Master Thesis

Performance Benchmarking of Stream Processing Architectures with Persistence and Enrichment Support

Author(s):
Kaya, Ozan

Publication Date:
2011

Permanent Link:
https://doi.org/10.3929/ethz-a-006487719

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 14

Systems Group, Department of Computer Science, ETH Zurich

Performance Benchmarking of Stream Processing Architectures with Persistence and Enrichment Support

by

Ilkay Ozan Kaya

Supervised by

Prof. Dr. Nesime Tatbul

December 2010–May 2011
Abstract

Operational Business Intelligence requires real-time monitoring of business processes and activities. Common need of these applications is ensuring persistence and integrated access to both historical and live data. Persistence is vital because losing data in case of failures is intolerable for most of the business scenarios. Moreover, integrated access is essential for business intelligence since access to historical data is required for the enrichment of the incoming streams. Therefore, systems that integrate streaming functionality with database persistence are required. Integrated Stream Processing Systems are the solution for this need. MaxStream is a federated stream processing system developed at ETH Zurich that seamlessly integrates SPEs and databases. It follows a "DBMS-on-top" approach and extends SAP MaxDB system, providing SPE-DBMS integration to support continous queries with persistence and enrichment as well as SPE-SPE integration to support the interoperability of heterogeneous SPEs. As an alternative to the MaxStream approach, SPEs such as Coral8 follow an "SPE-on-top" approach, where the SPE is the main interface to the client and uses an external database for its persistence and enrichment needs. In this thesis, we have benchmarked the two different architectural approaches to Integrated Stream Processing. We have modified the TPC-H benchmark so that it is tailored down to a stream processing benchmark. After specifying the tables, rewriting the queries, and generating the data, we have tested the two alternatives and evaluated them through experiments. Our results showed that MaxStream outperforms Coral8 by at least a factor of four in terms of end-to-end latency.
I would like to express my sincere thanks to Prof. Dr. Nesime Tatbul for her constant support and guidance during the course of this thesis. I would also like to thank Roozbeh Derakhshani and Nihal Dindar from the ETH Systems Group for the valuable discussions and useful advice through the period of study.
# Contents

Abstract i

Acknowledgements ii

List of Figures vi

List of Tables vii

Abbreviations viii

1 Introduction 1
   1.1 Background 2
   1.2 Problem Statement 2
   1.3 Motivation 3
   1.4 Contributions and Outline 4

2 SPE-On-Top Architecture for Integrated Stream Processing 5
   2.1 Overview 5
      2.1.1 Architectural Details 5
         2.1.1.1 Scenarios for SPE-On-Top Architecture 6
      2.1.3 Why Coral8? 7

3 DBMS-On-Top Architecture for Integrated Stream Processing 9
   3.1 Overview 9
   3.2 MaxStream 10
      3.2.1 Architectural Overview 10
      3.2.2 The Federation Layer 11

4 Benchmark for SPE’s 14
   4.1 Evaluated Benchmarks 14
      4.1.1 TPC-C 14
      4.1.2 TPC-E 15
      4.1.3 TPC-H 15
   4.2 Why TPC-H? 15
   4.3 Modifications on TPC-H 15
# 4.3.1 Data Generation

4.3.2 Queries and Schemas

# 5 Experiment Setups

## 5.1 MaxStream

### 5.1.1 MaxStream Client Application

#### 5.1.1.1 DataFeeder Thread

#### 5.1.1.2 Monitoring Thread

### 5.1.2 MaxStream Setup Configuration

### 5.1.3 Sequential Case

#### 5.1.3.1 Persist Scenario

#### 5.1.3.2 Enrich&Persist Scenario

## 5.2 Coral8 Setup

### 5.2.1 Sequential

#### 5.2.1.1 Persist Scenario

#### 5.2.1.2 Enrich&Persist Scenario

### 5.2.2 Parallel

## 5.3 Comparison

# 6 Experiment Results

## 6.1 Parameter Tuning

### 6.1.1 MaxStream Client Application Parameters

#### 6.1.1.1 Input Rate

#### 6.1.1.2 Batch Size

### 6.1.2 Coral8 Studio Parameters

#### 6.1.2.1 Input Rate

#### 6.1.2.2 Batch Size

#### 6.1.2.3 Poll Interval

#### 6.1.2.4 Maximum Delay

#### 6.1.2.5 Alternative Databases

## 6.2 Comparison of the Two Systems

### 6.2.1 Selectivity

### 6.2.2 Latency of Subintervals

### 6.2.3 Scale Factor vs Latency

#### 6.2.3.1 Product Table with Index on ProductId Column

#### 6.2.3.2 Product Table without Index on ProductId Column

### 6.2.4 Input Rate

### 6.2.5 MaxDelay

### 6.2.6 Poll Interval

### 6.2.7 ODBC Connection

## 6.3 Coral8 - Parallel Setup Results

# 7 Conclusion and Future Work

## 7.1 Conclusion

## 7.2 Future Work
List of Figures

2.1 SPE-on-top Architecture ............................................. 6
3.1 DBMS-On-Top Architectures ......................................... 9
3.2 High Level View of MaxStream Architecture ..................... 10
4.1 TPCH schema .......................................................... 16
5.1 MaxStream - Setup ................................................... 17
5.2 MaxStream-Sequential-Persist: Detailed delay analysis of persist case ......................................................... 20
5.3 Maxstream-Sequential-Enrich&Persist: Detailed delay analysis of enrich&p persist case .................................................. 21
5.4 Coral8 - Setup ....................................................... 21
5.5 Coral8-Sequential-Persist: Detailed delay analysis of persist case ................................................................. 23
5.6 Coral8-Sequential-Enrich&Persist: Detailed delay analysis of enrich&p persist case .................................................. 24
5.7 Coral8-Parallel-Persist: Detailed delay analysis of parallel-persist case .......................................................... 24
5.8 Coral8-Parallel-Enrich&Persist: Detailed delay analysis of parallel-enrich&p persist case .................................................. 25
5.9 Sequential-Persist Case Comparison for Coral8 & MaxStream ................................................................. 25
5.10 Sequential-Enrich&Persist Case Comparison for Coral8 & MaxStream .......................................................... 26
6.1 MaxStream - Input Rate vs Latency ................................. 29
6.2 MaxStream - Batch Size vs Throughput ............................. 29
6.3 MaxStream - Batch Size vs Latency ................................. 30
6.4 MaxStream - Throughput vs Latency ............................... 30
6.5 Coral8 - Input Rate vs Latency .................................... 31
6.6 Coral8 - Batch Size vs Throughput ................................. 32
6.7 Coral8 - Batch Size vs Latency .................................... 32
6.8 Coral8 - Throughput vs Latency ................................... 33
6.9 Coral8 - Poll Interval vs Latency ................................... 33
6.10 Coral8 - Max Delay vs Latency ................................... 34
6.11 MaxStream - Selectivity vs Latency ............................... 37
6.12 Coral8 - Selectivity vs Latency ................................. 37
6.13 Coral8 - Latency of Subintervals .................................. 39
6.14 Coral8 - Latency of Subintervals .................................. 39
6.15 MaxStream - Scale Factor vs Latency ............................ 41
6.16 Coral8 - Scale Factor vs Latency ............................... 41
6.17 Coral8 - Scale Factor vs Latency (no index) ..................... 42
6.18 Coral8 - Parallel Setup Experiment Results .................... 44
List of Tables

6.1 End-to-End Latency Results for Databases Tested .......................... 35
6.2 Threshold Values for Selectivity Ratios ................................. 38
6.3 Latency of Subintervals ................................................. 40
6.4 Latency of Subintervals ................................................. 40

B.1 MaxStream - Input Rate vs Latency, Figure 6.1 ......................... 59
B.2 MaxStream - Batch Size vs Throughput, Figure 6.2 ..................... 59
B.3 MaxStream - Batch Size vs Latency, Figure 6.3 ......................... 60
B.4 MaxStream - Throughput vs Latency, Figure 6.4 ......................... 60
B.5 Coral8 - Input Rate vs Latency, Figure 6.5 ............................. 60
B.6 Coral8 - Batch Size vs Throughput, Figure 6.6 ......................... 60
B.7 Coral8 - Batch Size vs Latency, Figure 6.7 ............................ 61
B.8 Coral8 - Throughput vs Latency, Figure 6.8 ............................ 61
B.9 Coral8 - Poll Interval vs Latency, Figure 6.9 ............................ 61
B.10 Coral8 - Max Delay vs Latency, Figure 6.10 .......................... 61
B.11 MaxStream - Selectivity vs Latency, Figure 6.11 ....................... 61
B.12 Coral8 - Selectivity vs Latency, Figure 6.12 .......................... 62
B.13 Coral8 - Latency of Subintervals, Figure 6.13 ......................... 62
B.14 MaxStream - Latency of Subintervals, Figure 6.14 ..................... 62
B.15 MaxStream - Enrich-Persist- Scale Factor vs Latency, Figure 6.15 ... 62
B.16 Coral8 - Enrich-Persist- Scale Factor vs Latency, Figure 6.16 ....... 62
B.17 Coral8 - Scale Factor vs Latency (no index), Figure 6.17 ............. 62
B.18 Coral8 - Parallel Setup Experiment Results, Figure 6.18 ............. 63
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISP</td>
<td>Integrated Stream Processing</td>
</tr>
<tr>
<td>SPE</td>
<td>Stream Processing Engine</td>
</tr>
<tr>
<td>TPC</td>
<td>Transaction Processing Performance Council</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Values</td>
</tr>
<tr>
<td>ACID</td>
<td>Atomicity Consistency Isolation Durability</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
</tbody>
</table>
Dedicated to my family...
Chapter 1

Introduction

It is a well acknowledged fact that databases are essential for most of the business applications. In this thesis, we focus on a subset of stream processing architectures where databases are integrated to the systems. Databases are used mainly for two reasons in Stream Processing. First is to persist the stream data. Both input and output streams are needed to be persisted in most of the scenarios for fault tolerance. Secondly, access to historical data from database tables are required for business intelligence. For instance, streams are processed according to previously stored data from database or incoming streams are enriched with some static database tables. For all these scenarios, integrated stream processing systems are crucial.

There are two different approaches for integrated stream processing architectures. First is where the stream processing engine is the main actor and interface to the client (applications, data sources etc.). SPE receives the data from the client and returns the result back after processing it. Moreover, SPE connects to the database when it needs to persist the data or needs to retrieve historical data from database. Second approach is database management system on top architecture. DBMS is the interface to the client and retrieves the stream of data. According to business logic either persists it to the database or redirects it to the SPEs integrated underneath.

Both of the approaches have their own advantages and drawbacks. In this thesis, we will use a modified version of an industry standard benchmark to compare the performance of the two architectures described above where persistence of input and output is a requirement in all scenarios.

This chapter provides an introduction to Stream Processing Systems. Section 1.1 provides the background and motivation for the need of Stream Processing Systems. Section
1.2 provides the detailed problem statement. A summary of the contributions of this thesis and outline of the report are provided in Section 1.4.

1.1 Background

Stream Processing is a field of research in database systems. This field emerged due to the need of processing data on the fly rather than “Store-and-Pull” model of traditional data processing systems [1].

Streaming applications require processing of large volumes of highly dynamic, time-sensitive continuous streams of data. An example can be sensor-based monitoring devices which are used variable tolling of urban express ways. In order to prevent traffic congestion, drivers will be charged higher amounts during busy hours. Linear Road Benchmark [2] simulates such a scenario and shows that stream data management system outperforms a relational database on streaming data applications.

MaxStream federated stream processing architecture has been proposed by ETH Zurich as a solution to both heterogeneity and stored-streaming challenges [3]. MaxStream introduces a federation layer between client applications and stream processing systems (SPEs and database engines). Federator encapsulates the technical details of the underlying systems and hides it from the application. Application communicates with MaxStream and submits queries to it. MaxStream performs global optimizations\(^1\) and necessary translations to the interfaces of underlying systems.

1.2 Problem Statement

Dominating Problems Although several academic and commercial stream processing engines are available today, developing and maintaining stream-based applications are difficult. The two dominating reasons are mentioned in studies [3, 4] conducted in ETH Zurich in cooperation with SAP Labs. First is the lack of standards. Since the requirements of each application are different, SPEs have also different data and query models which result in some organizations using multiple SPEs. In order to overcome this heterogeneity problem, standardization initiatives [5] started. Second problem is integrated access to both stored and streaming data. Most of the SPEs support this integration via connections to external DBMS engines. However this type of SPE-DBMS integration is thought and added later to the architecture. Therefore, it is not a tight integration and has a limited support.

\(^1\)Proposed and designed but not yet implemented in the first prototype of MaxStream.
Where do we need integrated stream processing? Business Intelligence (BI) is needed for better decision making process. It comes with mining data or facts and providing information about the historic and current views of business operations which is essential for fraud detection, inventory management and supply-chain optimization. Common need of these applications is ensuring persistence and integrated access to both historical and live data. Persistence is vital because losing data in case of failures is intolerable for most of the business scenarios. Moreover, integrated access is essential for business intelligence since access to historical data is required for the enrichment of the incoming streams. Therefore, systems that integrate streaming functionality with database persistence are deserved.

Performance needs The critical requirement for real-time business intelligence applications [6, 7] require dealing with high data volumes, data rates and have low latency requirements. Moreover, due to industry regulations there might be a requirement for persisting all inputs and outputs in order to prevent loss of events. All these requirements show that we need to answer the following questions:

- Which approach is better for the business requirements? SPE on top or DBMS on top?
- What are the limits of the stream processing engines? Where are the bottlenecks?
- When we compare the two alternative approaches to SPE-DBMS integration using MaxStream and Coral8, what will be the performance difference in terms of different performance metrics: end-to-end latency, input rate, throughput.

1.3 Motivation

Streaming data source integration, SPE-SPE integration, and SPE-DBMS integration are different techniques for integrated stream processing. However, as we will discuss in Chapter 2, all these approaches have different limitations such as optimization and integration issues. Therefore, we decided to compare it with an alternative method which is proposed by ETH Zurich: creating a federation layer in order to integrate a set of SPEs and databases. In this thesis, our motivation is to test our prototype with an industry standard benchmark which is extended for streaming and prove that it outperforms other approaches by far in terms of different performance metrics.
Chapter 1. Introduction

1.4 Contributions and Outline

The main steps that summarize the work done during the course of the thesis are the following:

First, we provided a detailed description of the architecture of Stream Processing Systems. SPE on top and DBMS on top approaches are discussed in details. Later we have examined the available TPC benchmarks in order to test our systems. We modified TPC-H benchmark so that it is compatible with streaming scenarios. We ran experiments on both architectures and demonstrate the feasibility and performance of MaxStream in terms of end-to-end latency, input rate and throughput. Via these reproducible and verifiable results, we proved that DBMS on top approach (MaxStream) performs better compared to SPE on top approach where data persistence is a requirement. In other words, it is better in cases where input should not be processed or output should not be returned back to client before their persistence is ensured.

The rest of the thesis report is organized as follows:

- **Chapter 2** gives a background and a short overview of the SPE based approach where Stream Processing Engines are the main interface to the client.

- **Chapter 3** explains the system architecture and components of MaxStream which is an alternative approach with DBMS on top and interface to the client.

- **Chapter 4** provides an overview of the benchmark used for the experiments. It also includes an overview of the queries used for the experiments.

- **Chapter 5** elaborates on the experiment setups for both MaxStream and Coral8. Explains the system architecture and components involved.

- **Chapter 6** presents the results of all experiments done and the performance analysis performed on Coral8 and MaxStream.

- **Chapter 7** covers the conclusion and future work ideas.
Chapter 2

SPE-On-Top Architecture for Integrated Stream Processing

This chapter gives background information and a short overview of the SPE based approach where Stream Processing Engines are the main interface to the client.

2.1 Overview

We have described the need for integrated streaming systems in Chapter 1. First approach we explained for SPE-DBMS integration is SPE-on-top architectures. Stream processing engine is the gateway for the client application and it communicates with the database underneath to persist the output or to enrich the incoming tuples with static tables. SPE gets the stream from the client application using the input adapters and returns back the results via output adapters. The connection between SPE and the DMBS is provided with ODBC or JDBC.

2.1.1 Architectural Details

High level view of an Integrated Stream Processing System with SPE-on-top Architecture is illustrated using Figure 2.1.

Stream processing engine is the interface to the clients. Here, clients can be applications that provide input or programs which monitor the outputs. Client can also be a file containing the input stream as comma seperated values or a database table. SPE has input and output adapters for each of these types. Streams are created from any of these sources via adapters. After the creation of streams, SPE can decide what to do
with the incoming tuples according to the application logic. It can persist the input in
the database or enrich the stream with historical data from database. Alternatively it
can start processing the stream directly without interacting with the database. This is
also valid for the output stream. SPE can dump the resulting stream into the database
before providing it to the client via output adapters. There are alternative ways for
client to receive the output. It can listen to a port of the SPE. Rather it can get the
results from a database table or a text file with comma seperated values.

2.1.1.1 Scenarios for SPE-On-Top Architecture

In this thesis, we consider the scenarios with persistence. Therefore, the created streams
are persisted in the database first. The connection between the SPE and database can
be established via JDBC or ODBC interface. After the persistence is guaranteed, the
streams are processed inside the SPE. We have two different options at this point. First
is persistence only scenario. The stream is processed according to the continuous query
directly. However, in persist&enrich scenario, we enrich the stream with static database
table before processing with the continuous query. In the next step, we dump the output
stream into the database. Lastly, the results are sent to the client.

2.1.2 Available Stream Processing Systems [1]

As we can see from Figure 2.1, SPE-On-Top architectures in integrated stream processing
can be composed of different pairs of SPEs and DBMSs. Therefore we have examined
the following Stream Processing Engines:

- **STREAM** Its SQL based language model constitutes the ground for many com-
  mercial systems. STREAM’s query language is an extension to SQL:1999. It adds
“stream” datatype to the model. “Stream-to-relation” and “relation-to-stream” operations are introduced as well. Notion of time is brought to continuous query execution. However, STREAM doesn’t allow integrated access to streams and relations.

- **Aurora/Borealis** It is built on Berkeley DB. Allows table operations\(^1\) with the arrival of a stream tuple. This is the same approach with Coral8 [8] and StreamBase [9].

- **TelegraphCQ**[10] Is an extension to PostgreSQL relational engine. It is a “stream-relational” system which means it runs SQL queries continuously over data before it gets stored in the database.

- **DataCell** Extends MonetDB relational database for stream processing [11]. Stream tuples are accumulated and are accessed by continuous queries in a periodic fashion.

- **Dejavu** provides declarative pattern matching techniques over live and archived streams of events [12]. It extends MySQL relational database engine. Both streaming and historical data sources can be attached into a common query engine.

### 2.1.3 Why Coral8?

Among all the other SPEs, we have decided to use Coral8 Stream Processing Engine due to following advantages:

- **Poll Adapter** We have used this adapter in order to create streams from database tables. This is essential for designing a sequential setup. The tuples will be persisted into the database first and will be retrieved from there to create the stream.

- **Maximum Delay** In order to process the tuples without waiting for the consecutive ones, we need an option in SPE. Maximum Delay is a time-out value that initiates the execution. It is important for ensuring the lowest end-to-end latency possible.

- **Database Support** Coral8 has support for integrating any database using ODBC driver of the selected product. This gives us the opportunity to test our system with alternative databases and run the experiments with the best performing database.

\(^{1}\)selection, insertion, deletion and update
• **Detailed Tuning** Using the “services.xml” file, we can tune CoralS server parameters which are described in Section 5.2. These are important parameters for exact measurements of system under test.
Chapter 3

DBMS-On-Top Architecture for Integrated Stream Processing

This chapter introduces the alternative approach to integrated stream processing with DBMS on top and interface to the client. System architecture and components of MaxStream will be explained.

3.1 Overview

General architecture of DBMS-On-Top approach is shown in Figure 3.1. This approach is a layer between client applications and a collection of SPEs and databases. It integrates multiple SPEs with traditional databases behind a common SQL-based declarative query interface and a common API.

Figure 3.1: DBMS-On-Top Architectures
The federation layer of DBMS is on top and interface to client applications. Client communicates with the DBMS federation layer and is totally isolated from the rest of the system. For instance, client sends the tuples and queries to federation layer. According to the application logic queries are sent to the relevant SPE or the input is persisted in database. Client is isolated from the underlying systems. Although the underlying systems are heterogeneous and differs from each other, it doesn’t need to know the syntax of each SPE’s query language. Federation layer will translate and optimize it accordingly.

3.2 MaxStream

3.2.1 Architectural Overview

Figure 3.2 shows the high level view of MaxStream Architecture. As one can observe from this figure, MaxStream is a middle layer between client applications and the SPE and database underneath. It has its own query language which is very similar to SQL syntax and programming interface to client applications. The middle layer, which is called Federation Layer as well, has wrappers for SPE and database integrated to MaxStream. Federation Layer performs global optimizations\(^1\) and then wrappers translate queries to native interfaces of the underlying systems. Moreover, the query model used in MaxStream is developed after the standardization efforts [5, 13]. Current prototype can handle basic SQL queries and time-based windowing but it is extendable for supporting other types of windows and streaming constructs.

\(^1\)Proposed and designed but not yet implemented in the first prototype of MaxStream.
MaxStream prototype is implemented on top of MaxDB which is a federated relational database architecture. This means support for SQL, persistence, transactions and other database functionalities exist already. Commercial stream processing engines mostly work with SQL-based query languages so translating queries into the integrated SPEs is easier. Persistence is one of the default functionality provided with relational databases which is also a requirement for business applications. MaxStream prototype has long term persistence support which is important for stream-store integration.

Design and implementation details for MaxStream will be discussed further in Section 3.2.2. We will concentrate on the critical need for a single application which is capable of input and output persistence in addition to stream processing. We want to show that federation architecture is suitable for meeting these needs and feasible for seamless integration between a persistent store and a stream engine without performance degradation.

### 3.2.2 The Federation Layer

MaxStream is extended from SAP’s MaxDB database federator. It has additional ability to federate stream data. A wrapper has been implemented for communication with Coral8 Stream Processing Engine. We will describe the Federation Layer Components in this section. We refer the reader to the technical report for more detailed information about design and implementation of MaxStream [4].

**+EVENT** The queries that are submitted by the client application are either persistent or transient. The data related to a continuous query will be recognized by MaxStream and pushed down to the SPE. These queries will be identified by the hint “+EVENT”.

**Federation** MaxStream’s federation layer has been built on top of SAP’s MaxDB so it is using MaxDB’s existing federation features. For instance, MaxDB can perform distributed joins over multiple external database sources. Thanks to federator technique, a new database can be added to MaxDB without any effort which is important for scalability of SAP applications. MaxStream federator has the same architecture with extended input-output data streaming mechanisms, query parsing and translation for continuous queries.

**SQL Parser** MaxStream Federator Language is an extension of SQL. It has support for continuous queries with windowing. Applications can read from and write into streams. Streams can be joined with static tables. There is also an effort for creating formal query model [5, 13].
Continuous Query After parsing the continuous queries, they are sent to the query optimizer module. If necessary, queries are rewritten and passed to Query Executer. Streaming queries are pushed down to SPEs and all others are executed in MaxStream.

SQL Dialect Translator In MaxStream, MaxDB’s translator has been extended for translating the execution plans into streaming SQL dialect of the underlying SPEs. It is capable of taking any complex query plan and decompile it into the form that SPE can receive.

Data Agent It refers to the communication bridge between MaxStream and SPEs underneath. Control messages are exchanged and input streams are forwarded to SPE via Data Agent. However, output streams are passed back to MaxDB tables using an ODBC connection, they are not passing through Data Agent. So this bridge is for one way communication, from MaxStream to the SPE. If we want to integrate a new SPE to MaxStream, we should build a Data Agent for that streaming engine.

ISTREAM Streams are created from user input. Here user may refer to client application. Most of the business applications require no event loss under any circumstances. Therefore MaxStream uses the ISTREAM “relation to stream operator” for creating streams from static tables. Client application sends the tuples to the database with periodic insert statements. After commit is performed by the client, all the newly inserted tuples are copied into the in-memory table and assigned timestamp values. The results are forwarded to the relevant SPE for processing. Tuples which are sent to SPE are removed from the in-memory table. Following is an example of ISTREAM use:

```sql
INSERT INTO STREAM OrdersStream
  SELECT ClientId, OrderId, ProductId, Quantity
FROM ISTREAM(OrdersTable);
```

Streams are created from the tuples persisted in OrdersTable. Whenever a new tuple arrives, ISTREAM gets the columns “ClientId”, “OrderId”, “ProductId”, and “Quantity” and inserts it into the temporary in-memory table. Stream data is created and sent to SPE from there.

Transient Events If persistence is not a requirement for input tuples, MaxStream can use the in-memory tuple mechanism. Its advantage is higher data rates with lower latency. However, input tuples might not be processed and get lost in case of a failure in the system.

Monitoring SELECT The output of Stream Processing Engines underneath MaxStream must be returned back to the client. However, client interfaces are not pushed based like SPEs, they are pull based. In order to overcome this issue several alternatives are proposed, such as monitoring by client application, periodic select queries,
database triggers and ISTREAM to client applications. However they were either cumbersome or error-prone. Or they require re-implementing same application logic for each client. Therefore “monitoring select mechanism” is preferred. It is inspired by blocking select mechanism of ANT’s Data Server [14]. Basically, stream’s output table is monitored and server returns back to the client application whenever it finds a row. This way, client doesn’t need to poll the database periodically. Being blocked till a tuple is found is more efficient than continuously querying. Moreover, it doesn’t require any change in the client application. Following is an example of MONITORING SELECT use:

```sql
SELECT *
FROM /*+ EVENT */ TotalSalesTable
WHERE TotalSales > 500000;
```

Client is subscribed to TotalSales table. Whenever a new tuple with a “TotalSales” value higher than 500,000 is inserted into the table, server pushes the tuple to the client.
Chapter 4

Benchmark for SPE’s

4.1 Evaluated Benchmarks

In order to evaluate the performance of MaxStream and Coral8 we examined the available Stream Data Management Benchmarks. Linear Road was one of them. It tests Stream Data Management Systems (SDMS) by measuring how many expressways a SDMS can support by giving accurate and timely results to four types of queries which are “Toll Notification”, “Accident Notification”, “Account Balance Query”, and “Daily Expenditures Query”. However, Linear Road is not designed for evaluating Integrated Stream Processing Architectures. There is no explicit stream persistence and enrichment involved in this benchmark.

Therefore we need to have a use case for our experiments where ISP is needed. It is Operational Business Intelligence where real time monitoring of business processes and activities are used to detect and identify interruptions and bottlenecks [15]. We decided to use one of Transaction Processing Performance Council’s (TPC) benchmarks and considered the following standards:

4.1.1 TPC-C

The TPC Benchmark C [16] is an on-line transaction processing (OLTP) benchmark. It has multiple transaction types, complex database and execution structure. The transactions are executed on-line or queued for deferred execution. TPC-C’s performance is measured with tpm-C metric which is the number of New-Order transactions executed per minute. Since these transactions simulates a complete business activity, tpm-C metric is a measure of business throughput.
4.1.2 TPC-E

The TPC Benchmark E [17] is an on-line transaction processing benchmark that simulates the OLTP workload of a brokerage firm. Its performance metric is transactions per second (tps).

4.1.3 TPC-H

The TPC Benchmark H [18] is an ad-hoc, decision support benchmark. This benchmark simulates systems that work with large volumes of data and execute complex queries. Its performance metric is TPC-H Composite Query-per-Hour (QphH@Size) and shows the query processing power against the selected database size.

4.2 Why TPC-H?

Among all these standards, we have chosen the TPC-H. To justify our decision, we need to explain our simulation scenario. We want to benchmark a company which receives orders continuously and wants to detect the important orders which differ from others according to a columns value (total amount of order, product type etc.). In our scenario, we also need historical data because the orders received will be enriched using static tables from the database. Lastly, orders received and the static tables must be scalable. In other words, these tables must be populated according to a scale factor and their sizes must be directly proportional for a fair comparison. TPC-H supports all the requirements we have where an operational business intelligence scenario can be applied to.

4.3 Modifications on TPC-H

As one can observe from the figure 4.1 although TPC-H has the best structure for Stream Processing Benchmark, it still needs to be tailored according to our needs. The orders table will be used for continuous input from the client. Lineitem table is merged with orders table and formed the new orders table. Part and partsupp are merged to constitute products table. PARTKEY-SUPPKEY are appended and called product id to create the foreign key between these two newly formed tables (orders_table and products_table). The other tables are kept same.

SPE Adapted version of TPC-H has an extended orders_table with a foreign key column called product_id which points to a unique product in products_table.
4.3.1 Data Generation

We have used the “dbgen” which is a database population program for use with the TPC-H benchmark to populate the database tables. This tool generates the data according to a given scale factor. For our experiments we have generated a wide range of data from scale factor 0.1 to scale factor 10. Then we applied the modifications mentioned in section 4.3 for each scale of data.

4.3.2 Queries and Schemas

We have modified the Large Volume Customer Query (Q18) of TPC-H benchmark for our experiments. The Large Volume Customer Query ranks customers based on their having placed a large quantity order. Large quantity orders are defined as those orders whose total quantity is above a certain level. Instead of defining a certain quantity threshold, we have retrieved all the results and ordered them according to either “ProductId” or “ProductName”.

SQL Queries and database schemas for Coral8 and MaxStream can be checked from Appendix A.
Chapter 5

Experiment Setups

This chapter elaborates on the experiment setups for both MaxStream and Coral8. Explains the system architecture and components involved.

5.1 MaxStream

Figure 5.1 illustrates the MaxStream setup which is used for the experiments in this study.

System Under Test (SUT) starts when the TPC-H orders data is read from text file by the client. It ends when client retrieves the results from MaxStream Federation Layer.
and writes them to the result file. The components of SUT will be explained in following sections.

5.1.1 MaxStream Client Application

Client application has been implemented in C++. It has two threads. One is responsible for feeding the input to the system. Other monitors the results getting out of the system. All the parameters are passed to the application using a configuration file, therefore different experiments are done without recompiling the source code with related parameters.

5.1.1.1 DataFeeder Thread

DataFeeder is responsible for sending tuples with a rate defined in configuration file. The order tuples are generated according to the modified version of TPC-H as described in Section 4.2. Before sending the orders, a timestamp is placed inside every tuple for latency calculations. If the batching option is enabled, commit performed after sending a batch of tuples to the database. Commit refers to making a set of tentative changes permanent and visible to other users (SPEs). The cost of commit operation and the bottleneck that it causes will be discussed further in section 6.1.1.2.

DataFeeder thread’s configuration file has the following parameters:

- Input&Output Method: DataFeeder thread has three different options for input and output method; direct, persist and non-persist. In our experiments, we have used persistence for both methods.

- Datasource: Instead of a database table, we preferred to read from CSV file because the systems we are testing have different internal representations for same data types.

- Module Definition: The queries that will be used for the experiments are passed to the application using the module definition file. Its directory is in the configuration file so we can easily change the experiment type from persist case to enrich&persist case.

- Stream Processing Engine: The stream processing engine used underneath MaxStream and the configuration details (such as host address, port, username and password etc..) for this engine are listed in this section.
Chapter 5. Experiment Setups and Tuning

- **Think Time**: The delay between two tuples sent to the system. Via this parameter, we determine the input rate for MaxStream. Can be set to zero for maximum input rate.

- **Run Time**: Can be called timeout value as well. Experiment will be terminated if run-time value exceeded. Can be set to zero for unlimited run time.

- **Batch Size**: The number of tuples sent to system before performing a commit operation. Default and minimum value is 1.

### 5.1.1.2 Monitoring Thread

This thread is responsible for receiving the tuples using the Monitoring Select Statement. Only the new tuples are retrieved from the database. This is the key advantage of MaxStream compared to other Stream Processing Engines. Instead of querying the database table with a constraint on the where clause, we use monitoring select to retrieve the new tuples.

In each tuple, insertion timestamp is available. Using the time of retrieval, end-to-end latency is calculated and written into the result tuple which is sent back to the client. This tuple not only has the start and end time information but also includes other timestamps taken after each subinterval for further analysis (See 6.2.2).

### 5.1.2 MaxStream Setup Configuration

Since MaxStream is built on top of MAXDB, we need to enable streaming property first:

```
param_directput EnableStreamProcessing "YES"
```

Then we need to add volume to the data and log file. This is important for running the experiments long enough:

```
param_addvolume 1 DATA ETHD0001 F 600000
param_addvolume 1 LOG ETHL0001 F 600000
```

Lastly, we can activate the system:

```
db_activate
```

Since MaxStream architecture allows only persistent data, we will examine solely sequential case.
5.1.3 Sequential Case

The term sequential refers to the flow of stream data through our system. It means the tuples will move from one consistent state to another and will not be processed by two different parts of the system at the same time. This is important since we want to provide ACID properties.

5.1.3.1 Persist Scenario

The tuples received from the client application are persisted in the database first. The table used is called “OrdersTable”. Then they are retrieved from orderstable by ISTREAM operation as described in chapter 3. Retrieved tuples are passed to the underlying SPE which is Coral8 in our case. The tuples are used to feed the ”ordersstream” inside Coral8. Input stream is processed according to the query which is summation over quantity column in our case. The results are fed into ”totalordersstream” which is eventually used to populate ”TotalOrdersTable” in the database. Monitoring thread selects the newly inserted rows from the database using ”MONITORING SELECT” query and returns them back to client.

Figure 5.2: Maxstream-Sequential-Persist: Detailed delay analysis of persist case

Measurement details, exact timestamp positions and latency intervals can be observed from the figure 5.2.

5.1.3.2 Enrich&Persist Scenario

This scenario is slightly modified version of persist case. We have a static table called ”productstable” in our database. The ISTREAM operation will be used to retrieve tuples from ”orderstable” by enriching them with ”productstable”. It is an extra join operation over the ”productId” column.
Chapter 5. Experiment Setups and Tuning

21

INPUT
OrdersTable
ISTREAM(OrdersTable) &
JOIN with ProductsTable:
Create OrdersStream
TS_2
TS_1
SUM(Quantity):
Create TotalOrdersStream
TS_3
Insert into DB
Insert into DB
TS_4
TotalOrdersTable
MONITORING SELECT
Write .csv file
TS_5
OUTPUT

ENRICH & PERSIST SCENARIO:

MAXSTREAM FLOW DIAGRAM:

Interval 1: Input Persistence & Query Feed From Persisted Input
Interval 2: Query Processing
Interval 3: Output Persistence
Interval 4: Client Feed From Persisted Output

Figure 5.3: Maxstream-Sequential-Enrich&Persist: Detailed delay analysis of enrich&persist case

Measurement details, exact timestamp positions and latency intervals can be observed from the figure 5.3.

5.2 Coral8 Setup

Figure 5.4 illustrates the Coral8 setup which is used for the experiments in this study.

Figure 5.4: Coral8 - Setup

System Under Test starts when the TPC-H orders data is read from text file by Coral8 Studio. It ends when the server writes the results to output file.

Except for the relatively unimportant parameters (username, password, port, etc..), we have tuned Coral8 Studio, which is used as the client application for the Coral8 Server, with the following settings:

- MaxCallExecutionTime: Defines the maximum time that a database call is allowed to be executing. Any value greater than 0 will cause a termination of any request taking longer than MaxCallExecutionTime to execute. We set this value to 0 for unlimited execution time.
• CacheMaximumAge: CacheMaximumAge is the time one would like fetched rows to remain cached. If an identical DB request with the same 'parameters' arrives within the age, the data will be retrieved from cache rather than the DB, greatly improving performance in many cases. However, this property is not desired in our experiments because we want to measure the end-to-end latency for each order sent to the system. If some of the tuples are retrieved from the cache instead of database, we won't have exact measurements. So we set it to 0 for no cache.

• CacheMaximumSizeBytes: The maximum size that an individual DB subquery cache can hold. Cache size is defined as the sum of the size of the results obtained from the DB subquery and the input tuple that triggered the query. Like CacheMaximumAge parameter, we set this option to 0 in order not to use caching property.

• MaxPoolSize: Sets maximum size of the DB Connection pool per server. We wanted to setup one connection and use it for all further database requests. So the pool size in our experiments is one.

• MaxParallelInvocationsPerCall: Defines the maximum number of concurrent executions for a DB subquery. A value greater than 1 enables concurrency. Since we want to utilize the same connection from the pool, we used 1 for this parameter.

• DBWriteMaxBatchSize: Defines the maximum number of DB writes for a DB execute statement. Since for each request we want to measure the minimum end-to-end latency, we preferred 1 for batch-size.

• DBReadNoCommit: Coral8 automatically issues a COMMIT after every executed DB statement, regardless of whether this is a read, or a write. We have disabled COMMITs for DB Reads (subqueries) only because committing causes unnecessary delay after retrieving data from database where no change is made on the tables.

• EnableTracing: If enabled, logs all DB requests for the service to the Server Log as Info Level messages. We have used this option for debugging only because it results in performance degradation.

5.2.1 Sequential

We have mentioned the sequential experiment setup for MaxStream in section 5.1.3. We have the same requirements for Coral8. In order to ensure the sequential flow, we divided the Coral8 projects into 3 sub-projects inside Coral8 studio. Each sub-project
consist of three steps: reading the tuples and creating the input stream, processing the stream with some logic and lastly writing the results in a database table. The data flow between these sub-projects are handled with database poll adapters. By this way, we ensure the sequential flow of data. None of these steps can work parallel, each step requires data from the previous one.

Between these three sub-projects there are two connections with poll adapters. These adapters poll the database with the frequency defined in poll internal parameter. We set it to 1 millisecond.

5.2.1.1 Persist Scenario

As one can see from the figure 5.5, we have divided the flow of the data into four intervals. First interval starts when the client sends the tuples to the system (TS_1) and ends with the creation of stream using the received tuples (TS_3). Second interval is the period between creation of the stream (TS_3) and end of query processing (TS_4). Third interval is the output persistence from timestamp 4 to 5. Last interval is similar to first internal and includes stream creation using poll adapter and output persistence which is between timestamp 5 and 7.

![Persist Flow Diagram](image)

**Figure 5.5:** Coral8-Sequential-Persist: Detailed delay analysis of persist case

5.2.1.2 Enrich&Persist Scenario

Enrich&Persist scenario (figure 5.6) and its time intervals are almost same with Persist case (5.2.1.1). The first interval is not only polling but also enrichment of the data using static database table (ProductsTable). Tuples are enriched with productname information. Further in the second interval, query processing is done using the productname column of the stream. The rest is the same with the persist scenario.
Chapter 5. Experiment Setups and Tuning

5.2.2 Parallel

The term parallel refers to the flow of stream data through our system. It means the tuples can be processed by two different parts of the system at the same time. Parallel flow can be seen from the detailed measurement setup for parallel scenario in figures 5.7 and 5.8.

Input is read from the file and sent directly to both orderstable and stream processor. Stream processor doesn’t wait for the persistence in the database and starts processing the tuple immediately. Summation on the quantity column for the current window is calculated for each tuple and result stream is created. Result stream is sent to totalordersstable and written into the output file parallelly. End time is either the insertion to the database or returning the output back to the client. End-to-end latency is calculated using the following formula:

\[ \text{EndtoEndLatency} = \max(\text{db.timestamp}, \text{text.timestamp}) - \text{arr.timestamp} \]

\[ \text{EndtoEndLatency} = \max(\text{TS}_2, \text{TS}_5, \text{TS}_6) - \text{TS}_1 \]
5.3 Comparison

In order to compare Coral8 and MaxStream, we designed the sequential setups similarly. Although MaxStream architecture does not allow us to insert time measurement hooks as detailed as Coral8, we have matched the timestamps so that subintervals will be comparable. For the persist scenario, figure 5.9 shows the details of the measurement.

MaxStream has 5 timestamp hooks and Coral8 has 7. The corresponding timestamps are shown with dotted lines. In the first interval, MaxStream’s ISTREAM and Coral8’s POLL operations are correspondent since they are responsible for retrieving data from database. It is also valid for the fourth interval where MaxStream’s MONITORING SELECT and Coral8’s POLL are matching.

Enrich&Persist case is examined in figure 5.10.
Chapter 5. Experiment Setups and Tuning

Chapter 5. Experiment Setups and Tuning

Figure 5.10: Sequential-Enrich&Persist Case Comparison for Coral8 & MaxStream
Chapter 6

Experiment Results

This chapter starts introducing the performance metrics of the experiments. We start with parameter tuning and setup for individual systems. After finding the best configuration, we have benchmarked MaxStream and Coral8 for different scenarios. The results of the experiments are discussed in the last section of the chapter.

6.1 Parameter Tuning

We have two goals while running the experiments. First is to maximize the throughput which is the maximum number of tuples the system can process per time unit. Secondly, minimizing the end-to-end latency which is the interval between the time a row arrives to be processed by an input adapter and the time it is completely processed by an output adapter. The detailed analysis for measuring end-to-end delay is described in chapter 5. We aim the highest throughput while keeping the end-to-end latency as low as possible. Therefore, following parameters are tuned for the best experiment setups:

- Input Rate
- Batch Size
- Poll Interval
- MaxDelay
- Alternative Databases

6.1.1 MaxStream Client Application Parameters

Input rate and batch size will be determined in this section.
6.1.1.1 Input Rate

In this experiment, we wanted to show the change in end-to-end latency as the input rate increase. Input rate is the number of tuples that are sent to the input adapter per second. As one can observe from figure 6.1, in MaxStream architecture latency decreases as the rate increases. In order to understand this behavior we should first discuss the "time progress" concept in stream processing engines.

We will describe Coral8’s time concept since MaxStream is using Coral8 underneath for stream processing. There are two different time advancement in Coral8. First is the system time -as we can understand from its name- which is the actual time that all the external systems are using. Second one is the application time which is used internally by Coral8. The main difference between them is how the time progress. Contrary to system time, application time of Coral8 advances only when a tuple is received or sent by an adapter. In other words, application time stays the same between two consecutive tuples. This means that a tuple received by an input adapter is kept inside the adapter until another tuple is received by the system.

For an input rate of 1 tuple/sec, MaxStream’s End-to-End Latency is measured to be slightly higher than 1 second (1.0294 sec). When we increase the input rate to 10 tuples per second latency decreases to 0.10678 sec. As we can understand from the trend, tuples are waiting to be processed inside the adapters, which constitutes the major part of the latency. To decrease the latency, we increased the arrival rate of the tuples to the system. We can see from the figure 6.1 for 500 tuples/sec the latency is 0.0069 sec.

6.1.1.2 Batch Size

The client application which sends the tuples to the servers has a rate limitation because after sending each tuple it performs a database commit which is costly. In order to overcome the rate limitation (500 tuples/sec), we have implemented a batching option. Instead of committing after every tuple, we wait until certain number of tuples are sent.

Via batching higher arrival rates are achieved which in return result in higher throughput. As one can observe from figure 6.2, although the throughput for batch size 1 (means commit after every tuple sent) is around 500, with larger batches we reach 2750 tuples/sec.

However, batching has some drawbacks as well. End-to-End latency increases as the batch size increase. Since the client application will prepare a batch of tuples before performing a commit operation, the first tuples that are ready to be processed will wait for the whole batch to be ready in the system. This trend can be seen from figure 6.3.
To express the results better, we can check the throughput vs latency graph 6.4. MaxStream supports the queries that we mentioned in chapter 5 with a low end-to-end latency until a certain level. When we want to reach higher throughput rates, latency increases drastically around 2750 tuples/sec.
Figure 6.3: MaxStream - Batch Size vs Latency - (Input Rate: varies with batch size, MaxDelay: 1 second)

Figure 6.4: MaxStream - Throughput vs Latency - (MaxDelay: 1 second)

6.1.2 Coral8 Studio Parameters

Input rate, batch size, poll interval and MaxDelay parameters will be determined in this section. We have also tested alternative databases for Coral8 server.
6.1.2.1 Input Rate

Inside Coral8 studio, there is a parameter called ”Maximum Delay”. It means, even if there isn’t a tuple arriving to the system, process the tuples inside the adapters after a certain time which is set by maximum delay parameter. Thanks to this property, the rate-latency trend in Coral8 only scenario behaves contrarily to MaxStream (figure 6.5). Minimum latency is achieved with 1 tuple/sec rate which is 0.0268 sec. End-to-End latency stays the same till 200 tuples/sec, then increases drastically. This is due to the bottleneck of the connection between Coral8 Server and database used underneath it. This issue is discussed further in sections 6.1.2.5 and 6.2.7.

![Coral8 Input Rate vs Latency](image)

**Figure 6.5:** Coral8 - Input Rate vs Latency - (MaxDelay: 1 msec, Poll interval: 1 msec, database: DB2)

6.1.2.2 Batch Size

The batching-throughput graph 6.6 for Coral8 shows a similar behavior with MaxStream. We achieve higher throughput with larger batch sizes.

For Coral8, the trend for latency is same with MaxStream, average latency for tuples sent increase as the batch size increase (figure 6.7).

Similarly, in Coral8, higher throughput rates come with a cost of increase in latency (figure 6.8).
6.1.2.3 Poll Interval

We described the architecture of Coral8 experiment setup in section 5.2, the poll interval parameter is used for determining the frequency of checking the database table for newly arrived tuples. As expected checking the database too frequently (every microsecond) doesn’t make a difference in end-to-end latency. In figure 6.9 we can observe an increase
in latency with intervals longer than 10 milliseconds because the tuples which are sent to database are waiting to be processed by the input adapter. Therefore we have selected 1 millisecond as the optimal value for poll interval parameter.

\[\text{Figure 6.8: Coral8 - Throughput vs Latency - (MaxDelay: 1 msec, Poll interval: 1 msec, database: DB2)}\]

\[\text{Figure 6.9: Coral8 - Poll Interval vs Latency - (MaxDelay: 1 msec, Input Rate: 1 tuple/sec, database: DB2)}\]
6.1.2.4 Maximum Delay

Another parameter for Coral8 Experiment that we mentioned in section 5.2 is maximum delay. Coral8 server advances the time when it receives a tuple. If the arrival rate is low, tuples will wait for the next one in order to be processed and flow through the system. We have used the 1 millisecond value for maximum delay parameter in Coral8 since it has the lowest end-to-end latency as seen in figure 6.10.

![Coral8 - Maximum Delay vs Latency](image)

**Figure 6.10:** Coral8 - Max Delay vs Latency - (Input Rate: 1 tuple/sec, Poll interval: 1 msec, database: DB2)

6.1.2.5 Alternative Databases

As we discussed in Chapter 2 Stream Processing Engines work on top of a database and persist the streamed data in the database. Therefore the database used has a high influence on the performance of stream processing engine. We have benchmarked Coral8 with different databases in order to find the highest performance setup.

- **MaxDB:** MaxDB is owned by SAP AG. MaxStream is built on top of MaxDB with streaming functionality. We have tested Coral8 with MaxDB v7.8.01.18. For ODBC connection between Coral8 and MaxDB, we used the available UNIX/Linux drivers: libsqlod.so and libsqlodw.so.

- **Oracle:** Benchmarks are done using Oracle XE 10g. Oracle does not provide an ODBC driver for Linux/UNIX platforms. A third party firm called “easysoft”
provides the drivers commercially. We have used "Easysoft ODBC-Oracle Driver v3.3" for 64 bit linux platform for our experiments with Oracle Database.

- **MySQL**: MySQL Red Hat & Oracle Enterprise Linux 5 (x86, 64-bit) v5.5.12. We have used the ODBC driver "mysql-connector-odbc-5.1.8-1.rhel5.x86_64.rpm".

- **PostgreSQL**: Stable release of PostgreSQL 9.0.4 is installed with the official PostgreSQL ODBC driver "psqlODBC" for 64 bit systems (version 09-00-02 which is released on 2010-10-30).

- **MonetDB**: We have used the ODBC driver provided with column-oriented database management system MonetDB-11.3.1.

- **DB2**: DB2 v9.7 with ODBC driver v9.7 for linux 64 bits systems is used.

Among all these database systems, DB2 performed the best in terms of end-to-end latency. The performance difference comes from the ODBC driver since query execution in all databases approximately takes same amount of time. Coral8 needs to check the database for new tuples very frequently\(^1\) and DB2’s ODBC driver handles the requests very efficiently compared to other databases tested. Measurement details can be checked from Table 6.1.

<table>
<thead>
<tr>
<th>Database</th>
<th>End-to-End Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxDB</td>
<td>above 0.4 sec</td>
</tr>
<tr>
<td>Oracle XE</td>
<td>above 0.7 sec</td>
</tr>
<tr>
<td>MySQL</td>
<td>0.3 sec</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0.7 sec</td>
</tr>
<tr>
<td>MonetDB</td>
<td>1.3 sec</td>
</tr>
<tr>
<td>DB2</td>
<td>0.025 sec</td>
</tr>
</tbody>
</table>

### 6.2 Comparison of the Two Systems

In this section we will compare MaxStream and Coral8, also describe the limitations faced when we benchmarked the two systems. From Section 6.1 we have concluded that the best configuration for minimum latency:

**Coral8**: Input Rate (throughput): 1 tuple/sec, MaxDelay: 1 msec, Poll Interval: 1 msec, Database: DB2, Batch Size: 1 tuple/batch

**MaxStream**: Input Rate (throughput): 500 tuples/sec, MaxDelay: 1 sec, Batch Size: 1 tuple/batch

\(^1\)every millisecond
However, if we want to get the highest throughput while keeping the end-to-end latency at an acceptable level, here are the configuration details:

**Coral8**: Input Rate (throughput): 200 tuple/sec, MaxDelay: 1 msec, Poll Interval: 1 msec, Database: DB2, Batch Size: 200 tuple/batch

**MaxStream**: Input Rate (throughput): 2665 tuples/sec, MaxDelay: 1 second, Batch Size: 50 tuples/batch

### 6.2.1 Selectivity

Selectivity experiments are done in order to test the systems how they will perform when certain percentage of the queries does not fetch any data from database. We managed this by adding another constraint on the query. For instance when we want 10% of the queries return back result, we adjust the value in the where clause of the query respectively. The threshold used in the query is calculated by counting the number of tuples which are above the desired percentage.

**MaxStream - Enrich&Persist - MONITORING SELECT Selectivity Query**

```sql
SELECT /*+ MONITOR(TOTALSALES)*/ event_time, OrderID, ProductName, Sum_Quantity, TV_SEC, TV_USEC, TS FROM TOTALSALES, Product P WHERE ProductID=P.ProductPID AND Sum_Quantity > ? AND TS > ? ORDER BY TS
```

The selectivity experiments showed that the number of tuples fetched from database per query doesn’t change end-to-end latency except for 1% MaxStream case (figure 6.11). We have observed the same result in input rate experiments in subsection 6.1.1.1 due to maximum delay parameter\(^2\). With 1 percent selectivity the number of tuples fetched from database are really less in number. This low arrival rate result in tuples’ waiting in input adapters which increases end-to-end latency.

Experiments on Coral8 (see figure 6.12) showed that end-to-end latency is not affected by selectivity.

**Coral8 - Enrich&Persist - Selectivity Query**

```sql
select count(*) from totalorderstable where totalquantity > ?
```

Threshold values for each selectivity ratio can be seen from Table 6.2.

\(^2\)The solution for this issue is decreasing the maximum delay parameter’s value in MaxStream and recompiling the source code. We have tried this approach but after the changes we couldn’t overcome the connection problems between Coral8 server and MaxStream prototype. Therefore we rolled back to the previous version and mentioned this in future work, Section 7.2
6.2.2 Latency of Subintervals

In order to analyze the end-to-end latency of the system in depth, we have divided the whole process into 4 subintervals:

- Interval 1 - Input Persistence & Query Feed From Persisted Input: This
interval starts with reading the input from file. The tuples are persisted into database. Interval ends when the stream is created from persisted data inside the stream processing engine.

- **Interval 2 - Query Processing:** Covers the time passed between creation of the stream and execution of the query over the stream.

- **Interval 3 - Output Persistence:** The processed stream is persisted into database.

- **Interval 4 - Client Feed From Persisted Output:** Starts when the output is read from the database, ends with writing the result in output file for client application.

Figure 6.13 shows the intervals explained above for persist and enrich&persist cases of Coral8. Persisted Coral8 (C8-P) and enrich&persist Coral8 (C8-EP) have the same intervals except for I1. It takes slightly more time for C8-EP to perform join operation which causes the difference in first interval.

As you can see from Figure 6.14 persisted MaxStream (MXSTRM-P) and enrich&persist MaxStream (MXSTRM-EP) have the same trend with Coral8. First interval takes a little bit longer due to join operation performed with products table.

Moreover, output persistence interval (I3) is not seen in all experiments. It shows that writing the tuples to database takes a negligible amount of time compared to reading them. Although y-axis of the Figures 6.13 and 6.14 show the average latency, the error-bars on the figures are not seen. That is because of the sample size of the experiments. Variance of the mean is calculated as following:

\[
\text{var}(\bar{x}) = \frac{s^2}{n}
\]

where \( n \) is the sample size. Even with the smallest scale factor (0.1), our sample set constitutes of 150,000 tuples. Therefore the variance and consequently the standard error is negligible when we divide dividend with such a high number.

---

Table 6.2: Threshold Values for Selectivity Ratios

<table>
<thead>
<tr>
<th>Selectivity Ratio</th>
<th>Threshold Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>61</td>
</tr>
<tr>
<td>10%</td>
<td>46</td>
</tr>
<tr>
<td>25%</td>
<td>39</td>
</tr>
<tr>
<td>50%</td>
<td>26</td>
</tr>
<tr>
<td>100%</td>
<td>0</td>
</tr>
</tbody>
</table>

---
When we compare the MaxStream and Coral8, the difference between them comes from the time spent on reading data from database (I1 and I4). MaxStream performs these 4 times better in persist scenario and 5 times better in enrich\&persist scenario. These results clearly show that “Monitoring Select” and “ISTREAM” which are used by MaxStream implementation are superior to “Polling Database” approach of Coral8. As we showed in subsection 6.1.2.3, even using the shortest poll interval possible, Coral cannot retrieve new tuples from the database as fast as MaxStream.
Figure 6.13’s graph dataset can be examined from Table 6.3. Subinterval latency unit is second.

<table>
<thead>
<tr>
<th>Experiment Type</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coral8-Persist</td>
<td>0.01546</td>
<td>0.00235</td>
<td>0.0</td>
<td>0.00614</td>
</tr>
<tr>
<td>Coral8-Enrich&amp;Persist</td>
<td>0.02064</td>
<td>0.00240</td>
<td>0.0</td>
<td>0.00806</td>
</tr>
</tbody>
</table>

Figure 6.14’s graph dataset can be examined from Table 6.4. Subinterval latency unit is second.

<table>
<thead>
<tr>
<th>Experiment Type</th>
<th>(I1+I2)</th>
<th>(I3+I4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxStream-Persist</td>
<td>0.00420</td>
<td>0.00322</td>
</tr>
<tr>
<td>MaxStream-Enrich&amp;Persist</td>
<td>0.00544</td>
<td>0.00179</td>
</tr>
</tbody>
</table>

### 6.2.3 Scale Factor vs Latency

We have measured the end-to-end latency of Coral8 and MaxStream with respect to the scale factor. Scale factor determines the size of the database table (productstable) and the number of order tuples which are populated according to modified version of TPC-H (see Chapter 4). Scale factor value and table size are directly proportional. We performed the tests once with and once without index on productstable.

#### 6.2.3.1 Product Table with Index on ProductId Column

As we can see from Figures 6.15 and 6.16, end-to-end latency does not change when we increase the scale factor. Even with a scale factor of 10 which has ten times larger database tables, the latency stays the same. This is thanks to the index on ProductId Column in ProductsTable. The join operation that we perform on ProductsTable fetches the data without scanning the whole table so no additional delay occurs due to data size.

#### 6.2.3.2 Product Table without Index on ProductId Column

When we remove the index on the ProductsTable, end-to-end latency increases with the scale factor as expected (see Figure 6.17). This experiment is done with input rate of 1 tuple per second using Coral8 setup. When we want to increase the rate, system fails because the database cannot answer too many requests.
Similarly in MaxStream, the optimum input rate we found was 500 tuples/sec (see Figure 6.1). Database couldn’t respond that many requests and stopped responding. Hence, we don’t have a graph that shows the end-to-end latency versus scale factor for MaxStream.
6.2.4 Input Rate

Input rate was the first limitation that we encountered as we discussed in section 6.1.1.1. For Coral8, we cannot increase the rate because end-to-end latency increases as well. In order to run the tests with the lowest latency possible, we should keep the rate at 1 tuple per second.

On the other hand, MaxStream achieves lower latency with higher input rates. This was shown in Figure 6.1. The tuples that are sent to Coral8 from MaxStream are waiting to be processed till another tuple enters to the system. When we send the tuples more frequently (with a higher input rate), the pause between two tuples decreases. Hence we have lower end-to-end latency. The optimum input rate we measured was 500 tuples per second with a batch size of 1. We can increase the input rate up to 2665 with a batch size of 50.

6.2.5 MaxDelay

From Coral8 Studio, we can set the maximum waiting time before processing the tuples inside the input adapters. Via MaxDelay parameter, we can reduce the delay to its optimum value. We chose 1 millisecond from Figure 6.10.
MaxStream is implemented on top of MaxDB. Coral8 communicates with MaxStream using ODBC driver of MAXDB. When we decrease the maximum latency by recompiling the source code of MaxStream prototype, the communication between Coral8 and MaxStream stalls. Therefore we left the MaxDelay parameter as 1 second for MaxStream. The workaround for this problem is to increase the input rate so that maximum delay parameter will not be used by the system at all. For instance, the input rate we used for MaxStream was 500 tuples/sec which leads to 1/500 seconds (2 milliseconds) of negligible delay.

### 6.2.6 Poll Interval

MaxStream uses “Monitoring Select” and “ISTREAM” instead of polling the database. This is the key advantage of MaxStream and does not require a parameter tuning.

For Coral8, we need to determine a frequency to check the database. As we discussed in section 6.1.2.3, the optimal latency is 1 millisecond to achieve the minimum end-to-end latency (See figure 6.9).

### 6.2.7 ODBC Connection

Short for Open DataBase Connectivity, a standard database access method developed by the SQL Access group in 1992. All the inter-system connections (Client-MaxDB, MaxStream-Coral8, Coral8-DB2) are achieved with ODBC. The dominant factor in latency is the communication between different layers. So the odbc drivers that we use have a high influence on the measurements. MaxDB’s odbc driver had limitations on the number of frequent requests using the same connection. The connection was lost when Coral8 queries the database too frequently. Therefore we have tested many other databases (section 6.1.2.5) with different odbc drivers. DB2’s delay in inter-system connections was the shortest compared to other databases. Therefore, we preferred to use DB2 underneath Coral8.

### 6.3 Coral8 - Parallel Setup Results

We described Coral8 parallel measurement setup in Section 5.2.2. The results of the experiments can be seen from Figure 6.18. Parallel persist scenario’s end-to-end latency is 0.0038 sec. This is shorter than MaxStream’s persist case because MaxStream is not sending the tuples to the stream processor before ensuring the persistence of the input. This is causing a delay compared to parallel scenario of Coral8.
Parallel enrich\&persist case of Coral8 has an end-to-end latency of 0.011 sec. This interval is longer compared to persist only scenario because streams are enriched using the static database table. Coral8 server needs to retrieve data from database for processing each tuple.

We cannot compare the results of parallel scenario with MaxStream because MaxStream has only sequential setups. Although it is not possible to create a parallel scenario for MaxStream, we included the results of Coral8 parallel setup to show its capabilities with different constraints.
Chapter 7

Conclusion and Future Work

The conclusion of this thesis is presented in Section 7.1 while Section 7.2 provides a summary of the possible avenues for future work.

7.1 Conclusion

There are two different approaches in Stream Processing as we discussed in Chapter 2 and Chapter 3. First is the SPE based approach where Stream Processing Engines are the main interface to the client. This approach aims SPE-DBMS integration. On the other hand, MaxStream is a stream federator, not a full-fledged SPE. It aims mainly integration. Different SPEs and traditional databases are coordinated underneath the federation layer of MaxStream. SAP’s MaxDB has been modified to cover the concept of stream. It has new data definition statements and query compilation logic for stream creation. Continuous queries can be directed to SPEs and the result of them returns back to MaxStream.

In this thesis, we have mentioned the limitations of available integrated stream processing systems and described the architecture of federated stream processing prototype developed by ETH Zurich. After tailoring the TPC-H down to a streaming benchmark scenario, we have tested the new promising direction of stream engine federation and compared it with its alternatives. Our thorough experiments with this modified version of TPC-H showed the feasibility of our approach from the performance perspective. Our results showed that MaxStream outperforms Coral8 by at least a factor of 4 in terms of different performance metrics, such as end-to-end latency and throughput.

Main advantage of MaxStream is “ISTREAM” and “MONITORING SELECT” functionalities. ISTREAM operator is used for creating streams from static database tables.
Only the newly inserted tuples are retrieved from the table. To continue with the MONITORING SELECT, it utilizes the blocking select mechanism. Output table is monitored and returns the results when they are available. Alternative stream processing systems use “POLL DB” implementation. Detailed comparison results of the two approaches can be checked from Chapter 6.

7.2 Future Work

While MaxStream in its current state outperforms other integrated stream processing systems, several avenues for future work exist. One such avenue is to provide integration with different SPEs underneath. Future work in this area includes implementing a Data Agent for control messages and input streams forwarded to SPE.

Another aspect of future work is to repeat the experiments using a different benchmark. We have modified TPC-H benchmark and tailored it down to our streaming scenario. Same method can be applied to TPC-C or TPC-E.

As we discussed in Section 6.2.1 MaxDelay parameter issue of MaxStream should be solved in order to benchmark the system with low input arrival rates.

Lastly, MaxStream prototype should benefit from its federated architecture more such as allowing richer applications over a particular SPE and load balancing.
Appendix A

Database Schemas, Streams and Continuous Queries

A.1 Coral8

A.1.1 Coral8 - Persist Scenario - Database Schemas

CREATE TABLE OrdersTable (eventTime INTEGER, OrderId INTEGER, ClientId INTEGER, ProductId INTEGER, Quantity INTEGER, OrderDate INTEGER, Priority INTEGER, Tax INTEGER, Discount INTEGER, Price INTEGER, Comments INTEGER, TS_1 TIMESTAMP, TS_2 TIMESTAMP)

CREATE TABLE TotalOrdersTable (eventTime INTEGER, ProductId INTEGER, TotalQuantity INTEGER, TS_1 TIMESTAMP, TS_2 TIMESTAMP, TS_3 TIMESTAMP, TS_4 TIMESTAMP, TS_5 TIMESTAMP)

A.1.2 Coral8 - Persist - Project 1

- Input Persistence: Project 1 starts with reading the input from file. Tuples are persisted into database (OrdersTable).

```sql
-- INPUT starts
EXECUTE STATEMENT DATABASE "DB2"
[[
    Insert into OrdersTable values(?eventTime,?OrderId,?ClientId,?ProductId, ?Quantity,?OrderDate,?Priority,?Tax,?Discount,?Price,?Comments,?TS_1, ?TS_2)
]]
SELECT eventTime as eventTime,
    OrderId as OrderId,
    ClientId as ClientId,
```
(productId, quantity, orderdate, priority, tax, discount, price, comments, gettimestamp(ordersstream) as ts_1, now() as ts_2) 
FROM ordersstream;

-- Create a schema for input and output streams
--
CREATE SCHEMA ordersstreamschema ( 
  eventtime INTEGER, 
  orderid INTEGER, 
  clientid INTEGER, 
  productid INTEGER, 
  quantity INTEGER, 
  orderdate INTEGER, 
  priority INTEGER, 
  tax INTEGER, 
  discount INTEGER, 
  price INTEGER, 
  comments INTEGER 
); 

-- Create an input stream
-- c8_input_orders_f01.csv
CREATE INPUT STREAM ordersstream SCHEMA ordersstreamschema;
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType 
TO STREAM ordersstream
PROPERTIES 
  FILENAME = "/local/Coral8/coral8/server/Coral8Repository/INPUT/c8_input_orders_f01.csv", 
  USECURRENTTIMESTAMP = "true", 
  RATE = "1", 
  TIMESTAMPCOLUMN = "false" 
;

A.1.3 Coral8 - Persist - Project 2

- Query Feed From Persisted Input: Stream is created using tuples from OrdersTable.
- Query Processing: Execution of the summation query over the stream.
• **Output Persistence:** The processed stream is persisted into database (TotalOrdersTable).

```sql
CREATE SCHEMA PollOrdersStreamSchema (
    eventTime INTEGER,
    ProductId INTEGER,
    Quantity INTEGER,
    TS_1 TIMESTAMP,
    TS_2 TIMESTAMP
);

CREATE INPUT STREAM PollOrdersStream SCHEMA PollOrdersStreamSchema;
ATTACH INPUT ADAPTER PollDBAdapter TYPE ReadFromDBAdapterType TO STREAM PollOrdersStream
PROPERTIES
    QUERY = [[[ SELECT eventTime, ProductId, Quantity, TS_1, TS_2
                   FROM OrdersTable WHERE eventTime > ?C8_COUNTER_FIELD
                   ORDER BY eventTime ASC ]],
    DBNAME = "DB2",
    COUNTERFIELD = "eventTime",
    COUNTERFIELDINITVALUE = "0",
    POLLINTERVAL = "1000"
);

INSERT INTO TotalOrdersStream
SELECT
    eventTime, ProductId, SUM(Quantity), TS_1, TS_2, GETTIMESTAMP(PollOrdersStream)
FROM
    PollOrdersStream
KEEP 1 MINUTE
GROUP BY ProductId;

EXECUTE STATEMENT DATABASE "DB2"
[[
    Insert into TotalOrdersTable values(?eventTime,?ProductId,?TotalQuantity
    ,?TS_1,?TS_2,?TS_3,?TS_4)
]
SELECT
    eventTime as eventTime,
    ProductId as ProductId,
    TotalQuantity as TotalQuantity,
    TS_1 as TS_1,
    TS_2 as TS_2,
    TS_3 as TS_3,
    NOW() as TS_4
FROM
    TotalOrdersStream;

CREATE SCHEMA TotalOrdersStreamSchema (
    eventTime INTEGER,
    ProductId INTEGER,
    TotalQuantity INTEGER,
    TS_1 TIMESTAMP,
    TS_2 TIMESTAMP,
    TS_3 TIMESTAMP,
    TS_4 TIMESTAMP
);
```
Appendix A. Database Schemas, Streams and Continuous Queries

CREATE OUTPUT STREAM TotalOrdersStream SCHEMA TotalOrdersStreamSchema;

A.1.4 Coral8 - Persist - Project 3

- Client Feed From Persisted Output: Output is read from the database and sent to the client application.

CREATE SCHEMA ResultStreamSchema (  
eventTime INTEGER,  
ProductId INTEGER,  
TotalQuantity INTEGER,  
TS_1 TIMESTAMP,  
TS_2 TIMESTAMP,  
TS_3 TIMESTAMP,  
TS_4 TIMESTAMP  
);

CREATE INPUT STREAM ResultStream SCHEMA ResultStreamSchema;
ATTACH INPUT ADAPTER PollDBAdapter TYPE ReadFromDBAdapterType TO STREAM ResultStream
PROPERTIES
QUERY = [[ SELECT eventTime, ProductId, TotalQuantity, TS_1,  
    TS_2, TS_3, TS_4  
    FROM TotalOrdersTable WHERE eventTime >  
    ?C8_COUNTER_FIELD  
    ORDER By eventTime ASC]],
DBNAME = "DB2",
COUNTERFIELD = "eventTime",
COUNTERFIELDINITVALUE = "0",
POLLINTERVAL = "1000";

INSERT INTO
FileStream
SELECT  
eventTime, ProductId, TotalQuantity, TS_1, TS_2, TS_3, TS_4,  
GETTIMESTAMP(ResultStream)  
FROM
ResultStream;

CREATE SCHEMA FileStreamSchema (  
eventTime INTEGER,  
ProductId INTEGER,  
TotalQuantity INTEGER,  
TS_1 TIMESTAMP,  
TS_2 TIMESTAMP,  
TS_3 TIMESTAMP,  
TS_4 TIMESTAMP  
);
A.1.5 Coral8 - Enrich&Persist Scenario - Database Schemas

CREATE TABLE ProductsTable (ProductPID INTEGER, ProductName INTEGER, myType INTEGER, Brand INTEGER, mySize INTEGER, Container INTEGER, RetailPrice INTEGER, SupplyCost INTEGER, Available_QTY INTEGER, Comments INTEGER)

CREATE INDEX IDX_PROD_OZ ON ProductsTable (ProductPID)

CREATE TABLE OrdersTable (eventTime INTEGER, OrderId INTEGER, ClientId INTEGER, ProductId INTEGER, Quantity INTEGER, OrderDate INTEGER, Priority INTEGER, Tax INTEGER, Discount INTEGER, Price INTEGER, Comments INTEGER, TS_1 TIMESTAMP, TS_2 TIMESTAMP)

CREATE TABLE TotalOrdersTable (eventTime INTEGER, ProductName INTEGER, TotalQuantity INTEGER, TS_1 TIMESTAMP, TS_2 TIMESTAMP, TS_3 TIMESTAMP, TS_4 TIMESTAMP, TS_5 TIMESTAMP)

A.1.6 Coral8 - Enrich&Persist - Project 1

- **Input Persistence:** Project 1 starts with reading the input from file. Tuples are persisted into database (OrdersTable).

```sql
-- INPUT starts
EXECUTE STATEMENT DATABASE "DB2"
[[
]]
SELECT eventTime as eventTime, OrderId as OrderId,
```
ClientId as ClientId,
ProductId as ProductId,
Quantity as Quantity,
OrderDate as OrderDate,
Priority as Priority,
Tax as Tax,
Discount as Discount,
Price as Price,
Comments as Comments,
GETTIMESTAMP(OrdersStream) as TS_1,
NOW() as TS_2
FROM
OrdersStream;

-- schema for input and output streams
CREATE SCHEMA OrdersStreamSchema (  
eventTime INTEGER,
OrderId INTEGER,
ClientId INTEGER,
ProductId INTEGER,
Quantity INTEGER,
OrderDate INTEGER,
Priority INTEGER,
Tax INTEGER,
Discount INTEGER,
Price INTEGER,
Comments INTEGER);

-- Create an input stream
-- c8_input_orders_f01 . csv
CREATE INPUT STREAM OrdersStream SCHEMA OrdersStreamSchema;
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM OrdersStream
PROPERTIES
FILENAME = " /local/Coral8/coral8/server/Coral8Repository/INPUT/c8_input_orders_f01.csv",
USECURRENTTIMESTAMP = "true",
RATE = "1",
TIMESTAMPCOLUMN = "false"
;

A.1.7 Coral8 - Enrich&Persist - Project 2

• Query Feed From Persisted Input & Enrichment of Data: Stream is created using tuples from OrdersTable. At the same time, stream is enriched using the static ProductsTable.

• Query Processing: Execution of the summation query over the stream.
Output Persistence: The processed stream is persisted into database (TotalOrdersTable).

CREATE SCHEMA PollOrdersStreamSchema (  
  eventTime INTEGER,  
  ProductId INTEGER,  
  Quantity INTEGER,  
  TS_1 TIMESTAMP,  
  TS_2 TIMESTAMP  
);

CREATE INPUT STREAM PollOrdersStream SCHEMA PollOrdersStreamSchema ;
ATTACH INPUT ADAPTER PollDBAdapter TYPE ReadFromDBAdapterType TO STREAM PollOrdersStream
PROPERTIES
  QUERY = [[ SELECT eventTime, ProductId, Quantity, TS_1, TS_2  
          FROM OrdersTable WHERE eventTime > ?C8_COUNTER_FIELD  
          ORDER By eventTime ASC ]],
  DBNAME = "DB2",
  COUNTERFIELD = "eventTime",
  COUNTERFIELDINITVALUE = "0",
  POLL_INTERVAL = "1000"
;

INSERT INTO TotalOrdersStream
SELECT P.eventTime, DBResult.ProductName, SUM(P.Quantity), TS_1, TS_2,  
GETTIMESTAMP(P)  
FROM PollOrdersStream P,  
(Database "DB2"  
SCHEMA (ProductPID INTEGER, ProductName INTEGER)  
[[  
  SELECT ProductPID, ProductName  
  FROM ProductsTable  
  where ProductPID=?)P.ProductId  
  ]])  
) as DBResult
KEEP 1 MINUTE
GROUP BY DBResult.ProductName;

EXECUTE STATEMENT DATABASE "DB2"
[[  
  Insert into TotalOrdersTable values (?)eventTime,?)ProductName,  
  ?TotalQuantity,?TS_1,?TS_2,?TS_3,?TS_4  
  ]]
SELECT  
  eventTime as eventTime,  
  ProductName as ProductName,  
  TotalQuantity as TotalQuantity,  
  TS_1 as TS_1,  
  TS_2 as TS_2,  
  TS_3 as TS_3,  
  NOW() as TS_4
FROM
TotalOrdersStream;

CREATE SCHEMA TotalOrdersStreamSchema ( 
  eventTime INTEGER,
  ProductName INTEGER,
  TotalQuantity INTEGER,
  TS_1 TIMESTAMP,
  TS_2 TIMESTAMP,
  TS_3 TIMESTAMP
);

CREATE OUTPUT STREAM TotalOrdersStream SCHEMA TotalOrdersStreamSchema;

A.1.8 Coral8 - Enrich&Persist - Project 3

• Client Feed From Persisted Output: Output is read from the database and sent to the client application.

CREATE SCHEMA ResultStreamSchema ( 
  eventTime INTEGER,
  ProductName INTEGER,
  TotalQuantity INTEGER,
  TS_1 TIMESTAMP,
  TS_2 TIMESTAMP,
  TS_3 TIMESTAMP,
  TS_4 TIMESTAMP
);

CREATE INPUT STREAM ResultStream SCHEMA ResultStreamSchema;
ATTACH INPUT ADAPTER PollDBAdapter TYPE ReadFromDBAdapterType
TO STREAM ResultStream
PROPERTIES
  QUERY = [[ SELECT eventTime, ProductName, TotalQuantity,
              TS_1, TS_2, TS_3, TS_4
           FROM TotalOrdersTable WHERE eventTime >
              ?C8_COUNTER_FIELD
           ORDER BY eventTime ASC]],
  DBNAME = "DB2",
  COUNTERFIELD = "eventTime",
  COUNTERFIELDINITVALUE = "0",
  POLLINTERVAL = "1000"
;

INSERT INTO
FileStream
SELECT
  eventTime, ProductName, TotalQuantity, TS_1, TS_2, TS_3, TS_4,
  GETTIMESTAMP(ResultStream)
FROM
  ResultStream;
CREATE SCHEMA FileStreamSchema (
    eventTime INTEGER,
    ProductName INTEGER,
    TotalQuantity INTEGER,
    TS_1 TIMESTAMP,
    TS_2 TIMESTAMP,
    TS_3 TIMESTAMP,
    TS_4 TIMESTAMP,
    TS_5 TIMESTAMP
);

-- Create an output stream
-- c8_output_f01.csv
CREATE OUTPUT STREAM FileStream SCHEMA FileStreamSchema;
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM FileStream
PROPERTIES
    FILENAME = "/local/Coral8/coral8/server/Coral8Repository/OUTPUT/c8_output_poll_enrichpersist_db2_detailed.csv",
    USECURRENTTIMESTAMP = "true";

A.2 MaxStream

A.2.1 Persist Scenario - Database Schemas

CREATE TABLE OrdersTable (eventTime INTEGER, OrderId INTEGER, ClientId INTEGER,
    ProductId INTEGER, Quantity INTEGER, OrderDate INTEGER, Priority INTEGER,
    Tax INTEGER, Discount INTEGER, Price INTEGER, Comments INTEGER, TV_SEC INTEGER,
    TV_USEC INTEGER, TS INTEGER)

CREATE TABLE TotalOrdersTable (TS INTEGER, OrderId INTEGER, ProductId INTEGER,
    Sum_Quantity INTEGER, TV_SEC INTEGER, TV_USEC INTEGER, TS_2 TIMESTAMP,
    TS_3 TIMESTAMP, TS_4 TIMESTAMP, PRIMARY KEY (TS))

A.2.2 MaxStream - Persist

CREATE SEQUENCE TotalOrdersSeq
CREATE DATASOURCE CORAL8DS1 TYPE CORAL8 EXECUTE INPROC OPTIONS
    (URL 'http://localhost:6789', WORKSPACE 'Default', ISSTREAMING 'YES')
CREATE MODULE OrdersModule IN DATASOURCE Coral8DS1 \
BEGIN \
CREATE INPUT STREAM OrdersStream (eventTime INTEGER, OrderId INTEGER, ClientId
    INTEGER, ProductId INTEGER, Quantity INTEGER, OrderDate INTEGER, Priority INTEGER,
    Tax INTEGER, Discount INTEGER, Price INTEGER, Comments
    INTEGER, TV_SEC INTEGER, TV_USEC INTEGER,
A.2.3 MaxStream - Persist - MONITORING SELECT

SELECT /*+MONITOR(TOTALSALES)*/ event_time, OrderID, ProductID, Sum_Quantity, TV_SEC, TV_USEC, TS FROM TOTALSALES, Product P WHERE ProductID = P.ProductID AND TS > ? ORDER BY TS

A.2.4 Enrich&Persist Scenario - Database Schemas

CREATE TABLE ProductsTable (ProductPID INTEGER, ProductName INTEGER, myType INTEGER, Brand INTEGER, mySize INTEGER, Container INTEGER, RetailPrice INTEGER, SupplyCost INTEGER, Available_QTY INTEGER, myComments INTEGER)

CREATE INDEX IDX_PROD_OZ_f01 ON ProductsTable (ProductPID)

CREATE TABLE OrdersTable (eventTime INTEGER, OrderId INTEGER, ClientId INTEGER, ProductId INTEGER, Quantity INTEGER, OrderDate INTEGER, Priority INTEGER, Tax INTEGER, Discount INTEGER, Price INTEGER, Comments INTEGER, TV_SEC INTEGER, TV_USEC INTEGER, TS INTEGER)

CREATE TABLE TotalOrdersTable (TS INTEGER, OrderId INTEGER, ProductID INTEGER, Sum_Quantity INTEGER, TV_SEC INTEGER, TV_USEC INTEGER, TS_2 TIMESTAMP, TS_3 TIMESTAMP, TS_4 TIMESTAMP, PRIMARY KEY (TS))
A.2.5 MaxStream - Enrich&Persist

CREATE SEQUENCE TotalOrdersSeq
CREATE DATASOURCE CORAL8DS1 TYPE CORAL8 EXECUTE INPROC OPTIONS
(URL 'http://localhost:6789', WORKSPACE 'Default', ISSTREAMING 'YES')
CREATE MODULE OrdersModule IN DATASOURCE Coral8DS1
BEGIN 
CREATE INPUT STREAM OrdersStream (eventTime INTEGER, OrderId INTEGER, ClientId
INTEGER, ProductId INTEGER, Quantity INTEGER, OrderDate INTEGER, Priority INTEGER, Tax INTEGER,
Discount INTEGER, Price INTEGER, ProductName INTEGER, TV_SEC INTEGER, TV_USEC INTEGER,
Min INTEGER); 
INSERT INTO OrdersStream (eventTime, OrderId, ClientId, ProductId, Quantity,
OrderDate, Priority, Tax, Discount, Price, ProductName, TV_SEC, TV_USEC, Min_)
SELECT /*+ STREAM_TIMESTAMP (TS )*/ eventTime, OrderId, ClientId, ProductId,
Quantity, OrderDate, Priority, Tax, Discount, Price, ProductName, TV_SEC, TV_USEC, TS
FROM ISTREAM (OrdersTable) O, DBA.ProductsTable P WHERE
O.ProductID = P.ProductPID;
CREATE LOCAL STREAM OrdersLocalStream (eventTime INTEGER, OrderId INTEGER, ClientId
INTEGER, ProductId INTEGER, Quantity, OrderDate INTEGER, Priority INTEGER, Tax,
Discount INTEGER, Price INTEGER, ProductName INTEGER, TV_SEC INTEGER, TV_USEC INTEGER,
Min INTEGER, TS_2 TIMESTAMP);
INSERT INTO OrdersLocalStream (eventTime, OrderId, ClientId, ProductId, Quantity,
OrderDate, Priority, Tax, Discount, Price, ProductName, TV_SEC, TV_USEC, Min_, TS_2)
SELECT eventTime, OrderId, ClientId, ProductId, Quantity, OrderDate, Priority, Tax,
Discount, Price, ProductName, TV_SEC, TV_USEC, Min_, now()
FROM OrdersStream;
CREATE OUTPUT STREAM TotalOrdersStream (eventTime INTEGER, OrderId INTEGER,
ProductName INTEGER, SUM_Quantity INTEGER, TV_SEC INTEGER, TV_USEC INTEGER, TS_2
TIMESTAMP, TS_3 TIMESTAMP); 
INSERT INTO TotalOrdersStream (eventTime, OrderId, ProductName, SUM_Quantity,
TV_SEC, TV_USEC, TS_2, TS_3)
SELECT eventTime, OrderId, ProductName, SUM(Quantity), TV_SEC, TV_USEC, TS_2,
now() 
FROM OrdersLocalStream KEEP 1 MINUTES GROUP BY ProductName;
INSERT INTO TotalOrdersTable (TS, OrderId, ProductName, SUM_Quantity, TV_SEC,
TV_USEC, TS_2, TS_3, TS_4)
SELECT TotalOrdersSeq.NextVal, OrderId, ProductName, SUM_Quantity, TV_SEC,
TV_USEC, TS_2, TS_3, now() AS TS_4 
FROM TotalOrdersStream;
END

A.2.6 MaxStream - Enrich&Persist - MONITORING SELECT

SELECT /*+ MONITOR (TOTALSALES)*/ event_time, OrderID, ProductName, Sum_Quantity,
TV_SEC, TV_USEC, TS FROM TOTALSALES, Product P
WHERE ProductID=P.ProductID AND TS > ? ORDER BY TS
Appendix B

Experiment Result Data of the Graphs

Table B.1: MaxStream - Input Rate vs Latency, Figure 6.1

<table>
<thead>
<tr>
<th>Input Rate</th>
<th>End-to-End Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0294143046904272</td>
</tr>
<tr>
<td>10</td>
<td>0.10678223950056756</td>
</tr>
<tr>
<td>50</td>
<td>0.027221791224803678</td>
</tr>
<tr>
<td>100</td>
<td>0.01728692753253383</td>
</tr>
<tr>
<td>200</td>
<td>0.011294625612189439</td>
</tr>
<tr>
<td>250</td>
<td>0.010597890764989635</td>
</tr>
<tr>
<td>300</td>
<td>0.008825082941799378</td>
</tr>
<tr>
<td>350</td>
<td>0.007630213838614157</td>
</tr>
<tr>
<td>400</td>
<td>0.00760948764683674</td>
</tr>
<tr>
<td>450</td>
<td>0.007035169870341845</td>
</tr>
<tr>
<td>500</td>
<td>0.006907819050895838</td>
</tr>
</tbody>
</table>

Table B.2: MaxStream - Batch Size vs Throughput, Figure 6.2

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>666</td>
</tr>
<tr>
<td>5</td>
<td>1395</td>
</tr>
<tr>
<td>10</td>
<td>2003</td>
</tr>
<tr>
<td>50</td>
<td>2665</td>
</tr>
<tr>
<td>100</td>
<td>2751</td>
</tr>
<tr>
<td>250</td>
<td>2740</td>
</tr>
<tr>
<td>500</td>
<td>2765</td>
</tr>
</tbody>
</table>
Appendix B. *Data of the Graphs from Chapter 6*

**Table B.3: MaxStream - Batch Size vs Latency, Figure 6.3**

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00742596531766204</td>
</tr>
<tr>
<td>5</td>
<td>0.00745638980848515</td>
</tr>
<tr>
<td>10</td>
<td>0.008603489922799516</td>
</tr>
<tr>
<td>50</td>
<td>0.025848468273561645</td>
</tr>
<tr>
<td>100</td>
<td>0.04421160527475092</td>
</tr>
<tr>
<td>250</td>
<td>0.0987991723434273</td>
</tr>
<tr>
<td>500</td>
<td>1.0574009772603437</td>
</tr>
</tbody>
</table>

**Table B.4: MaxStream - Throughput vs Latency, Figure 6.4**

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>666</td>
<td>0.00742596531766204</td>
</tr>
<tr>
<td>1395</td>
<td>0.00745638980848515</td>
</tr>
<tr>
<td>2003</td>
<td>0.008603489922799516</td>
</tr>
<tr>
<td>2665</td>
<td>0.025848468273561645</td>
</tr>
<tr>
<td>2751</td>
<td>0.04421160527475092</td>
</tr>
<tr>
<td>2740</td>
<td>0.0987991723434273</td>
</tr>
<tr>
<td>2765</td>
<td>1.0574009772603437</td>
</tr>
</tbody>
</table>

**Table B.5: Coral8 - Input Rate vs Latency, Figure 6.5**

<table>
<thead>
<tr>
<th>Input Rate</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.02676757457627119</td>
</tr>
<tr>
<td>10</td>
<td>0.023286010736842035</td>
</tr>
<tr>
<td>50</td>
<td>0.028161622101525022</td>
</tr>
<tr>
<td>100</td>
<td>0.027690623270385552</td>
</tr>
<tr>
<td>200</td>
<td>0.3517879676129104</td>
</tr>
<tr>
<td>250</td>
<td>3.0804514859395242</td>
</tr>
<tr>
<td>300</td>
<td>37.49891504502172</td>
</tr>
</tbody>
</table>

**Table B.6: Coral8 - Batch Size vs Throughput, Figure 6.6**

<table>
<thead>
<tr>
<th>Input Rate</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>8.9703977779833487</td>
</tr>
<tr>
<td>50</td>
<td>43.125</td>
</tr>
<tr>
<td>100</td>
<td>84.37837837837837</td>
</tr>
<tr>
<td>200</td>
<td>154.58227848101265</td>
</tr>
<tr>
<td>250</td>
<td>187.15151515151516</td>
</tr>
<tr>
<td>300</td>
<td>208.5625</td>
</tr>
</tbody>
</table>
Table B.7: Coral8 - Batch Size vs Latency, Figure 6.7

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0267675457627119</td>
</tr>
<tr>
<td>10</td>
<td>0.023286010736842035</td>
</tr>
<tr>
<td>50</td>
<td>0.028161622101525022</td>
</tr>
<tr>
<td>100</td>
<td>0.027690623270385552</td>
</tr>
<tr>
<td>200</td>
<td>0.3517879676129104</td>
</tr>
<tr>
<td>250</td>
<td>3.0804514859395242</td>
</tr>
<tr>
<td>300</td>
<td>37.49891504502172</td>
</tr>
</tbody>
</table>

Table B.8: Coral8 - Throughput vs Latency, Figure 6.8

<table>
<thead>
<tr>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0267675457627119</td>
</tr>
<tr>
<td>8.970397779833487</td>
<td>0.023286010736842035</td>
</tr>
<tr>
<td>43.125</td>
<td>0.028161622101525022</td>
</tr>
<tr>
<td>84.37837837837837</td>
<td>0.027690623270385552</td>
</tr>
<tr>
<td>154.58227848101265</td>
<td>0.3517879676129104</td>
</tr>
<tr>
<td>187.15151515151516</td>
<td>3.0804514859395242</td>
</tr>
<tr>
<td>208.5625</td>
<td>37.49891504502172</td>
</tr>
</tbody>
</table>

Table B.9: Coral8 - Poll Interval vs Latency, Figure 6.9

<table>
<thead>
<tr>
<th>Poll Interval</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^-3</td>
<td>0.02882048449039885</td>
</tr>
<tr>
<td>10^-2</td>
<td>0.027901218918918915</td>
</tr>
<tr>
<td>10^-1</td>
<td>0.04004835499999998</td>
</tr>
<tr>
<td>1</td>
<td>0.026218784615384617</td>
</tr>
<tr>
<td>10</td>
<td>0.036482193078324246</td>
</tr>
<tr>
<td>10^2</td>
<td>0.12307695088676662</td>
</tr>
<tr>
<td>10^3</td>
<td>1.403529908615998</td>
</tr>
</tbody>
</table>

Table B.10: Coral8 - Max Delay vs Latency, Figure 6.10

<table>
<thead>
<tr>
<th>Max Delay</th>
<th>Latency</th>
<th>std-err</th>
<th>std-dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.023963066137566148</td>
<td>0.000478616749</td>
<td>0.009305373140502513</td>
</tr>
<tr>
<td>10</td>
<td>0.051300500000000001</td>
<td>0.001033036740265457</td>
<td>0.008264293922123656</td>
</tr>
<tr>
<td>10^2</td>
<td>0.32144389189189176</td>
<td>0.000608133232</td>
<td>0.006407081170349551</td>
</tr>
<tr>
<td>10^3</td>
<td>3.0187310166666665</td>
<td>0.000397649544</td>
<td>0.004356032504820797</td>
</tr>
</tbody>
</table>

Table B.11: MaxStream - Selectivity vs Latency, Figure 6.11

<table>
<thead>
<tr>
<th>Selectivity Percentage</th>
<th>Latency</th>
<th>std-err</th>
<th>std-dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>0.0588922649656526</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10%</td>
<td>0.0028011974565641234</td>
<td>1.9239207304803872E-4</td>
<td>0.006062640250197913</td>
</tr>
<tr>
<td>25%</td>
<td>0.006136714919754624</td>
<td>8.179160681320392E-5</td>
<td>0.00496576777141119</td>
</tr>
<tr>
<td>50%</td>
<td>0.007084361509846916</td>
<td>1.523673827983237E-4</td>
<td>0.014088010996052489</td>
</tr>
<tr>
<td>100%</td>
<td>0.007505426213506085</td>
<td>9.59604996730525E-5</td>
<td>0.010186303224591346</td>
</tr>
</tbody>
</table>
### Table B.12: Coral8 - Selectivity vs Latency, Figure 6.12

<table>
<thead>
<tr>
<th>Selectivity Percentage</th>
<th>Latency</th>
<th>std-err</th>
<th>std-dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>0.02283393181181816</td>
<td>7.19653938095976E-4</td>
<td>0.00675124726478536</td>
</tr>
<tr>
<td>10%</td>
<td>0.02483726976741866</td>
<td>5.57676912515934E-4</td>
<td>0.008177160116674806</td>
</tr>
<tr>
<td>25%</td>
<td>0.02483726976741866</td>
<td>5.57676912515934E-4</td>
<td>0.008177160116674806</td>
</tr>
<tr>
<td>50%</td>
<td>0.02469676836158192</td>
<td>6.03401262638943E-4</td>
<td>0.008027731673872373</td>
</tr>
<tr>
<td>100%</td>
<td>0.03110129714285715</td>
<td>8.52227721154168E-4</td>
<td>0.01127391305284164</td>
</tr>
</tbody>
</table>

### Table B.13: Coral8 - Latency of Subintervals, Figure 6.13

<table>
<thead>
<tr>
<th>Experiment Type</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coral8-Persist</td>
<td>0.01546</td>
<td>0.00235</td>
<td>0.0</td>
<td>0.00614</td>
</tr>
<tr>
<td>Coral8-Enrich&amp;Persist</td>
<td>0.02064</td>
<td>0.00240</td>
<td>0.0</td>
<td>0.00806</td>
</tr>
</tbody>
</table>

### Table B.14: MaxStream - Latency of Subintervals, Figure 6.14

<table>
<thead>
<tr>
<th>Experiment Type</th>
<th>(I1+I2)</th>
<th>(I3+I4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxStream-Persist</td>
<td>0.00420</td>
<td>0.00322</td>
</tr>
<tr>
<td>MaxStream-Enrich&amp;Persist</td>
<td>0.00544</td>
<td>0.00179</td>
</tr>
</tbody>
</table>

### Table B.15: MaxStream - Enrich-Persist- Scale Factor vs Latency, Figure 6.15

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.007228869263975068</td>
</tr>
<tr>
<td>0.5</td>
<td>0.0072055319272445574</td>
</tr>
<tr>
<td>1</td>
<td>0.0071055319272445574</td>
</tr>
<tr>
<td>10</td>
<td>0.007497034127891646</td>
</tr>
</tbody>
</table>

### Table B.16: Coral8 - Enrich-Persist- Scale Factor vs Latency, Figure 6.16

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.029218784615384617</td>
</tr>
<tr>
<td>0.5</td>
<td>0.031137377777777794</td>
</tr>
<tr>
<td>1</td>
<td>0.032177763005780355</td>
</tr>
<tr>
<td>10</td>
<td>0.03683800573065905</td>
</tr>
</tbody>
</table>

### Table B.17: Coral8 - Scale Factor vs Latency (no index), Figure 6.17

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Latency</th>
<th>std-err</th>
<th>std-dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.03545151162790696</td>
<td>4.1098452435010943E-4</td>
<td>0.007622627372751358</td>
</tr>
<tr>
<td>0.3</td>
<td>0.1291004147727273</td>
<td>0.004247605228918079</td>
<td>0.07696213802890306</td>
</tr>
<tr>
<td>0.5</td>
<td>0.21356399750498975</td>
<td>0.0016942345379241592</td>
<td>0.07584420255806082</td>
</tr>
<tr>
<td>0.7</td>
<td>0.8723715016722413</td>
<td>0.009198902007981383</td>
<td>0.2249503040561236</td>
</tr>
<tr>
<td>0.9</td>
<td>1.329463082935153</td>
<td>0.006712818019914547</td>
<td>0.812505923126276</td>
</tr>
<tr>
<td>Experiment Type</td>
<td>End-to-End Latency</td>
<td>std-err</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>Persist</td>
<td>0.003821467361579057</td>
<td>0.00046224030521669745</td>
<td></td>
</tr>
<tr>
<td>EnrichPersist</td>
<td>0.010941994243673144</td>
<td>0.0007676144837195222</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


