the dynamic environment of data streams, queries can join and leave a streaming system at any time [15]. It is therefore imperative for a query processing accelerator to support on-the-fly configurability for easy adaptation to the dynamic environment. In order to overcome the limitation of the previous approaches, this Chapter proposes a fundamentally novel approach compared to those presented in Chapter 3 and Chapter 4. In particular, the main contributions of the Chapter are summarized as follows.

- Three configurable hardware modules are designed to execute sliding-window aggregate queries:
  1. selection module with efficient support for multiple selection conditions,
  2. group-by module based on a scalable systolic architecture, and
  3. window-aggregation module supporting a large number of overlapping sliding-windows.
- Two-phase configuration approach is adopted to provide on-the-fly configurability for given queries:
  1. a fully automated integration of the hardware modules to implement CQPH on FPGA, and
  2. run-time query configuration with a dedicated query compiler implemented for CQPH.
- The proposed approach is evaluated in terms of
  1. latency, throughput, and configuration time;
  2. resource utilization and maximum clock frequency with a case study; and
  3. performance measurement of a prototype system implemented on a Xilinx FPGA platform.

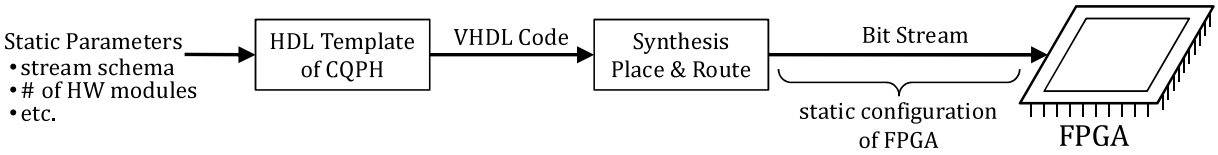
The proposed approach overcomes the limitations of the previous work such as [25, 26, 34, 35, 36], by offering a great degree of flexibility for on-the-fly query configuration. To the best of our knowledge, this is the first work that presents an FPGA-based query processor that can support run-time configuration of sliding-window aggregate queries with a large number of overlapping sliding-windows.

## 5.3 Design Concept

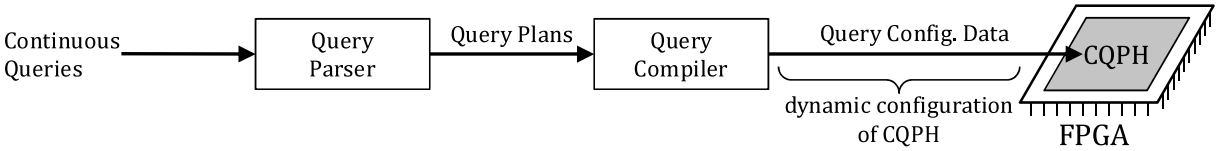
### 5.3.1 On-the-fly Query Configuration

#### Two-phase Configuration Approach

In this Chapter, we adopt two-phase configuration approach to support on-the-fly configuration of continuous queries, instead of implementing a static query processing hardware that is fully tailored for a




(a) Static Configuration: One-time-only configuration of an FPGA to implement CQPH by using a standard FPGA tool chain. The first phase (*i.e.,* static configuration) typically requires a relatively long period of time. In particular, the compilation steps (*i.e.,* synthesis and place-and-route) can take on the order of minutes or even up to hours to complete.




(b) Dynamic Configuration: On-the-fly configuration of continuous queries by updating internal registers of CQPH at run time. In contrast to the first phase, the second phase (*i.e.,* dynamic configuration) only requires a very short period of time (*e.g.,* in the order of microseconds).

Figure 5.3.1: Two-phase configuration: static configuration of FPGA (Fig. 5.3.1a) and dynamic configuration of CQPH (Fig. 5.3.1b).

specific query. The basic idea of the two-phase configuration approach is illustrated in Fig. 5.3.1. The proposed approach is based on *static* and *dynamic* configuration mechanisms.

**Static Configuration of FPGA**

The first phase is static configuration of FPGA (see Fig. 5.3.1a). CQPH is designed as a highly parameterized HDL model; therefore, static configuration parameters should be provided to generate an application-specific CQPH instance. Without any hardware design experience, users only need to provide a few static parameters according to their applications' need (see Table 5.1). In addition, advanced users can provide more detailed configuration parameters to make trade-offs between area and flexibility, by scaling the number of each configurable module included in CQPH template design.

The implementation of CQPH follows a normal FPGA design flow as shown in Fig. 5.3.1a. Given a set of static configuration parameters, VHDL description of CQPH is fed to a standard FPGA tool chain (*e.g.,* synthesis, place-and-route) to generate the actual low-level representation of the FPGA-specific circuit. After that, the generated circuit is used to program an FPGA, by downloading a bit-stream file into the FPGA. It should be noted that users are required to go through the static configuration process

Table 5.1: List of Statically-configured Parameters.

| |
|---|
| Stream schema (*i.e.,* Tuple Width & the Number of Attributes) |
| Pane Buffer Size (*i.e.,* Data Width & the Number of Entries) |
| # of Selection Predicate Modules (see Section 5.4.2 for details) |
| # of Boolean Expression Trees (see Section 5.4.2 for details) |
| # of Group-by Manager Modules (see Section 5.4.3 for details) |
| # of Aggregation Pipelines (see Section 5.4.4 for details) |
| # of Pipeline Stages of Union (see Section 5.4.5 for details) |

only once to implement CQPH prior to run-time execution of continuous queries.

**Dynamic Configuration of CQPH**

The second phase is dynamic configuration of CQPH (see Fig. 5.3.1b). Once CQPH is implemented on an FPGA through the static configuration process, it is now ready for CQPH to execute continuous queries. In order to support run-time configuration of continuous queries, we have implemented a dedicated parser/compiler, *CQPH-compiler*, which can compile continuous queries into *query configuration data* for CQPH. In the proposed design, query configuration data are divided into a set of *configuration tuples* which are then streamed into CQPH. Fig. 5.3.1b illustrates the compilation process of continuous queries to create query configuration data for CQPH.

It is mentioned in [29] that a common limitation of FPGA-based approaches relates to the inherent complexity of the design synthesis process, such as logic optimization and technology mapping, which leads to the drastic increase in synthesis time as applications grow in size. CQPH-compiler does not suffer from this limitation because the query compilation process of CQPH does not require a time-consuming synthesis process of FPGA. As a result, CQPH can provide a significant degree of flexibility for run-time query configuration.

CQPH template design consists of a number of configurable hardware modules. Fig. 5.3.2 illustrates a black-box view of a configurable hardware module and its wiring interface. Each hardware module has its own *bit flag field* and *data field* as a connection interface. Bit flag field consists of one-bit signals one of which is a configuration flag. In addition, data field represents $n$-bit-wide data which is regarded as a set of $n$ parallel wires. It should be noted that each hardware module adheres the same wiring interface to connect another module. For example, datum on the input data field is considered as a part of the query configuration data (*i.e.,* a configuration tuple) when the configuration flag of the previous module

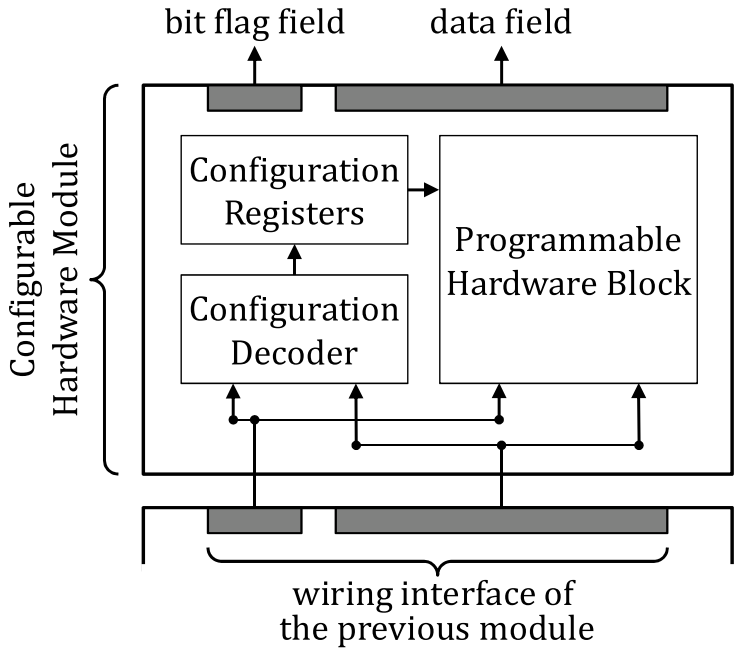



Figure 5.3.2: Black-box view of a configurable hardware module.

is asserted (*i.e.,* set to logic "1").

Each configuration tuple consists of two main parts: (i) configuration data field, and (ii) target ID field. As its name suggests, configuration data field contains query configuration data for a specific hardware module. Target ID field contains a unique identifier assigned to each configurable hardware module. Given a configuration tuple, each hardware module evaluates whether the target ID field matches its own ID number. In accordance with the result, configuration data is stored to configuration registers inside a configurable hardware module, or the configuration tuple is simply forwarded to the next module. It should be emphasized that each hardware module can be dynamically configured in just one clock cycle by a single configuration tuple. With the proposed approach, users can easily update or modify configuration registers of each hardware module to change the behavior of a programmable hardware block at run time.

### 5.3.2 Supported Capabilities of CQPH

The current prototype of CQPH can be configured to execute continuous queries that follow certain patterns. In particular, CQPH supports queries from simple filtering to window-based aggregation (see Fig. 5.3.3 and Fig. 5.3.4). As indicated in Query $Q_5$ and $Q_6$, the main focus of this work is on those queries that process a single data stream without joins. For those interested in the implementation of joins, we refer readers to *handshake join algorithm* [44] and our previous works [30, 31, 32, 33] for the

```
SELECT * FROM Stream WHERE <boolean expression>
```

Figure 5.3.3: $Q_5$: Template of selection-based filtering.

```
SELECT <windowing attribute>,
       <grouping attribute>,
       <aggregate function>
  FROM Stream [RANGE <window size>
               SLIDE <hop size>
               WATTR <windowing attribute>]
 WHERE <boolean expression> (optional)
 GROUP BY <grouping attribute> (optional)
```

Figure 5.3.4: $Q_6$: Template of window-based aggregation.

details of our implementation of the join algorithm on an FPGA.

Note that, in Query $Q_5$ and $Q_6$, any expression between '<' and '>' can be configured at run time (*i.e.,* dynamically configurable via configuration registers). This is a significant difference between CQPH and a static FPGA-based query processor. In fact, CQPH enables users to add, modify or remove continuous queries at negligible cost compared to previous approaches such as Glacier [25], the WID-based implementation (Chapter 3), and the pane-based implementation (Chapter 4).

It should be also emphasized that available resources (*i.e.,* configurable hardware modules) do not have to be separated into queries statically. For example, one can assign an arbitrary number of aggregation pipelines for a specific query at run time. The resource allocation of CQPH is somehow similar to those of skeleton automata [45, 46]. As stated in [45], this lets us make efficient use of resources. In our case, the same circuit can be utilized for either many queries each of which is assigned to single aggregation pipeline or fewer queries with multiple aggregation pipelines. In either case, the total number of configurable hardware modules provisioned in the CQPH limits the number of queries that can be processed simultaneously.

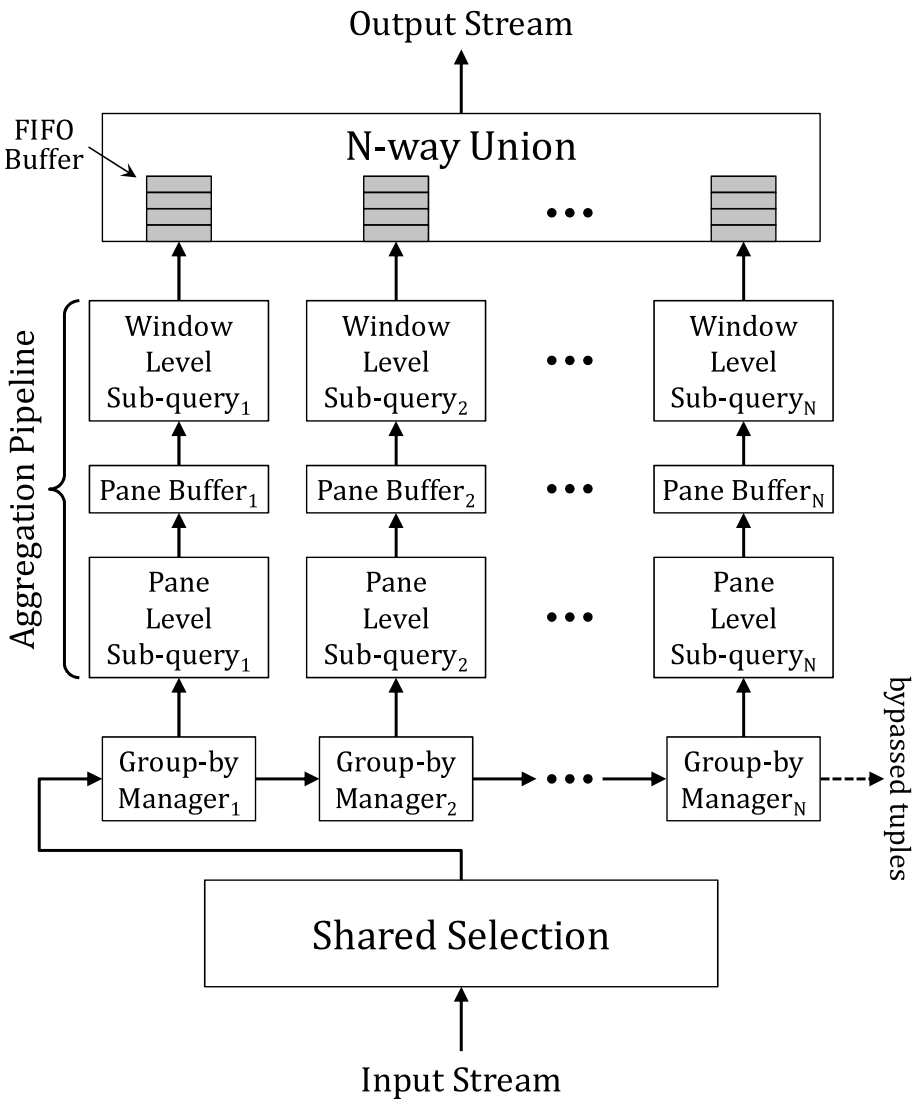



Figure 5.4.1: Overview of the data flow of CQPH Architecture.

## 5.4 CQPH Architecture

### 5.4.1 Overview of CQPH

In this section, we present the details of CQPH architecture. An overview of the CQPH architecture is illustrated in Fig. 5.4.1. CQPH contains a collection of configurable hardware modules each of which is designed for selection, group-by, or sliding-window aggregate operator. In the proposed design, we adopt push-based processing model with a fully-pipelined implementation of the configurable hardware modules. Arrows in Fig. 5.4.1 represent the direction of data flow between each module, and there is no loopback connections between any two modules.

It should be emphasized that CQPH architecture designed this way remains fully pipelined and oper-

```
SELECT key, SUM(value)
  FROM KeyValue [RANGE 1 minute
                 SLIDE 1 minute
                 WATTR time]
 WHERE value > 10
 GROUP BY key
```

Figure 5.4.2: $Q_7$: "Given an input stream *KeyValue* = ⟨*key*, *value*, *time*⟩, first filter out all the tuples that do not satisfy the condition in WHERE clause (*i.e.,* ***value*** > 10). After that calculate the sum of ***value*** for each different ***key*** for the past ***1 minute*** and update the result every ***1 minute***."

ates in a strict streaming fashion at wire-speed rate. This guarantees wire-speed performance and CQPH can accept one input tuple per clock cycle independent of the query workload. As a simple example, CQPH architecture is capable of processing queries like Query $Q_7$ (see Fig. 5.4.2) in a strict streaming fashion at wire-speed rate (for ease of presentation, this example assumes *RANGE* = *SLIDE* = 1 minute).

CQPH processes incoming tuples in a series of pipeline stages, which are designed for filtering, grouping, and aggregation operations as shown in Fig. 5.4.3. In Fig. 5.4.3, input tuples are simple key-value pairs with a timestamp attribute. The schema of the input stream is defined as follows: *KeyValue* = ⟨*key*, *value*, *time*⟩. The ***key*** and ***value*** attributes contain string (*e.g.,* "KEY_X", "KEY_Y", or "KEY_Z") and integer (*e.g.,* 5, 10, 15, 20, 30, or 40) values, respectively. First, the *filtering* operator filters out all the tuples that do not satisfy the condition in WHERE clause (*i.e.,* ***value*** > 10). As shown in Fig. 5.4.3, 4 tuples (t1, t6, t8, and t9) are filtered out and the other tuples (t2, t3, t4, t5, t7, t10, t11, and t12) are bypassed by the filtering operator. After that, the *grouping* operator splits the bypassed tuples into 3 different groups according to their ***key*** attributes (*i.e.,* "KEY_X", "KEY_Y", and "KEY_Z"). As shown in Fig. 5.4.3, the first group ("KEY_X") contains 3 tuples (t4', t7', and t10'). Similarly, the second group ("KEY_Y") contains 3 tuples (t2', t5', and t11'). Finally, the third group ("KEY_Z") contains 2 tuples (t3' and t12'). The last step is to calculate the sum of ***values*** for each group separately. As shown in Fig. 5.4.3, the *aggregation* operator calculates those ***values*** and generates an output tuple for each group with the following schema: ⟨*key*, SUM(*value*)⟩. In particular, the sum of the ***value*** of t4', t7', and t10' yields a result of 85 (*i.e.,* 15 + 30 + 40 = 85) for the first group ("KEY_X"). Similarly, the sum of the ***value*** of t2', t5', and t11' yields a result of 55 (*i.e.,* 15 + 20 + 20 = 55) for the second group ("KEY_Y"). Finally, the sum of the ***value*** of t3' and t12' yields a result of 50 (*i.e.,* 20 + 30 = 50) for the third group ("KEY_Z") as indicated in the top of Fig. 5.4.3.

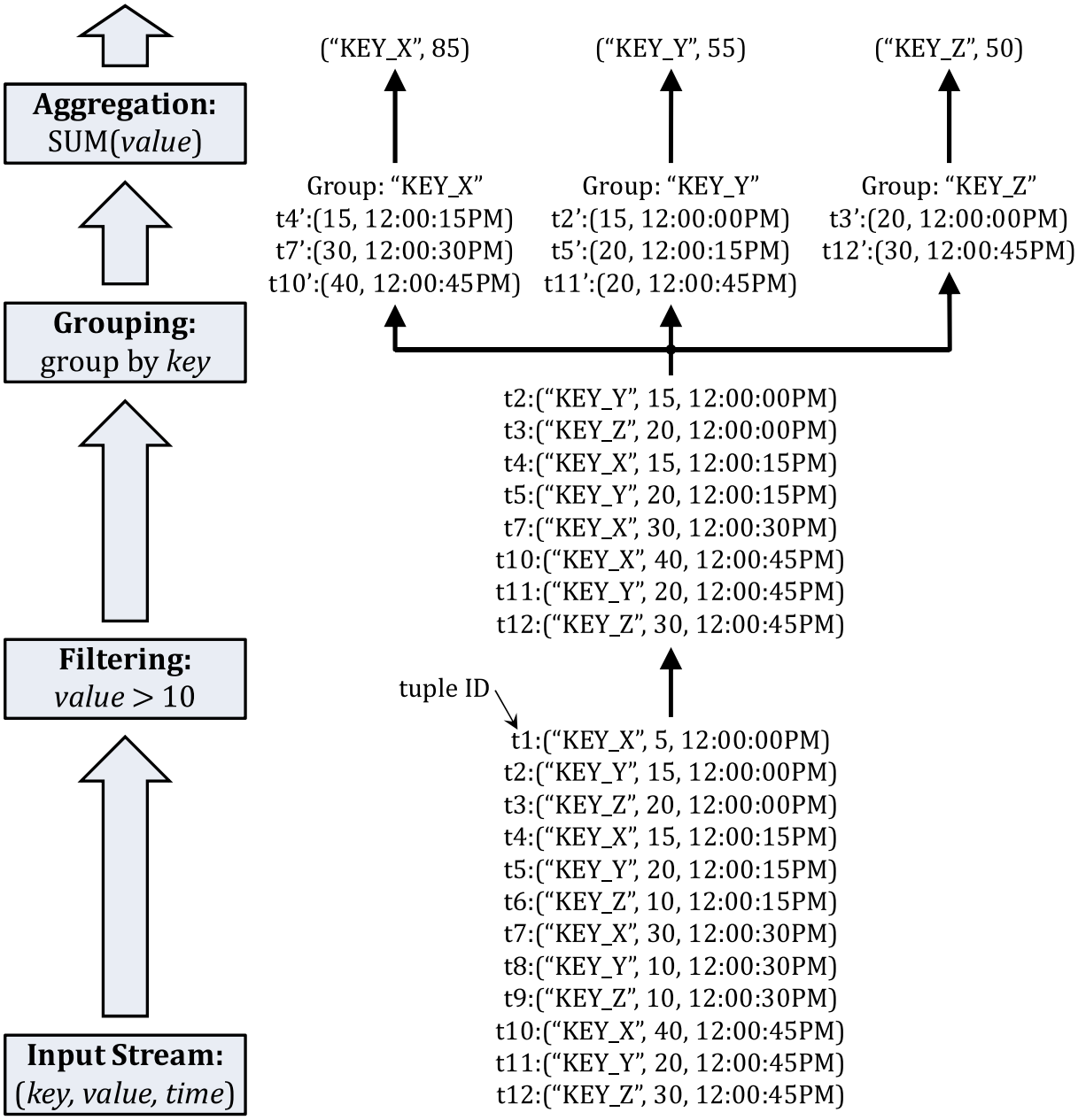



Figure 5.4.3: An example of filtering, grouping, and aggregation operations for Query $Q_7$ for the first window (between 12:00:00 and 12:01:00 p.m.).

Inside CQPH, the filtering, grouping, and aggregation opertions are performed by *(i) Shared Selection Module*, *(ii) Group-by Managers*, and *(iii) Aggragation Pipelines*, respectively. As illustrated in Fig. 5.4.1, each Agggregation Pipeline includes pane-level and window-level sub-queries along with a pane buffer. The following subsections describe each hardware module in more detail.

```
<boolean expr.> ::= True | False | <tree>
<tree> ::= <predicate>
        | (<tree> AND <tree>)
        | (<tree> OR <tree>)
<predicate> ::= <attribute> <op.> <literal>
<op.> ::= = | ≠ | > | ≥ | < | ≤
```

Figure 5.4.4: Boolean expressions supported by CQPH.

### 5.4.2 Selection Operator

Shared selection module (see the bottom of Fig. 5.4.1) determines whether or not an incoming tuple satisfies a given set of selection predicates. A selection predicate, or simply a predicate, specifies a condition that is either true or false about an input tuple. In SQL-like queries, a selection predicate is typically given as a Boolean expression in WHERE clauses. The current prototype of CQPH only accepts predicates on fixed-length attributes; however, it can still support different kinds of selection conditions, ranging from a single predicate to complex Boolean expressions (see Fig. 5.4.4).

In the proposed design, the shared selection module consists of two types of configurable hardware modules: (i) selection predicate module, and (ii) binary reducer module. These hardware modules do not have to be allocated to queries statically. Rather, one can assign an arbitrary number of selection predicates for a specific query at run time. In other words, the same circuit can be utilized for either many queries each of which is assigned to a single predicate or fewer queries with complex Boolean expressions. In either case, the total number of the selection predicate modules limits the number of Boolean expressions that can be processed simultaneously. For example, Fig. 5.4.5 illustrates a simplified block diagram of the shared selection module, which is assigned to Query $Q_8 \sim Q_{10}$ (see Fig. 5.4.6).

With the proposed design, each *Boolean expression tree* (*i.e.,* Stage 2 of Fig. 5.4.5) can be configured to evaluate a selection condition, by sharing the results of the *selection predicate modules* (*i.e.,* Stage 1 of Fig. 5.4.5). For this purpose, CQPH-compiler keeps track of information about the conditions that are already assigned to each module. When a new query is registered, CQPH-compiler compares the new Boolean expression with currently registered ones and decides whether it is possible to share any of selection predicates. In contrast, when a registered query is removed, corresponding modules are cleared unless these modules are shared by the other queries.

Recall from Fig. 5.3.2 that the wiring interface of each module consists of a bit flag field and data field. The valid flags of Boolean expression trees are integrated into the bit flag field of the shared selection module. It should also be mentioned that the shared selection module simply forwards data

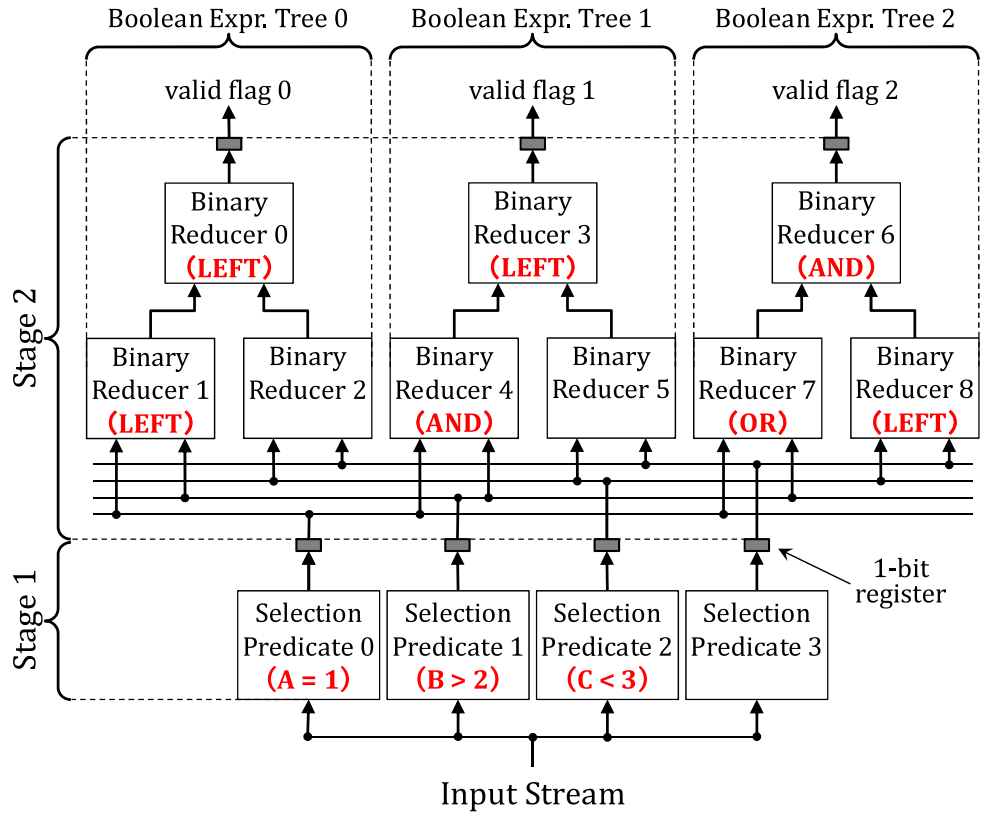



Figure 5.4.5: A simplified block diagram of the shared selection module instantiated with the following parameters: *(i)* # of selection predicate modules = 4 (**Stage 1**) and *(ii)* # of Boolean expression trees = 3 (**Stage 2**).

.

```
Q8: SELECT * FROM S WHERE A=1
Q9: SELECT * FROM S WHERE A=1 AND B>2
Q10: SELECT * FROM S WHERE (A=1 OR B>2) AND C<3
```

Figure 5.4.6: $Q_8 \sim Q_{10}$: "Given an input stream $S = \langle A : \text{int}, B : \text{int}, C : \text{int} \rangle$, select all tuples that satisfy predicates of each query."

field to the next module. Therefore, for example, when *valid flag* 0 and 2 are asserted (*i.e.,* set to logic "1") and *valid flag* 1 is negated (*i.e.,* set to logic "0"), datum on the data field is considered as a valid tuple for Query $Q_8$ and $Q_{10}$, but not for $Q_9$.

Note that even though the Boolean expression trees provide a certain degree of flexibility, probably the most flexible approach is to use *truth tables* as proposed in the Ibex system [49]. In fact, the same technique as that of Ibex could be used to implement the shared selection module of CQPH. Nevertheless,

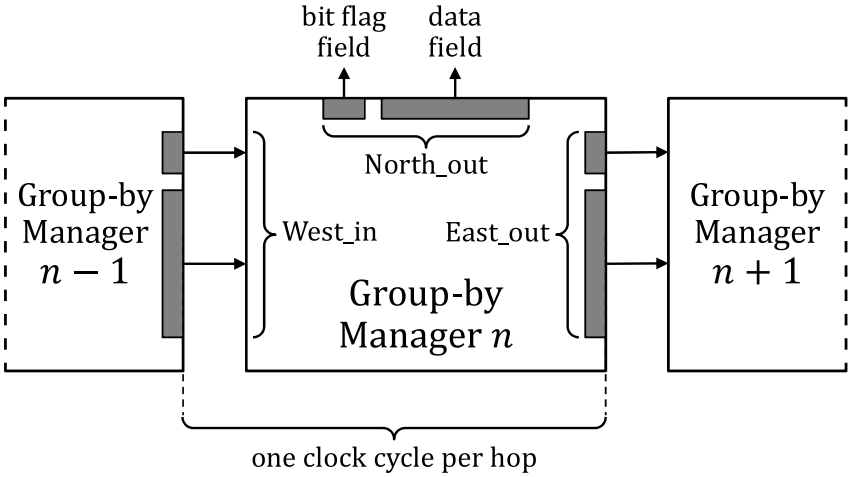



Figure 5.4.7: Wiring interface of a Group-by Manager module. Each module accepts its input from West_in port and transfers its output to East_out and/or North_out ports.

there is always a trade-off between flexibility and hardware cost. In particular, Ibex consumes on-chip block memories (*i.e.,* BRAMs) to store complex Boolean expressions as explicit truth tables. In our case, however, a large number of BRAMs are required to implement multiple aggregation pipelines as shown in Fig. 5.4.1 (each pane buffer is implemented using BRAM primitives). In this work, therefore, we choose binary tree-based approach (which requires no BRAMs) to implement the shared selection module, rather than BRAM-based truth tables.

### 5.4.3 Group-by Operator

In SQL-like queries, a GROUP-BY clause can be used along with an aggregate function to calculate an aggregate value for each group of tuples. As shown in Query $Q_6$, the template query can optionally support GROUP-BY clause. The group-by operation is regarded as a tuple-routing problem in the proposed CQPH architecture. The routing logic basically relies on the *valid flag fields* and *grouping-attribute value* of each tuple. Recall from Fig. 5.4.5 that valid flags are generated by the Boolean expression trees of the shared selection module. The grouping-attribute value can simply be extracted from the data field of each tuple.

As indicated in Fig. 5.4.1, CQPH architecture includes a number *Group-by Manager (GM)* modules that provide grouping functionality. The number of the GM modules can be given as a configuration parameter in the static configuration phase of FPGA. Therefore, an arbitrary number of GM modules can be instantiated in CQPH architecture where the only limit is the available area on an FPGA.

Each GM module accepts a new tuple from West port and forwards it to either (i) East port or (ii) both of the two output ports (*i.e.,* North and East ports) as shown in Fig. 5.4.1. Fig. 5.4.7 illustrates

the wiring interface of a GM module in more detail. Each GM module includes two sets of the wiring interface: East_out and North_out, respectively. As shown in Fig. 5.4.7, $GM_n$ takes its input from $GM_{n-1}$ and forwards its output to either one or both of the output interfaces. As indicated in Fig. 5.4.7, East_out port of $GM_n$ is connected to $GM_{n+1}$ In addition, North_out port of $GM_n$ is connected to an aggregation pipeline as shown in Fig. 5.4.1.

As mentioned above, we consider the group-by operation as a tuple-routing problem; therefore, GM modules require a routing logic to provide grouping functionality. In fact, there are two kinds of routing logic inside a GM module: one for queries *with* a GROUP-BY clause (see Fig. 5.4.8) and the other for queries *without* a GROUP-BY clause (see Fig. 5.4.9). Since the latter is a simplified version of the former, we focus on the former case in the following example.

Fig. 5.4.10 illustrates the basic idea of how input tuples are processed by GM modules. Each GM module includes a *Query ID (qid)* register which can be configured at the dynamic-configuration phase. The example in Fig. 5.4.10 assumes that $GM_{n-1}$ and $GM_n$ have already been configured for qid = 0. Similarly, $GM_{n+1}$ has been configured for qid = 1. These IDs are related to the valid flag fields of each tuple, which means that both $GM_{n-1}$ and $GM_n$ are assigned to *valid flag 0* (*i.e., qid 0*) while $GM_{n+1}$ is assigned to *valid flag 1* (*i.e., qid 1*).

According to the given routing logic, input tuples are processed based on the *valid flag fields* and *grouping-attribute value*. For instance, at the time $t = t_0$, $GM_{n-1}$ receives a new tuple which belongs to *Group X*. Notice that *valid flag 0* is asserted and this means that the input tuple is valid for *qid 0*. Since $GM_{n-1}$ is not yet assigned to any group (*i.e.,* Grp=Null), *Group X* is registered to $GM_{n-1}$ at the next clock cycle (*i.e.,* $t = t_0 + 1$). At the same time, the input tuple is forwarded to both North (aggregation pipeline) and East (next GM module) ports. It is also important to note that *valid flag 0* is negated to indicate that the corresponding tuple has already been processed for *qid 0*.

Note that after the time $t = t_0$, $GM_{n-1}$ takes responsibility for tuples with *Group X* and the other tuples are simply bypassed to $GM_n$. For example, when $t = t_0 + 1$, $GM_{n-1}$ receives a new tuple with *Group Y*, which is simply bypassed to $GM_n$ at the next clock cycle (*i.e.,* $t = t_0 + 2$). In contrast, when $t = t_0 + 2$, $GM_{n-1}$ receives a new tuple with *Group X*. In this case, the input tuple (with *Group X*) is forwarded to both North and East ports with *valid flag 0* negated as shown in Fig. 5.4.10.

For those queries *with* a GROUP-BY clause, each aggregation pipeline receives tuples from one group only. In addition, incoming tuples are always processed on a first-come-first-served basis, by aggregation pipelines independently of each other. It should be also mentioned that, for those queries *without* a GROUP-BY clause, GM modules use the simplified version of the routing logic (*i.e.,* Fig. 5.4.9). In this case, input tuples are routed based only on *valid flag fields*; therefore, an aggregation pipeline can receive different kinds of tuples.

```
Input:
 1: West_in port
Output:
 2: East_out port
 3: North_out port
Initialization (dynamically-configurable registers):
 4: qid_reg ← a specific Query ID
 5: group_reg ← NULL
Synchronous Update:
 6: for each clock cycle do
 7:     extract valid_flag[qid_reg] from West_in port
 8:     if valid_flag[qid_reg] = 1 then
 9:         extract grouping_attribute from West_in port
10:         if group_reg = NULL then
11:             group_reg ← grouping_attribute
12:             North_out ← West_in
13:             valid_flag[qid_reg] ← 0
14:         else if group_reg = grouping_attribute then
15:             North_out ← West_in
16:             valid_flag[qid_reg] ← 0
17:         end if
18:     end if
19:     East_out ← West_in
20: end for
```

Figure 5.4.8: Pseudo code of the routing logic for queries *with* a GROUP-BY clause.

It should also be emphasized that GM modules perform the same operation on every single clock cycle (in a synchronous manner). In fact, all operations described lines between 7 ~ 19 of Fig. 5.4.8 (or 6 ~ 11 of Fig. 5.4.9) can be completed in just one clock cycle. In addition, each GM module requires only local communication for transferring tuples to its adjacent modules. From this point of view, GM modules and their connections can be regarded as a one-dimensional linear systolic array [16]. The data processing and communication model of GM modules are consistent with the properties of systolic architectures.

**Input:**
1: West_in port
**Output:**
2: East_out port
3: North_out port
**Initialization (dynamically-configurable registers):**
4: $qid_{\text{reg}} \leftarrow$ a specific Query ID
**Synchronous Update:**
5: **for each** clock cycle **do**
6: extract $valid\_flag[qid_{\text{reg}}]$ from West_in port
7: **if** $valid\_flag[qid_{\text{reg}}] = 1$ **then**
8: North_out ← West_in
9: $valid\_flag[qid_{\text{reg}}] \leftarrow 0$
10: **end if**
11: East_out ← West_in
12: **end for**

Figure 5.4.9: Simplified version of the routing logic for queries *without* a GROUP-BY clause.

In the current implementation of CQPH-compiler, the compiler requires the maximum number of groups that should be handled by CQPH. In practice, the total number of GM modules provisioned in the CQPH limits the number of groups (or queries) that can be processed simultaneously. Note that CQPH can support the same order of groups as Glacier [25] (*e.g.,* less than a hundred). If the number of groups exceeds the limit of CQPH at run time, these groups can be obtained from East port of the last GM module (see the dashed arrow of $GM_N$ in Fig. 5.4.1). By using a similar approach as that of Ibex [49], these groups could be bypassed to a host system for further processing (though the processing of bypassed tuples is out of scope of this work).

### 5.4.4 Window-aggregation Operator

CQPH architecture supports the same aggregate functions as previous Chapters (*i.e.,* Chapter 3 and Chapter 4). In particular, the current version of CQPH can support the following aggregate functions: COUNT, SUM, MIN, and MAX. Recall from Table 4.1 (Chapter 4) that these aggregate functions can be decomposed into pane-level and window-level sub-queries (*i.e.,* PLQ and WLQ), respectively.

The proposed CQPH architecture adopts the two-step aggregation technique using panes. In Chap-

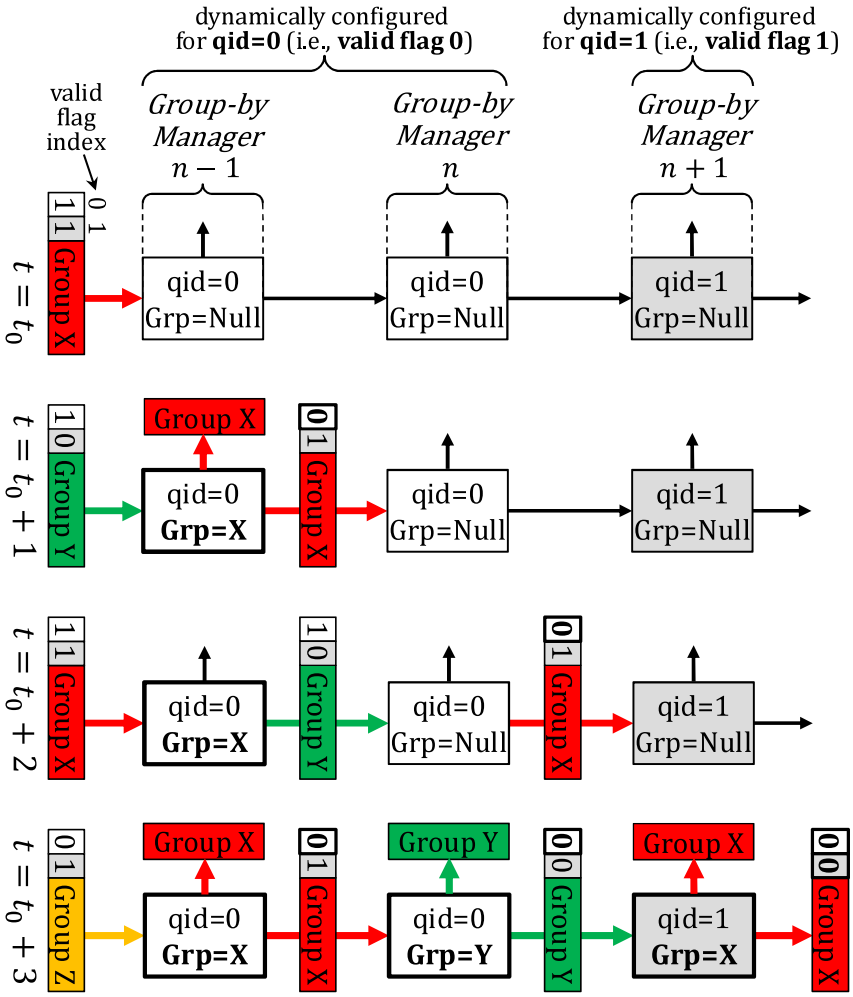



Figure 5.4.10: A simple example for the group-by operation.

ter 4, we have presented a pane-based implementation of a sliding-window aggregate query. This can be characterized as a static FPGA-based query processor, especially tailored for a specific query. On the other hand, in the proposed CQPH architecture, the same idea (*i.e.,* the two-step aggregation technique) is implemented based on the configurable hardware modules (recall from Fig. 5.3.2). This is the main difference between the pane-based implementation presented in Chapter 4 and the CQPH architecture.

As shown in the overview of the CQPH architecture (Fig. 5.4.1), a pair of PLQ and WLQ is implemented in a pipelined fashion with a pane buffer for each pair of two sub-queries. As described in Section 5.4.3, each of the aggregation pipelines can be mapped to a group or query at run time. With the proposed design, the multi-pipeline architecture of CQPH allows users to execute multiple continuous queries concurrently.

It is important to note that each aggregation module processes incoming tuples immediately after arrival, rather than batching them up until a pane or window closes. In particular, PLQ and WLQ modules

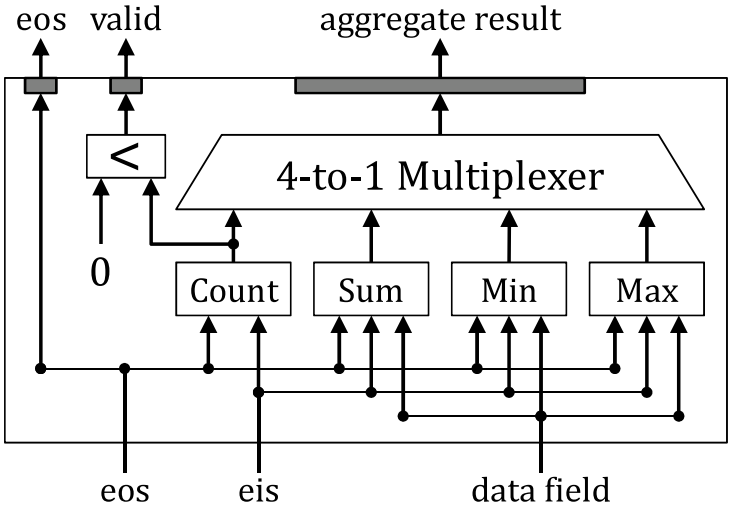



Figure 5.4.11: A simplified block diagram of the aggregation module that includes four sub-modules (*i.e.,* aggregate circuits): COUNT, SUM, MIN, and MAX.

do not store all of the incoming tuples. Instead, CQPH adopts a similar approach as that of Glacier [25] and the input tuples are directly forwarded to aggregation circuits inside PLQ or WLQ module (see Fig. 5.4.11). In fact, as mentioned in [25], this approach has the advantage that each aggregation circuit needs to provide storage just for the amount of state it requires (*i.e.,* a fixed amount of state), rather than maintaining the entire pane or window. In other words, each aggregate circuit incrementally computes its own aggregate value and only stores the current (*i.e.,* partial) result of the aggregation.

Each aggregation module requires two control signals: *enable input stream (eis)* and *end of stream (eos)* as shown in Fig. 5.4.11. Note that these signals can be generated with a simple counter for count-based windows whereas time-based windows require additional logic to generate the control signals. For example, Fig. 5.4.12 describes how *eis* and *eos* signals can be generated for a time-based window (*e.g.,* PLQ). Whenever *eis* is asserted, *data field* should be considered as a valid tuple and the aggregation module accepts the input tuple. Once the input stream reaches the end of the current pane, *eos* is asserted and the aggregate value is emitted to the upstream data path.

### 5.4.5 Union Operator

From a data flow point of view, a union operator accumulates several source streams into a single output stream [25]. CQPH adopts a similar approach as that of Glacier [25] to implement a union operator. That is to say, all source streams (*i.e.,* output streams of the aggregation pipelines) are buffered by FIFOs as indicated in Fig. 5.4.1, and the union module forwards output tuples in a round-robin fashion from the FIFO buffers.

**Initialization (dynamically-configurable registers):**

1: $pane_{\text{begin}} \leftarrow time_{\text{start}}$
2: $pane_{\text{end}} \leftarrow time_{\text{start}} + RANGE_{\text{PLQ}}$
3: $pane_{\text{move}} \leftarrow SLIDE_{\text{PLQ}}$

**Synchronous Update:**

4: **for each** clock cycle **do**
5: $eos \leftarrow 0$ (default value)
6: $eis \leftarrow 0$ (default value)
7: **if** $punctuation\_flag = 1$ **then**
8: **if** $pane_{\text{end}} \leq timestamp_{\text{punc.}}$ **then**
9: $eos \leftarrow 1$
10: $pane_{\text{begin}} \leftarrow pane_{\text{begin}} + pane_{\text{move}}$
11: $pane_{\text{end}} \leftarrow pane_{\text{end}} + pane_{\text{move}}$
12: **end if**
13: **else if** $valid\_flag = 1$ **then**
14: **if** $pane_{\text{begin}} < timestamp_{\text{tuple}} \leq pane_{\text{end}}$ **then**
15: $eis \leftarrow 1$
16: **end if**
17: **end if**
18: **end for**

Figure 5.4.12: Generation of *eis* and *eos* signals for PLQ.

The average input rate of all source streams should not exceed more than 1 tuple/cycle, which is the maximum tuple rate of the output port of the union operator. This constraint is necessary to prevent possible data loss due to a buffer overflow. In the proposed design, the union module provides an admission control signal for the input interface of CQPH when any of the FIFO buffers is close to overflow. With the admission control mechanism, the union module takes responsibility for input tuples accepted by CQPH. This means that no data loss occurs inside the union module (*i.e.,* between each aggregation pipeline and the output channel).

On the other hand, this does not always prevent loss of the actual query results. Strictly speaking, a lossless flow of all query results is impossible when the output channel could not keep up with a high input data rate. In such a situation, load shedding [42] or distribution [1] techniques can be used, even though the implementation of such a technique is beyond the scope of this work. Note that the

admission control mechanism is consistent with load shedding techniques because CQPH can produce more valuable results with the appropriately filtered inputs once a load shedding mechanism reduces the input load.

It should be mentioned that the union module is implemented in a pipelined fashion, and the number of pipeline stages ($N_S$) can be given as a static configuration parameter. A typical value of $N_S$ is one or two, depending on the size of the union module. Note that there is a trade-off between latency and the maximum clock frequency. A large value of $N_S$ imposes a latency cost (*i.e.,* two clock cycles per stage); however, this can increase the allowable operating-frequency range by reducing the critical path delay of the union module.

## 5.5 Evaluation

### 5.5.1 Latency and Throughput

#### Performance Metrics

In this Section, we evaluate the circuit characteristics of the proposed CQPH architecture. As stated in [25], the performance characteristics of a query processing circuit can accurately be derived by solely analyzing the circuit design. We follow a similar approach and evaluate the circuit characteristics of CQPH in terms of *latency* and *issue rate*. Latency and issue rate are important metrics to determine the performance of hardware circuit. In particular, latency directly corresponds to the observable response time, whereas issue rate determines throughput [25].

#### Latency

Table 5.2 summarizes latencies and issue rate of each operation in CQPH. Latency is measured in terms of the number of the clock cycles between the time when a tuple enters the circuit and the time when a result item is produced. In Table 5.2, $N_G$ and $N_S$ represent the number of Group-by Manager modules and the number of pipeline stages of Union module, respectively. Recall from Table 5.1 that both of these parameters are provided in the static configuration phase of FPGA.

Note that grouping and windowing operators cannot produce their outputs before they have seen the last tuple of the respective window [25]. For these operators, therefore, latency is defined as the number of clock cycles between the closing of the input window and the generation of the *first* output tuple. In Table 5.2, the lower bound indicates the latency of the *first* output tuple whereas the upper bound corresponds to the *last* output tuple of each window. For instance, assuming $N_S = 1$, CQPH can produce the first output tuple of each window in just 13 clock cycles. In other words, the proposed design can respond with a latency of only 130 nanoseconds when clocked at 100 MHz.

Table 5.2: Latency and Issue Rate of Each Operation.

| | **Latency**[†] | | **Issue Rate**[‡] |
|---|---|---|---|
| | **Lower** | **Upper** | |
| Selection | 2 | 2 | 1 |
| Group-by | 1 | $N_G$ | 1 |
| Window Aggregation | 6 | 6 | 1 |
| Union | $2N_S + 2$ | $2N_S + 2$ | 1 |
| **Overall Operations** | $2N_S + 11$ | $N_G + 2N_S + 10$ | 1 |

[†] In this table, latencies are given in terms of the number of clock cycles.

[‡] Issue rate is defined as the number of tuples that can be processed per clock cycle.

#### Throughput

Issue rate is defined as the number of tuples that can be processed per clock cycle. As shown in Table 5.2, issue rate of each operation is 1 tuple/cycle, which means that each operator can accept a new tuple every clock cycle. The overall issue rate of CQPH is determined by the slowest operation (*i.e.,* the minimum issue rate); thus, the issue rate of the overall operation is also 1 tuple/cycle. It is important to note that CQPH operates in a strict streaming fashion independent of the query workload, and the maximum throughput of the circuit is directly dependent on its clock rate. For example, the proposed design can process up to 100 million tuples per second when clocked at 100 MHz.

### 5.5.2 Dynamic Configuration Time

Table 5.3 summarizes dynamic configuration time of CQPH for a given query. In Table 5.3, $N_G$ and $N_{SP}$ represent the number of Group-by Manager modules and the number of Selection Predicate modules, respectively. Recall from Section 5.3.1 that query configuration data are divided into a set of *configuration tuples* each of which corresponds to a single hardware module. Given the configuration tuples, dynamic configuration time can be measured in terms of the number of clock cycles required to send all of these tuples to CQPH.

The lower bound of Table 5.3 corresponds to a simple sliding-window aggregate query without WHERE or GROUP-BY clause. On the other hand, the upper bound represents a more generic form of the aggregate query with WHERE and GROUP-BY clause (*e.g.,* Query $Q_6$). In either case, the proposed

Table 5.3: Dynamic Configuration Time† for a Given Query.

| | **Lower** | **Upper** |
|---|---|---|
| Selection Predicate | 0 | $N_{SP} = 2^n, n \in \mathbb{Z}^+$ |
| Binary Reducer | 1 | $N_{SP} - 1$ |
| Group-by Manager | 1 | $N_G$ |
| Pane-level Sub-query | 2 | $2N_G$ |
| Window-level Sub-query | 2 | $2N_G$ |
| **Total Configuration Time** | 6 | $2N_{SP} + 5N_G - 1$ |

† In this table, configuration times are given in terms of the number of clock cycles.

CQPH architecture can offer run-time configurability at negligible cost compared to query-tailored circuits such as Glacier [25], the WID-based implementation (presented in Chapter 3), and the pane-based implementation (presented in Chapter 4).

### 5.5.3 Case Study

#### Evaluation Setup

In this Section, we evaluate hardware resource utilization and performance of the proposed CQPH architecture through a case study. This case study considers the same financial trading application as that of Chapter 3 and Chapter 4. In this application, over a million tuples can arrive per second, and at the same time, latency is critical and measured in units of microseconds ($\mu$s) [25]. It is therefore crucial for CQPH to meet these performance requirements.

In the evaluation, we assume an input stream with the following schema: $\langle Symbol, Price, Volume, Time \rangle$ as explained in Section 3.3 (Chapter 3). Note that, in practice, a stream schema may consist of a number of different attributes, and input tuples can be delivered using an application-specific protocol (*e.g.,* FIX protocol [13] for algorithmic trading). In such a case, input streams should be pre-processed before delivery to CQPH. For example, in the previous work [25], a *stream de-multiplexer* is implemented to process a compressed variant of the FIX protocol and dispatch each tuple to the proper query-processing circuit. A similar approach can be applied for CQPH. Moreover, if necessary, a projection and selection operator can be implemented with a segment-at-a-time processing model [29] to extract only necessary attributes for CQPH. However, the implementation of such an operator is beyond the scope of this case

Table 5.4: Specifications of the Kintex®-7 FPGA (XC7K325T).

| | |
|---|---|
| # of Slice Registers | 407,600 |
| # of Slice LUTs | 203,800 |
| # of Slices | 50,950 |
| # of BRAM (36Kbit) | 445 |

Table 5.5: Static Configuration Parameters.

| | |
|---|---|
| Stream schema | 128 bits, 4 attributes |
| Pane Buffer Size | 64 bits, 2048 entries |
| # of Selection Predicate Modules | $N_{SP} \in \{2, 4, 8, 16, 32, 64\}$ |
| # of Boolean Expr. Trees | $N_G \in \{2, 4, 8, 16, 32, 64\}$ |
| # of Group-by Manager Modules | $N_G$ |
| # of Aggregation Pipelines | $N_G$ |
| # of Pipeline Stages of Union | $N_S = \begin{cases} 1 & \text{if } N_G \leq 8 \\ 2 & \text{otherwise} \end{cases}$ |

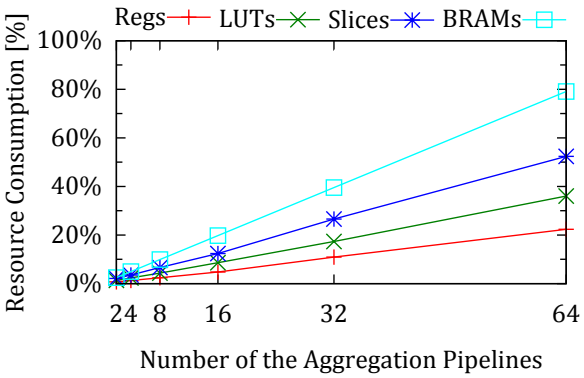
study.

For the above application, we have implemented CQPH on a Kintex-7 XC7K325T FPGA [50] with different parameter settings. The specification of the Kintex-7 FPGA and the configuration parameters are given in Table 5.4 and Table 5.5, respectively. Xilinx ISE 14.4 is used during the implementation process (*e.g.,* synthesis and place-and-route). The proposed CQPH architecture is synthesized with a timing constraint of 6.37 ns for each configuration, which yields the target clock frequency of 156 MHz. Note that when sufficient I/O bandwidth is available, the theoretical peak performance of CQPH can be calculated as follows (see Equation 5.5.1).

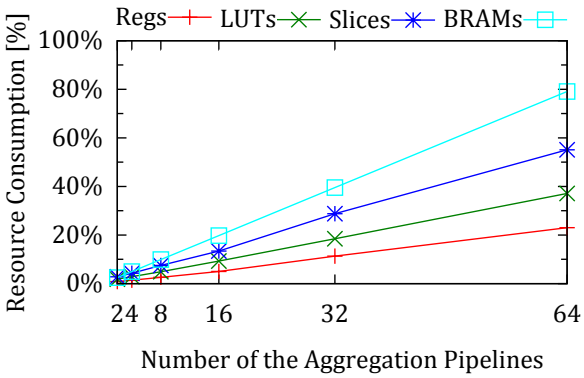
$$T_{peak} = d \times f \tag{5.5.1}$$

where:

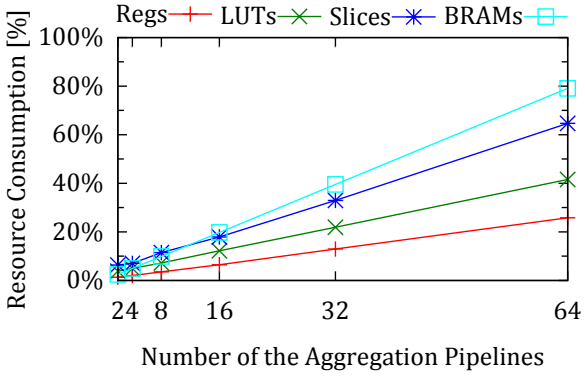
$T_{peak}$ = theoretical peak throughput
$d$ = data width of input tuple
$f$ = operating clock frequency




(a) Static Configuration Parameters: $N_{SP} = 4$ and $N_G = 2 \sim 64$.




(b) Static Configuration Parameters: $N_{SP} = 16$ and $N_G = 2 \sim 64$.



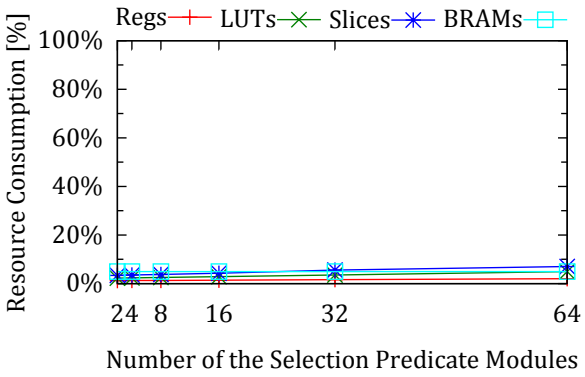

(c) Static Configuration Parameters: $N_{SP} = 64$ and $N_G = 2 \sim 64$.

Figure 5.5.1: Overall resource consumption of FPGA to implement CQPH with an increasing value of $N_G = 2 \sim 64$.

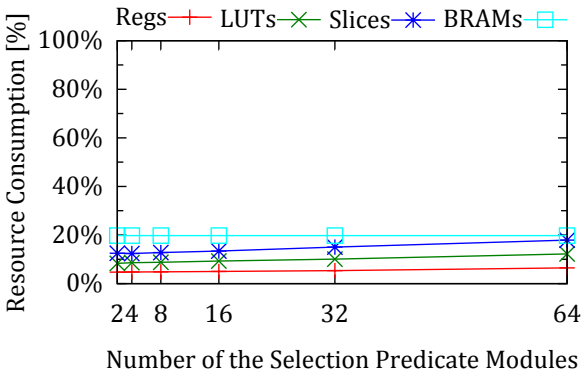
For example, data width is 128 bits in the case study (see Table 5.5) and if we assume a clock rate of 156 MHz, the peak performance is equivalent to nearly 20 Gbps (*i.e.,* 128 bits × 156 MHz = 19,968 Mbps).

### Hardware Resource Utilization and Performance

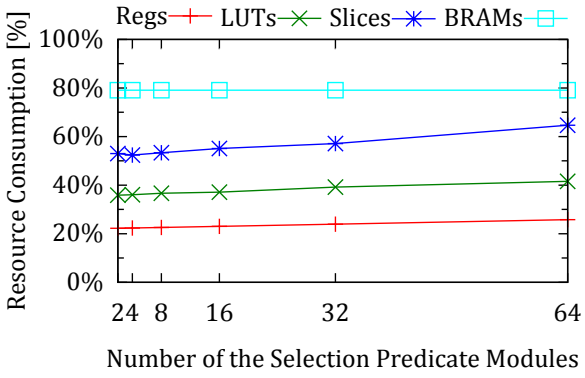
**Hardware Resource Utilization.** The overall resource consumption is shown in Fig. 5.5.1 and Fig. 5.5.2. Each graph of Fig. 5.5.1 and Fig. 5.5.2 represents the resource consumption in terms of percentages of the total available resources on the target FPGA device. These graphs indicate trade-offs between area (*i.e.,* resource utilization) and flexibility of CQPH. For example, all four graphs (*i.e.,* Registers, LUTs, Slices, BRAMs) of Fig. 5.5.1a, 5.5.1b, and 5.5.1c linearly increases with increasing $N_G$ values. Results of these graphs suggest that CQPH provides linear scalability in terms of the hardware resource utilization. Furthermore, each graph of Fig. 5.5.2a, 5.5.2b, and 5.5.2c indicates a relatively small increase with




(a) Static Configuration Parameters: $N_{SP} = 2 \sim 64$ and $N_G = 4$.




(b) Static Configuration Parameters: $N_{SP} = 2 \sim 64$ and $N_G = 16$.




(c) Static Configuration Parameters: $N_{SP} = 2 \sim 64$ and $N_G = 64$.

Figure 5.5.2: Overall resource consumption of FPGA to implement CQPH with an increasing value of $N_{SP} = 2 \sim 64$.

respect to $N_{SP}$ values. This means that we can easily pre-allocate a large number of selection predicate modules to support complex Boolean expressions.

**Performance**. Each implementation meets the timing constraint of 6.37 ns and achieves the target clock frequency of 156 MHz. Since we have obtained almost similar results for each configuration, a typical result of the maximum clock frequency is shown in Fig. 5.5.3. The clock frequency is obtained from post-place & route static timing report, which is provided by Xilinx's Timing Analyzer tool. Since the issue rate is equal to 1 tuple/cycle, CQPH can process up to 156 million tuples/second. As for latency, CQPH can respond in the order of microseconds with a cycle time of 6.37 ns. Moreover, even if we consider the worst-case configuration time (*i.e.,* $N_{SP} = 64$ and $N_G = 64$), a given query can be configured within 447 clock cycles. With a cycle time of 6.37 ns, this is equivalent to less than 3 microseconds. These data lead us to the conclusion that the proposed design can accomplish both high-throughput (over 150 million

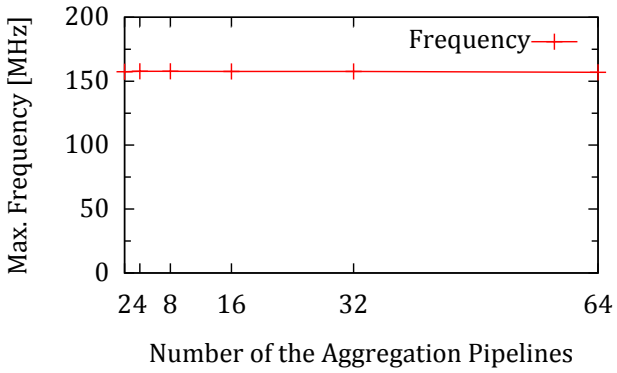



Figure 5.5.3: Maximum clock frequency of CQPH implemented with the following parameters: $N_{SP} = 64$ and $N_G = 2 \sim 64$.

tuples/second) and low-latency (in the order of microseconds) requirements of the application.

It is also important to emphasize that the maximum clock frequency remains unaffected by the increasing number of $N_{SP}$ or $N_G$. The fact that CQPH can still sustain a given clock rate is a good indication for the scalability as indicated in Fig. 5.5.3. In other words, by increasing $N_{SP}$ or $N_G$, we can easily increase the flexibility and the workload capacity of CQPH. For example, if we use a larger FPGA and increase $N_G$ value, the maximum number of groups can be increased for GROUP-BY aggregate queries. Moreover, the number of aggregation pipelines (*i.e.,* $N_G$) determines the upper limit on the number of parallel queries that can be executed by CQPH at one time. For instance, if we assume simple aggregate queries (without a GROUP-BY clause) and increase $N_G$ from 64 to 128, CQPH can simultaneously execute up to 128 parallel queries without sacrificing the performance (*i.e.,* throughput).

**Experimental Measurement**

As mentioned in Chapter 4, a key aspect of using an FPGA for data stream processing is its flexibility that enables us to insert custom logic into an existing data path. For example, the proposed CQPH architecture can be tightly integrated with a physical network interface [11] inside an FPGA. Our experiments are based on a KC705 board [50], which includes a Gigabit Ethernet interface and a 1 GB DDR3 SDRAM. The experimental system consists of a KC705 board and a host pc, which are directly connected by a dedicated Ethernet cable. In this experiment, input stream is generated based on historical stock data for securities traded on NASDAQ*. In particular, we have focused on a subset of the stock symbols (25 different companies) of the market data as listed in Table 5.6. The schema of the input stream is same as that of Chapter 3 and Chapter 4, which is defined as follows: $Trades = \langle Symbol : \text{char}[4], Price : \text{int}, Volume : \text{int}, Time : \text{int} \rangle$. The *Symbol* attribute contains one of the constant strings listed in Table 5.6

*The historical data was obtained on June 13, 2014 from the following URL: http://thebonnotgang.com/tbg/.

Table 5.6: List of Symbols.

| | | | | |
|---|---|---|---|---|
| "AAPL" | "AMAT" | "BPOP" | "BRCD" | "CALP" |
| "CMCS" | "CSCO" | "DELL" | "EBAY" | "FITB" |
| "GOOG" | "HBAN" | "INTC" | "LVLT" | "MSFT" |
| "MU␣␣" | "NVDA" | "NWSA" | "ORCL" | "QCOM" |
| "RIMM" | "SIRI" | "SNDK" | "WIN␣" | "YHOO" |

```
SELECT Time, Symbol, <AGGREGATE>
  FROM Trades [RANGE <WIN_RANGE> seconds
               SLIDE <WIN_SLIDE> seconds
               WATTR Time]
 WHERE Symbol in (<SYMBOL_LIST>)
 GROUP BY Symbol
```

Figure 5.5.4: $Q_{11}$: Template of a benchmark query for CQPH.

(4 bytes). The *Price* attributes contains price data in fixed-point number representation (they are treated as 32-bit integers internally). The *Volume* attributes contains volume data as 32-bit integers. The *Time* attribute contains timestamp (*i.e.,* epoch time) as 32-bit integers. A data generator on the host computer merges the historical stock data of 25 different securities into a single stream. The test data includes 6,337,580 input tuples in non-decreasing order with respect to their *Time* attribute. In the following series of experiments, the *SLACK* value is set to zero.

As for the query workload, we use random query sets each of which consists of 1, 2, 4, 8, 16, 32, or 64 queries. Each query is based on a template query given in Fig. 5.5.4. From Table 5.7, a single value is selected for each of <**AGGREGATE**>, <**WIN_RANGE**>, and <**WIN_SLIDE**>. As for <**SYMBOL_LIST**>, we choose 1, 4, or 16 different symbols at random, respectively.

**Experiment 1**. We have measured the effective throughput of CQPH ($N_{\mathrm{SP}} = 64$ and $N_{\mathrm{G}} = 64$) on the KC705 FPGA board. Results of the experiments show that sliding-window aggregate queries on the CQPH achieves an effective throughput up to around 760 Mbps for each query set. This is the upper bound of the available bandwidth that the network interface [11] can handle without packet loss. It should be emphasized that this is equivalent to nearly 6 million tuples per second, which means that the proposed setup can process significantly high tuple rates at wire-speed with zero packet loss.

Table 5.7: Query Parameters.

| | | |
|---|---|---|
| <**AGGREGATE**> | : | count(*), max(Price), min(Price), sum(Price), max(Volume), min(Volume), sum(Volume) |
| <**SYMBOL_LIST**> | : | subset of symbols *e.g.,* "GOOG", "AAPL", "YHOO", etc. |
| <**WIN_RANGE**> | : | 60, 300, 600, 1800 |
| <**WIN_SLIDE**> | : | 1, 5, 10, 30 |

**Experiment 2**. Since the Gigabit Ethernet is not sufficient to saturate CQPH, we use the DDR3 memory as a data source (6,337,580 tuples) to emulate a 10 Gigabit Ethernet speed. In this setup, we have implemented a dedicated AXI master interface for CQPH, and a Xilinx AXI Interconnect core IP is used to connect the CQPH and the DDR3 memory. It should be mentioned that bus width of the AXI interconnect and data width of the AXI master interface are both set to 128 bits. In addition, we set 100 MHz of clock frequency for all hardware components including *(i)* the AXI interconnect, *(ii)* the AXI master interface of the CQPH, and *(iii)* the CQPH itself to achieve over 10 Gbps throughput; namely, the theoretical peak throughput of this setup is 128 bits $\times$ 100 MHz = 12,800 Mbps (recall from Equation 5.5.1).

By design, CQPH can accept one input tuple per clock cycle; therefore, in order to achieve the theoretical peak performance, it is important to provide the CQPH with a new tuple every clock cycle. In practice, however, the AXI interconnect and the AXI master interface become a critical bottleneck due to the AXI protocol overhead. In fact, when input tuples are transferred from the DDR3 memory via the AXI interconnect, the CQPH can achieve over 10,400 Mbps effective throughput for each query set in our experiments. This corresponds to the memory access speed of the evaluation setup; thus, in effect, the CQPH is limited by the memory read speed of the AXI master interface. Nevertheless, these results suggest that the CQPH can still support even faster 10 Gigabit Ethernet at line rate when clocked at 100 MHz.

**Experiment 3**. Finally, we focus on query processing performance of HW- and SW-based solutions, by comparing the performance for sliding-window aggregate queries. For the comparison, two different approaches are considered as SW-based solutions. The first option is to use Esper v5.1.0 [12], a software-based event processing engine with efficient support for in-memory query processing. Esper supports SQL-based continuous query language, called *Event Processing Language* (EPL), and it is well capable

```
SELECT Time, Symbol, <AGGREGATE>
  FROM Trades(Symbol in (<SYMBOL_LIST>)).win
    ↪ :ext_timed(Time, <WIN_RANGE> seconds)
 GROUP BY Symbol
OUTPUT last every <WIN_SLIDE> seconds
```

Figure 5.5.5: $Q_{12}$: Template of Esper EPL query.

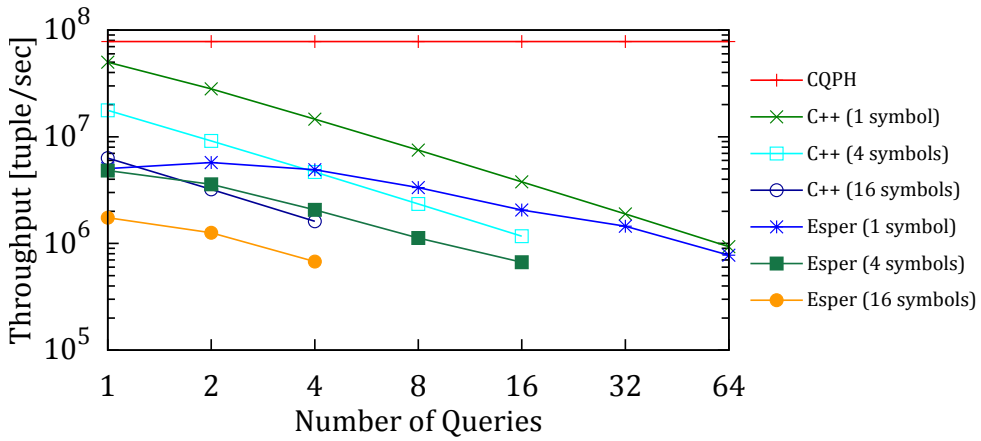



Figure 5.5.6: Performance comparison of multi-query execution between HW- and SW-based solutions.

of processing sliding-window aggregate queries. The second option is a prototype of a query processor written in C++, especially optimized for sliding-window aggregations. For these SW-based solutions, all experiments are conducted on a desktop PC featuring a modern Intel CPU (i7-4790K, 8 MB cache, 4.00 GHz) with 32 GB of main memory on a CentOS 6.6 (64-bit) platform. As for HW-based solution (*i.e.,* FPGA), a KC705 Evaluation Board [50] is used to implement the proposed CQPH architecture with $N_{SP} = 64$ and $N_G = 64$ values.

In this evaluation, we use a memory-cached file as a data source (6,337,580 tuples) and random query sets each of which consists of 1, 2, 4, 8, 16, 32, or 64 queries. Each query is based on a template query written in Event Processing Language (EPL) [12] for Esper (see Fig. 5.5.5), or its equivalent for the other solutions. As in the previous experiments (*i.e.,* Experiment 1 and Experiment 2), appropriate values are selected for each of <**AGGREGATE**>, <**WIN_RANGE**>, <**WIN_SLIDE**>, and <**SYMBOL_LIST**> from Table 5.7.

The performance comparison of CQPH, the C++-based system, and Esper is illustrated in Fig. 5.5.6. By design, CQPH can accept one input tuple per clock cycle; therefore, the throughput of the proposed system is independent of the query workload. This was confirmed by the measurements on the real

evaluation board for those queries with 1, 4 or 16 symbols (plotted as a single graph in Fig. 5.5.6 for brevity). We observed a maximum throughput of more than 78 million tuples per second on the KC705 board. This corresponds to the memory access speed of the evaluation setup; thus, in effect, CQPH is limited by the memory read speed (as explained in Experiment 2). On the other hand, the performance of the SW-based solutions (*i.e.,* C++ and Esper) is significantly degraded with respect to the number of queries as shown in Fig. 5.5.6. For example, when the number of queries equals to 64, CQPH outperforms the SW-based approaches by about two orders of magnitude ($\approx 100\times$).

These results lead us to the conclusion that the proposed CQPH architecture can process significantly high tuple rates even with increased workload—something which is not possible with the SW-based approaches. In the actual application, the Trades stream includes a subset of the stock indexes of market data [25]. In particular, with less than a hundred different stock symbols per stream (*e.g.,* $N_{\mathrm{G}} = 64$ symbols), CQPH architecture implemented on a moderate-sized FPGA (Kintex-7 XC7K325T) can be regarded as a promising application-specific accelerator for the window-aggregate workload.

# Chapter 6

# Conclustions

## 6.1 Summary

Data Stream Management Systems (DSMSs) deal with potentially infinite streams of data that should be processed for real-time applications, executing SQL-like continuous queries over data streams. It is essential for DSMSs that incoming data be processed in real time, or at least near real-time, depending on the applications' requirements. In order to meet the above-mentioned requirement, there is currently a great deal of interest in the potential of using field-programmable gate arrays (FPGAs) as custom accelerators for continuous query processing over data streams. In particular, Mueller *et al.* consider the use of FPGAs for data stream processing as co-processors [25], and they propose an implementation method for sliding-window aggregate queries on an FPGA. Nevertheless, there still remain three practical issues related to the implementation of sliding-window aggregation:

1. The first issue is that it is necessary to consider out-of-order arrival of tuples at a windowing operator.

2. The second issue is that a large number of overlapping sliding-windows cause severe scalability problems in terms of both performance and area.

3. The third issue is that there is a lack of run-time configurability, which severely limits the practical use in a wide range of applications.

In the present study, we have addressed the above issues and proposed alternative approaches to implement FPGA-based query accelerator, especially optimized for sliding-window aggregate queries.

The first objective of the dissertation is to address the problem of out-of-order arrival of tuples and propose an alternative approach to implement a sliding-window aggregate query on an FPGA. To this end, we have proposed the WID-based implementation of an FPGA-based accelerator for sliding-window

aggregates over disordered data streams (Chapter 3). With the proposed approach, a sliding-window query can be implemented on an FPGA as an order-agnostic operator, which can process input tuples in their arrival order without sorting them into the "correct" order. The proposed accelerator utilizes punctuations, and this significantly reduces the input-to-output latency because there is no need to buffer and reorder incoming tuples. Our experiments demonstrate that nearly 6 million tuples can be processed per second directly from the network interface. To the best of our knowledge, this is the first work that proposes design and implementation of a punctuation-aware sliding-window aggregate operator on an FPGA device.

The second objective of the dissertation is to address the scalability problem and propose another approach to implement a sliding-window aggregate query on an FPGA in an *efficient* and *scalable* manner. To this end, we have proposed the pane-based hardware design of sliding-window aggregate operator and its implementation on an FPGA (Chapter 4). The proposed design adopts a two-step aggregation method using panes and supports disordered data arrival with punctuations. The proposed implementation is scalable with the increasing $\frac{RANGE}{SLIDE}$ ratio and significantly reduces the required logic elements by efficiently utilizing Block RAMs. Results show that the proposed implementation can achieve considerable performance improvement over the baseline implementation for large $\frac{RANGE}{SLIDE}$ ratios. To the best of our knowledge, this is the first work that proposes design and implementation of an FPGA-based sliding-window aggregate operator by using panes.

The third objective of the dissertation is to address the problem of the lack of run-time configurability. To this end, we have presented Configurable Query Processing Hardware (CQPH), a highly-optimized and minimal-overhead query processing engine for data streams (Chapter 5). CQPH can support multiple continuous queries with a large number of overlapping sliding-windows. The two-phase configuration approach provides the fully automated integration of CQPH and dynamic configuration mechanism for given queries. With CQPH-compiler, SQL-like queries can be easily configured into CQPH at run time. Evaluation results indicate that CQPH can deliver real-time response (in the order of microseconds) for high-volume data streams (over 150 million tuples per second). It is also indicated that CQPH provides linear scalability in terms of area with a constant clock rate. Finally, our experiments demonstrate wire-speed performance by directly manipulating the network packets.

## 6.2 Future Work

This work focuses on the sliding-window aggregate queries that process a single data stream. In particular, the current version of CQPH architecture can be configured to execute continuous queries that follow a certain pattern, and does not provide any support for those queries including join operation that process multiple data streams. Therefore, one direction for future work is to provide support for processing

multiple data streams. Specifically, sliding-window join queries should be considered to complement the proposed CQPH architecture.

Another direction for future work is to expand the proposed CQPH architecture into multiple FPGAs. Since the present work mainly focuses on using a single FPGA chip, the capability of the CQPH architecture is limited by the available on-chip resources of an FPGA device. In order to address the limitation, the proposed idea in this work can be expanded by using multiple FPGAs. Efficient use of off-chip memory resources should also be considered as an important future work in this context. This enables us to achieve further scalability and flexibility.

Finally, the proposed CQPH architecture should be thoroughly evaluated for a wide range of applications. In this work, the current prototype of CQPH architecture is evaluated on a single case study only. Since the proposed CQPH is categorized as a co-processor for stream processing systems, integration of the CQPH with a general-purpose software-based DSMS is required to conduct experiments with a variety of real applications and real data streams.

# Acknowledgement

I would like to express my sincere gratitude to my advisor Prof. Tsutomu Yoshinaga for his supervision throughout my study and research as well as for his patience, motivation and enthusiasm. His guidance helped me in all the time of research and writing of this dissertation.

Besides my advisor, I would like to thank the other members of my dissertation committee: Prof. Satoshi Kurihara, Prof. Tadashi Ohmori, Prof. Hiroshi Nagaoka, and Prof. Hiroyoshi Morita.

My sincere thanks also goes to Dr. Masato Yoshimi, Dr. Takefumi Miyoshi, and Dr. Hideyuki Kawashima for their encouragement, insightful comments, and helpful advices.

Last but not the least, I would like to thank my family, especially my parents for giving birth to me at the first place and supporting me throughout my life.

# References


[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the Borealis stream processing engine," in *CIDR*, 2005, pp. 277–289.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[3] Y. Ahmad and U. Çetintemel, "Data stream management architectures and prototypes," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 639–643.

[4] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," Stanford InfoLab, Technical Report 2004-20, 2004.

[5] W. G. Aref, "Window-based query processing," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 3533–3538.

[6] S. Babu, "Continuous query," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 492–493.

[7] J. bo Qian, H. bing Xu, Y. Dong, X. jun Liu, and Y. li Wang, "FPGA acceleration window joins over multiple data streams," *Journal of Circuits, Systems, and Computers*, vol. 14, no. 4, pp. 813–830, 2005.

[8] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *SIGMOD Conference*, 2003, pp. 647–651.

[9] C. Dennl, D. Ziener, and J. Teich, "On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library," in *FCCM*, 2012, pp. 45–52.

[10] C. Dennl, D. Ziener, and J. Teich, "Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration," in *FCCM*, 2013, pp. 25–28.

[11] e-trees.Japan, Inc., "e7UDP/IP IP-core,"

http://e-trees.jp/index.php/en/products/e7udpip-ipcore/ [Online; accessed 6-May-2014].

[12] EsperTech, Inc., "Esper Version: 5.1.0,"

http://www.espertech.com/esper/ [Online; accessed 26-Dec-2014].

[13] FIX Protocol, http://www.fixtradingcommunity.org/

[Online; accessed 5-Jan-2015].

[14] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Total," in *ICDE*, 1996, pp. 152–159.

[15] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation," in *SIGMOD Conference*, 2006, pp. 623–634.

[16] H. T. Kung, "Why systolic architectures?" *IEEE Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[17] H. T. Kung and P. L. Lehman, "Systolic (VLSI) arrays for relational database operations," in *SIGMOD Conference*, 1980, pp. 105–116.

[18] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings 1978 (Symposium on Sparse Matrix Computations, 1978)*. SIAM, 1979, pp. 256–282.

[19] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.

[20] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *SIGMOD Conference*, 2005, pp. 311–322.

[21] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *PVLDB*, vol. 1, no. 1, pp. 274–288, 2008.

[22] L. Liu and M. T. Özsu, Eds., *Encyclopedia of Database Systems*. Springer US, 2009.

[23] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. A. Vissers, "A low-latency library in FPGA hardware for high-frequency trading (HFT)," in *Hot Interconnects*, 2012, pp. 9–16.

[24] R. Mueller, J. Teubner, and G. Alonso, "Data processing on FPGAs," *PVLDB*, vol. 2, no. 1, pp. 910–921, 2009.

[25] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires - a query compiler for FPGAs," *PVLDB*, vol. 2, no. 1, pp. 229–240, 2009.

[26] R. Mueller, J. Teubner, and G. Alonso, "Glacier: a query-to-hardware compiler," in *SIGMOD Conference*, 2010, pp. 1159–1162.

[27] M. Najafi, M. Sadoghi, and H. Jacobsen, "Configurable hardware-based streaming architecture using online programmable-blocks," in *ICDE*, 2015, pp. 819–830.

[28] M. Najafi, M. Sadoghi, and H. Jacobsen, "The FQP vision: Flexible query processing on a reconfigurable computing fabric," *SIGMOD Record*, vol. 44, no. 2, pp. 5–10, 2015.

[29] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Flexible query processor on FPGAs," *PVLDB*, vol. 6, no. 12, pp. 1310–1313, 2013.

[30] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "An implementation of handshake join on FPGA," in *ICNC*, 2011, pp. 95–104.

[31] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "Design and implementation of a handshake join architecture on FPGA," *IEICE Trans. Info. & Syst.*, vol. E95-D, no. 12, pp. 2919–2927, 2012.

[32] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "Design and implementation of a merging network architecture for handshake join operator on FPGA," in *MCSoC*, 2012, pp. 84–91.

[33] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "A fast handshake join implementation on FPGA with adaptive merging network," in *SSDBM*, 2013, pp. 44:1–44:4.

[34] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "An efficient and scalable implementation of sliding-window aggregate operator on FPGA," in *CANDAR*, 2013, pp. 112–121.

[35] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "FPGA-based implementation of sliding-window aggregates over disordered data streams," *IEICE Technical Report*, vol. 112, no. 376, pp. 105–110, 2013, CPSY2012-74.

[36] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "Wire-speed implementation of sliding-window aggregate operator over out-of-order data streams," in *MCSoC*, 2013, pp. 55–60.

[37] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014, pp. 13–24.

[38] M. Sadoghi, H.-A. Jacobsen, M. Labrecque, W. Shum, and H. Singh, "Efficient event processing through reconfigurable hardware for algorithmic trading," *PVLDB*, vol. 3, no. 2, pp. 1525–1528, 2010.

[39] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen, "Multi-query stream processing on FPGAs," in *ICDE*, 2012, pp. 1229–1232.

[40] M. Sadoghi, H. Singh, and H.-A. Jacobsen, "Towards highly parallel event processing through reconfigurable hardware," in *DaMoN*, 2011, pp. 27–32.

[41] M. Stonebraker, U. Çetintemel, and S. B. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[42] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *VLDB*, 2003, pp. 309–320.

[43] Y. Terada, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "A consideration of window join operator over data streams by using FPGA," in *(in Japanese), IEICE Technical Report, RECONF2010-80*, 2011.

[44] J. Teubner and R. Mueller, "How soccer players would do stream joins," in *SIGMOD Conference*, 2011, pp. 625–636.

[45] J. Teubner, L. Woods, and C. Nie, "Skeleton automata for FPGAs: reconfiguring without reconstructing," in *SIGMOD Conference*, 2012, pp. 229–240.

[46] J. Teubner, L. Woods, and C. Nie, "*XLynx* - an FPGA-based XML filter for hybrid xquery processing," *ACM Trans. Database Syst.*, vol. 38, no. 4, pp. 23:1–23:39, 2013.

[47] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 555–568, 2003.

[48] P. Vaidya, J. J. Lee, F. Bowen, Y. Du, C. H. Nadungodage, and Y. Xia, "Symbiote: a reconfigurable logic assisted data stream management system (RLADSMS)," in *SIGMOD Conference*, 2010, pp. 1147–1150.

[49] L. Woods, Z. István, and G. Alonso, "Ibex - an intelligent storage engine with support for advanced SQL off-loading," *PVLDB*, vol. 7, no. 11, pp. 963–974, 2014.

[50] Xilinx, Inc., "Kintex-7 FPGA KC705 Evaluation Kit,"

http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html
[Online; accessed 4-October-2015].

[51] Xilinx, Inc., "Virtex-6 FPGA ML605 Evaluation Kit,"

http://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html
[Online; accessed 4-October-2015].

[52] M. Yoshimi, R. Kudo, Y. Oge, Y. Terada, H. Irie, and T. Yoshinaga, "Accelerating OLAP workload on interconnected FPGAs with flash storage," in *CANDAR*, 2014, pp. 440–446.

[53] M. Yoshimi, R. Kudo, Y. Oge, Y. Terada, H. Irie, and T. Yoshinaga, "An FPGA-based tightly coupled accelerator for data-intensive applications," in *MCSoC*, 2014, pp. 289–296.

# List of Publications Related to the Dissertation

The contents of Chapter 3 are partially based on the following published paper:

- **Yasin Oge**, Masato Yoshimi, Takefumi Miyoshi, Hideyuki Kawashima, Hidetsugu Irie, and Tsutomu Yoshinaga, "Wire-speed implementation of sliding-window aggregate operator over out-of-order data streams," In Proc. of the IEEE 7th International Symposium on Embedded Multicore/Many-core SoCs (MCSoC-13), pp. 55–60, 2013.

The contents of Chapter 4 are partially based on the following published paper:

- **Yasin Oge**, Masato Yoshimi, Takefumi Miyoshi, Hideyuki Kawashima, Hidetsugu Irie, and Tsutomu Yoshinaga, "An efficient and scalable implementation of sliding-window aggregate operator on FPGA," In Proc. of the First International Symposium on Computing and Networking (CANDAR'13), pp. 112–121, 2013.

The contents of Chapter 5 are partially based on the following published paper:

- **Yasin Oge**, Masato Yoshimi, Takefumi Miyoshi, Hideyuki Kawashima, Hidetsugu Irie, and Tsutomu Yoshinaga, "Design and evaluation of a configurable query processing hardware for data streams," *IEICE Trans. on Information & Systems*, Vol. E98-D, No. 12, pp. 2207–2217, 2015.