Synchronizing data stream processing

Mohammad Siddique Fawad Qureshi
University of Wollongong

Synchronizing Data Stream Processing

A thesis submitted in fulfillment of the requirements for the award of the degree

Master of Computer Science (Research)

from

UNIVERSITY OF WOLLONGONG

by

M. S. Fawad Qureshi

B.Sc Computer Science – University of Sindh
M.Sc Computer Science – University of Sindh

SITACS
School of Information Technology and Computer Science

2007

Certification

I, M. S. Fawad Qureshi, declare that this thesis, submitted in fulfillment of the requirements for the award of Master of Computer Science, in the School of Information Technology and Computer Science, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

M. S. Fawad Qureshi

Date: 30 March 2007

Table of Contents

List of Figures

Chapter 3

Chapter 4

List of Tables

List of Publications

Publications arising from this thesis:

Qureshi, M. S. F. and Getta, J. R. (2007): Synchronizing Data Stream Processing. *Proc. IASTED International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, 233 – 238.

Peer-reviewed proceedings of an international conference.

Abbreviations

AQP……..Adaptive Query Processing

ART……...Average Response Time

CQL……Continuous Query Language

CQP……..Continuous Query Processing

DBMS……Data Base Management System

DSMS…….Data Stream Management System

DSP…..……Data Stream Processing

DSPN…..……..……Data Stream Processing Network

MDR……Maximum Data Rate

PQP……..Pipelined Query Processing

QoS……..Quality of Service

QP…….…..Query Processor

SPE……...…Stream Processing Engine

STREAM……Stanford Stream Data Manager

SQL……..Structured Query Language

TQL…….Tapestry Query Language

XML……Extended Markup Language

.

to my beloved father

Haji Muhammad Saleem Qureshi

Abstract

Synchronization of data stream processing has a significant impact on performance of systems where processing of long sequences of data items needs to be done simultaneously. In earlier works on stream processing, synchronization has been discussed to a limited extent or has been completely overlooked. This work describes a formal model of synchronization in a data stream processing network. We use a notation of data stream processing networks to identify circumstances that necessitate synchronization. We also express processing of groups of data items in terms of database transactions within a data stream processing network. A technique similar to timestamp ordering of database transactions is used to solve the problems. A solution is presented as a set of rules that govern processing of groups of data items. A proof of correctness has been provided for the strategy used to solve the problems.

Acknowledgments

I would like to thank my supervisor Dr. Janusz R.Getta for his invaluable guidance, support, and patience. He helped me in defining a suitable problem for my thesis, and always provided productive suggestions in the course of the writing process.

My special thanks to Professor John Fulcher and Dr. Heather Jamieson for their encouragement during the revision process.

I would like to thank my family and friends for their continuous support and help in the best way possible.

Synchronizing Data Stream Processing

Chapter 1

Introduction

In a number of applications data arrives in the form of a long sequence of items spanned over a long period of time. Such sequences, also called data streams require new data processing techniques different from techniques developed over the last two decades for relational database systems. These streams require processing in an online manner (Rastogi 2002) where the arrival of a new data item immediately triggers its processing. Data streams are generated in applications including sensor networks, satellite and traffic monitoring systems, security monitoring, financial services, weather measurements and many other real-time applications. In these applications, arrival rate of data is very high and rapid.

Effective processing of data streams is a difficult problem. Traditional database management systems (DBMS's) are incapable of efficient handling of data streams (Arsu et al. 2002) due to a lack of online data stream processing algorithms and architectures that prefer batch oriented data processing. Many fundamental assumptions that are basis of database systems are not valid for stream-oriented systems (Tian, F. and DeWitt, D. J. 2003). The continuous arrival of data requires something more than a DBMS. As data arrives in a stream it needs to be analyzed. The arrival rate of data is very high in a stream and storing all the data will result in loss of memory and time.

The debate in the data management research community is about developing a general purpose data stream management system (DSMS). Some researchers refer to DBMS as ill equipped (Babu, S. and Widom, J. 2001) for processing data streams. Few resist updating today's DBMS in the presence of data streams. In that case, many aspects of DBMS need to be reconsidered for processing streaming data. Vossough, E. (2004), claims there are no methods for a DBMS to handle synchronized processing of data streams.

The increase of streaming data will require a data management system that fulfills all the requirements for processing data streams. One of the main factors in the increase of streaming data is sensor technology. Sensor technology is decreasing in cost day by day and becoming more affordable. This will result in sensors embedded in almost every device and thereby generating more streaming data. For example when we have sensor's embedded in our mobiles then just by running a simple query, we would be able to find the current location of someone instead of calling and asking.

The concept of DSMS is of a dedicated system for processing a new class of data processing applications, called stream-based applications. The work done in regard of stream processing ranges from efficient algorithms for data streams to complete data stream management systems such as Borealis (Abadi D. et al. 2005), STREAM (Babcock, B. et al. 2003), Aurora (Carney. et al. 2002 and Cherniack, M. et al. 2003), NiagaraCQ (Chan, J. 2000), Telegraph (Chandrasekaran, S. et al. 2003), Tribeca (Sulliovan, M. et al. 1996) and few others.

1.1 The Problem

In a data stream processing network, if data items are processed one by one - i.e. sequentially - then this is not practical. The processing rate of data items will be extremely

slow. Even though sequential processing doesn't have any problems regarding synchronization, it is not feasible to apply this technique to a data stream. On the other hand, if data items are processed at the same time - i.e. simultaneously - then this makes processing more effective. The processing gets started immediately as data items arrive. But simultaneous processing involves circumstances where processing needs to be synchronized, otherwise it may result in incorrect execution of data items. We use a notation of data stream processing networks to identify circumstances that necessitate synchronization.

1.2 Strategy and Objectives

The synchronization method proposed in this work is defined as a set of rules that govern the processing of groups of data items. The rules synchronize the processing of data items at circumstances identified by this work. To prove the correctness of our approach, we will express processing of data items in a data stream network in terms of database transactions. The method used to check the execution of transactions for correctness will then be applied to our results.

The main objectives of this research project are:

- To perform a literature review on related work done in the area of stream processing and study previously proposed synchronization techniques for data streams.
- To propose a new class of networks for processing data items in a stream.
- To identify circumstances which require synchronization while processing data items simultaneously.

- To propose a solution in order to synchronize processing of groups of data items in the network.
- To prove the correctness of the solution.

1.3 Outline of the Thesis

Following is the brief description of the remaining chapters presented in this thesis:

- **Chapter 2 Background and Related Work**

  This chapter presents a background on stream processing and also some related work in respect to data streams.

- **Chapter 3 Data Stream Processing Network**

  This chapter proposes a new class of networks called data stream processing networks. It defines the basic structure of the network and also computations performed within the network. It describes an entity called windows, together with details regarding some operations.

- **Chapter 4 Synchronization Problems**

  This chapter identifies circumstances that require synchronization in a data stream processing network. It describes motivating examples showing consequences of not synchronizing the processing. This chapter shows how processing of data items in a data stream network can be expressed in terms of database transactions and interprets the motivating examples in terms of transactions.

- **Chapter 5 Synchronization Strategy and Correctness**

  This chapter presents a solution as a set of rules to synchronize processing of data items in a data stream processing network. It provides description of a regulator and collector and applies the rules to the motivating examples of chapter 4. It also provides a proof of correctness for the rules.

Chapter 2

Background and Related Work

The following review of the literature relevant to synchronization of data stream processing, is divided into two parts. The first part consists of three sections i.e. section 2.1 Data Stream Processing, section 2.2 Adaptive Query Processing and section 2.3 Continuous Query Processing. These sections review the background research relevant to processing of data streams.

The second part section 2.4 Synchronization Techniques and Related Work, examines the relevance of the background research with reference to synchronization of data stream processing.

2.1 Data Stream Processing (DSP)

The two major areas that contribute towards DSP are adaptive query processing (Levy 2000) and continuous query processing (Terry, D. B. et al. 1992). At the moment, numerous research works provide information on different aspects of DSP. A comprehensive review of major contributions is included in (Arsu, A. et al. 2002), (Babcock, B. et al. 2003), (Carney. et al. 2002), (Cranor, C. et al. 2003), (Das, A. et al. 2003), (Stonebraker, M. et al. 2003), and (Tucker, P. A. et al. 2003). A review of recent work is described in (Babcock, B. et al. 2002) and (Golab and Özsu 2003).

The approach in data stream processing is to start execution as soon as data item arrives in a stream and to process the maximum amount of data items at the same time. According to (Getta and Vossough 2004), methods for DSP should be *reactive*, *continuous*, *adaptable* and *efficient*. *Reactivity* means that processing of a data item should start immediately as it arrives in a stream. *Continuity* concerns the time to time recomputations of applications in order to update according to the changing contents of the stream. *Adaptability* allows for dynamically modifying a processing plan in reply to external factors, for example rapid change in the frequency of a stream. *Efficiency* concerns dictate that data processing rate must be higher than the data propagation rate.

A brief summary of some of the related work done in the area of DSP is as follows:

- `Aurora` is a DSMS built for a large distributed scale. It is a dataflow system that allows users to build query plans by arranging boxes (operators) and arrows (dataflow among operators). A box in a system accepts input streams and produces one or more output streams as arrows. Final outputs from boxes are streamed to other applications. The system represents a set of continuous queries as a loop-free directed graph of stream oriented operators. This is similar to the execution performed within our network.

  Aurora comprises operators such as a simple unary operator (Filter), a binary merge operator (Union), a mapping operator (Map), a time bound window sort (WSort), an aggregation operator (Tumble) and a join (Join). The users are allowed to define their own declarative queries in SQL and compile those queries into box and arrow model.

The main components of the Aurora system are the scheduler, storage manager and load shedder. The heart of the system is the scheduler that decides which operators to execute and in which order to execute them. It also gives special attention to issues like reducing operator scheduling and invocation overheads. The storage manager is designed for storing ordered queues of tuples and buffers the queues when main memory runs out. The load shedder detects and handles overload situations. Load shedding (Tatbul, N. et al. 2003) is an important strategy for adapting to higher rates of data arrival. The input tuples are dropped based on some criterion such as quality of service (QoS) in Aurora. Each query in Aurora defines the processing requirements and QoS specifies query performance requirements.

- QUAY (Lee, Ken C.K. et al 2004): QUAY (pronounced "key" meaning a platform over streams) is a data stream processing system that uses the chunking technique (Deshpande, A. et al. 1998 and Lee, Ken C.K. et al. 2002) for indexing queries. The chunking technique clusters and indexes both queries and data records in a unified way as chunks. This approach of indexing queries is different to approaches proposed in Eddy and NiagaraCQ. An adaptive selection-join arrangement for a huge number of selection-join queries is also proposed for processing window join operation from stream sources. The system architecture of QUAY consists of four cooperating modules, namely stream modules, a cross chunk join operation, a query registry and a set of programming interfaces. The stream module is for serving each individual data stream. The cross chunk join processor evaluates join queries. The query registry is a repository for query information. The set of programming interfaces is for providing access from stream applications.

- Borealis: A distributed stream processing engine and a successor to Aurora. Borealis is considered to be the first of the second generation stream processing engines (SPE's), while the previously proposed SPE's are assumed to be first generation. Borealis inherits the Aurora model (boxes and arrows) for specifying queries. An important addition to the Aurora query model is that Borealis has the capability of changing the semantics of a box on the fly. The boxes in Borealis are provided with special control lines. These lines carry control messages that contain revised box parameters and functions for changing box behaviour.

  The group of continuous queries submitted to Borealis can be seen as one big network of operators where processing is distributed to multiple sites. Each site runs a Borealis server where Query Processor (QP) - which is single site processor - forms the core piece for query execution to take place. The major run-time components of QP consist of the priority scheduler, box processors, storage manager and load shedder. The priority scheduler determines the order of box execution based on tuple priorities. The box processor changes the behaviour of the box, and each type of box is provided with one box processor. The storage manager is responsible for storage and retrieval of data. The load shedder discards low-priority tuples when the node is overloaded. Xing, Y. et al (200) presents a correlation based load distribution algorithm for Borealis that aims at avoiding overload and minimizing end-to-end latency and maximizing load correlation.

- STREAM: The STanford stREam datA Manager is a general purpose and a relation-based DSMS with an emphasis on memory management and approximate

query answering. The functionality and performance of STREAM is similar to a traditional DBMS but allows some or all data to be managed in the form of unbounded data streams. It supports CQL (i.e. continuous query language) as a declarative query language. The system operates in a manner that each operator works on a set of input data streams and produces an output stream. Data may be saved in a component called Scratch or may be discarded to the component called Throw. The current results are stored in the component called Store. The results may also be sent to output stream and are served as an input to other operators. The system uses a global scheduler that controls query execution operators. The scheduler uses a simple round-robin scheme for scheduling operators. The use of a scheduler not only minimizes total queue size of unpredictable streams but also reduces for example inaccuracy, latency and memory use.

- Optimization of Data Steam Processing (Getta and Vossough 2004): This work describes a formal model for processing data streams and describes optimization techniques that can be applied to the model. One of the optimization techniques described is for efficient synchronization of elementary operations on data streams. Processing of data streams have been distinguished between logical and dataflow levels and is one of the main contributions of this work. This work considers using a scheduler for simultaneous processing of data streams.

2.2 Adaptive Query Processing (AQP)

Adaptive query processing allows for dynamically modifying a processing plan in reply to the external environment. The origins of AQP can be traced to the Telegraph Project.

Examples of this processing approach include Rivers (Arpaci–Dusseau, R. H. et al. 1999) and Eddies (Avnur, R. and Hellerstein, J. 2000). The key to telegraph is a continuously adaptive query processing engine. The engine gathers feedback from the environment and uses that feedback for determining its behaviour. A feedback to an adaptive system makes the processing more efficient. It allows the system to make better decisions by observing the results of multiple decisions. According to Babu, S. and Bizarro, P. (2005)**,** adaptive query processing can be divided into the following three families:

- Plan-based AQP Systems: AQP for traditional plan-based systems for example Tukwila (Ives, Z. et al. 1999). The Tukwila system supports AQP by performing dynamic data integration over autonomous data sources.
- Continuous-Query –based AQP systems: AQP for long running continuous queries over data streams for example STREAM and NiagaraCQ.
- Routing-based AQP systems: AQP for DBMS's and continuous queries based on adaptive tuple routing where tuples are routed individually through operators for example Rivers and Eddies.

A brief summary of some of the related work done in regard to AQP is as follows:

- `StreaMon`: An Adaptive Engine for Stream Query Processing (Babu, S. and Widom, J. (2004)**)**: It is the AQP engine of STREAM prototype data stream management system that uses CQL as the query language. The system consists of three generic components, an executor, a profiler and a reoptimizer. An executor runs query plans for producing results, a profiler collects and maintains statistics about the stream, and a reoptimizer ensures that the plans and memory usage are

most efficient for the current input. StreaMon also uses several techniques for supporting AQP including adaptive memory minimization, adaptive join ordering and adaptive caching for joins. Adaptive memory minimization reduces run-time memory requirements for continuous queries by exploiting stream data. Adaptive join ordering is used to pipeline multiway stream joins.

- Tuple Routing Strategies for Distributed Eddies (Tian, F. and DeWitt, D. J. 2003): Routing tuples between operators of a distributed stream query plan have been used in many DSMS's as an adaptive query optimization technique. A routing policy can have a significant impact on the performance of a system. The original paper on Eddies introduced the idea of routing tuples between operators as a form of query optimization. This work extends the concept of an Eddy to a distributed environment. This work also proposes and evaluates some practical routing policies for a distributed stream management system. Two performance metrics - average response time (ART) and maximum data rate (MRD) - are defined for response time of a tuple when it enters and leaves operators and system throughput.

  Eddy is a query processing operator that dynamically chooses the order of tuples in a query plan by making independent routing decisions per tuple. The making of routing decisions per tuple may be too expensive. Deshpande, A. (2004) performs the initial study of the overheads of Eddies.

- Adaptive Stream Resource Management Using Kalman Filters (Jain, A. et al 2004): This work perceives stream resource management as fundamentally a filtering problem and proposes Kalman Filters as a general and adaptive filtering solution for conserving resources. The Kalman Filter is a stochastic, recursive data filtering

algorithm that can be used in a large variety of data-streaming applications. It is capable of adapting to various stream characteristics, sensor noise and time variance. A significant boost in performance can be achieved as the method does caching of dynamic procedures that can predict data reliably at the server without user involvement.

- Adaptive Ordering of Pipelined Stream Filters (Babu, S. et al. 2004**)**: The focus of this work is on the problem of ordering the filters adaptively for minimizing processing cost in an environment where stream and filter characteristics may change considerably. This work proposes an algorithm called A-Greedy (for Adaptive Greedy) that converges to an ordering within a small constant factor of optimal. One of the main features of the algorithm is that it monitors and responds to selectivities that are correlated across non independent filters. This provides a strong quality guarantee but may result in run-time overhead. This work also identifies a three-way tradeoff among provable convergence to good ordering, run-time overhead and speed of adaptivity. A collection of different variants of A-Greedy are also developed that lie at different points for the tradeoff spectrum.

- Adaptive Filters for Continuous Queries Over Distributed Data Streams (Madden, S. et al. 2002): This method reduces the overhead of a centralized processor that monitors continuous queries in an environment where distributed data sources continuously cause stream updates. The filters installed at remote data sources adapt to changing conditions for minimizing stream rate while guaranteeing that all continuous queries receive the necessary updates for providing answers of adequate precision. This method enables an application to trade precision for communication

overhead at a fine granularity by individually adjusting the precision constraints of continuous queries over streams. This work also demonstrates the effectiveness of the method for achieving low communication overhead and compares the method with other similar methods. This method relies on the use of system schedulers.

2.3 Continuous Query Processing (CQP)

Continuous and one-time queries (Terry, D. B. et al. 1992) are two important types of queries that can be related to a data stream model. One-time queries are a class of queries that includes traditional DBMS queries. They are evaluated once over a point in time snapshot of the data set while providing answers to the user. In comparison, continuous queries are evaluated continuously as data streams continue to arrive. The answer of a continuous query always reflects the stream data seen so far and are stored or updated as new data arrives.

A brief summary of some of the related work done in area of CQP is as follows:

- Dynamic Plan Migration for Continuous Queries Over Data Streams (Zhu, Y. et al. 2004): The "Dynamic Plan Migration" is about transforming from one query plan to a semantically similar but more effective plan. An effective migration plan ensures that results have not been altered during or after the migration process. For solving problems dynamically during migration, two types of strategies i.e. moving state strategy and parallel track strategy have been proposed. The moving state strategy assures timestamp order preservation for the tuples. The parallel track strategy is used for having correct results without any changes. The problems defined are for example duplication of tuples and incorrect order of results.

- Data Stream Management for Historical XML Data (Bose, S. 2004): This work presents a framework for continuous querying of time-varying streamed XML data. The model defined differs from other data stream processing models in terms of query language and data processing. The query language used is XCQL, which works on multiple XML data streams and is able to correlate data. This framework considers input data from a wide variety of streaming data sources and has the ability to synchronize between streams by issuing coincidence queries.
- `Tapestry` (Terry, D. B. et al. 1992): This system used continuous queries for content-based filtering over an append-only database. The database was created for email and bulletin board messages. A subset of SQL with some restrictions called Tapestry query language (TQL) was used to provide efficient evaluation and append-only query results.
- `Tribeca` is one of the early special purpose systems developed for on-line traffic monitoring. It provided restricted querying capabilities over network packet streams with a simple query language. It supports windows and a set of operators adapted from relational algebra.
- `NiagaraCQ` is a continuous query system that permits continuous XML queries to be presented over dynamic web content. It was built in regard to the Niagara Internet Query System (Naughton, J. F. et al. 2001). It considers continuous queries that transform a passive web page into an active environment that can support millions of queries. This model introduces predicate grouping and group optimization techniques for addressing scalability in terms of the number of queries.

Grouping queries shares common computations and can reduce input/output cost. NiagaraCQ uses an incremental group optimization strategy which dynamically regroups queries. If implemented in an internet environment NiagaraCQ uses a system of continuous query mangers, event detectors and data managers. The system performs well in an internet environment but can be very costly in a distributed environment because of the high volumes of data.

2.4 Synchronization Techniques and Related Work

Synchronization is a topic which still needs to be given more consideration in presence of data streams. In previous work on stream processing, synchronization has been discussed either to a limited extent or has been completely overlooked. Vossough, E. (2004), claims there are no methods for a DBMS to handle synchronized processing of data streams.

As stream applications rely on shared data and computations, where data may be in a table is updated by one query and read by the other. This sharing may result in data inconsistencies. The uniqueness of this work is that mainly focus on synchronization and identifies circumstances where processing needs to be synchronized.

After going through some of the major work performed for stream processing; now we examine the relevance of the background research with reference to synchronization of data stream processing:

- In STREAM operators are scheduled for execution by a central scheduler. During execution, an operator reads data from its input queues and writes results to its output queues. The scheduler dynamically determines an execution period for an

operation. When the period expires the operator returns the control back to the scheduler. The period of execution may be based on time or number of tuples. A need for synchronization within operators while scheduling multiple query plans is discussed but overlooked in presence of other major issues like memory management.

The heart of our solution is the regulator which is one of the synchronization assistants. The goals of using a regulator are to synchronize the processing and to remove any possibility that leads towards wrong results. The synchronization is performed in a way that an operation requests the regulator for executing at circumstances where processing needs to be synchronized. The regulator then performs certain checks in order to allow or stop a data item from executing further. We use another synchronization assistant which is a container called collector. It is a special purpose container which collects timestamps of data items as a data item is allocated a timestamp. The checks are performed by the regulator on the collector.

- The work by Getta and Vossough (2004) is geared towards optimizing stream processing where one of the optimizing techniques is regarding efficient synchronization of elementary operations on data streams. A scheduler is defined which can be considered as a formal model for concurrent processing of data streams and not as a hypothetical implementation. Running computations are represented in terms of a scheduling graph. The graph is dynamically maintained during the computations of dataflow expressions. Scheduling quality in the graph is measured by the total number of dashed edges. If the graph contains no edges then it implies no conflicts and no blocking of operations.

One of the similarities to our work is of showing computations on data streams implemented as flows of data items among the elementary operations. The concept of translating an expression into a data stream processing network is also similar to our work. The only difference is that we first translate an expression into a syntax tree and then to a network. This work derives data stream expressions from the syntax tree and then uses those expressions for creation of a network.

- Routing Tuple is used in many DSMS's as an adaptive query optimization technique for distributed stream query plan between operators. A model for distributed stream query plan is defined as a set of operators connected by a network where input tuples are added to a first-come first-served queue. This work describes a distributed symmetric join algorithm which implements a join operator with for two input streams – i.e. A ⋈ B. First an input tuple from stream A is processed by an operator which maintains a sliding window for stream A, then the tuple is sent to the operator that joins the tuple with window of stream B. A three way join A ⋈ B ⋈ C was also defined. Two different joins were performed on the sliding window of stream B – the right side join of A ⋈ B and a left side join of B ⋈ C.

  Eddies relates to our work as it implements a network of simple operators for processing data streams.

- In Aurora, the data flows through a loop-free system of boxes i.e. operators. The way data gets processed within the boxes is not clear. The output of the box may be shared by multiple queries or even a single query for merging intermediate computations. This is achieved through one of the boxes called WaitFor box. This

box buffers each tuple *t* on an input stream until a tuple arrives on the second input stream that with *t* satisfies *P*. The syntax of WaitFor is shown below:

WaitFor (P: Predicate, T: Timeout)

Incase of a read operation where a read must follow an update, the WaitFor buffers the read requests until a tuple output by the Update box indicates that read operation can proceed. Now in a situation where the Update box may not produce any results then the WaitFor box will wait forever. No strategies are defined for such kind of possibilities.

- Adaptive caching for multiway joins avoids recomputations of intermediate result and is similar to one of the problems described by this work. One more similarity to our work is the use of sliding windows.

- Dynamic Plan Migration for Continuous Queries Over Data Streams (Zhu, Y. et al. 2004) is about transforming from one query plan to a semantically similar but more effective plan. They describe problems which are conceptually similar to the problems defined in our work but are in the context of dynamic plan migration. The problems defined are for example duplication of tuples and incorrect order of results. We use a technique similar to timestamp ordering of database transactions for solving the problems. A solution is presented as a set of rules that govern processing of individual data items.

Chapter 3

Data Stream Processing Networks

This chapter proposes a new class of networks called Data Stream Processing Networks (DSPN). The first section defines basic structure and the main components of the network. It also deals with computations performed within the network along with details regarding some operations. The second section describes an entity called windows. The third and last section provides visualization of sample networks.

3.1 Network Model

We define a formal model for processing of groups of data items in a DSPN. The network is defined as a directed graph with set of nodes and edges. The formal definition of the network is as follows:

Definition:

A data stream processing network is a directed graph $G = (n,e)$ where n is a set of nodes - i.e. $n = (w \cup d \cup m)$ and e is a set of edges – i.e. $e = (e_1 \cup e_2 \cup e_3 \cup e_4 \cup e_5)$ where $e_1 \subseteq (w \times w)$, $e_2 \subseteq (d \times w)$, $e_3 \subseteq (w \times d)$, $e_4 \subseteq (w \times m)$, $e_5 \subseteq (m \times w)$.

In terms of nodes, node w represents a set of elementary operations, node d represents a set of windows and node m represents a set of merge points. In case of edges, $e_1 \subseteq (w \times w)$

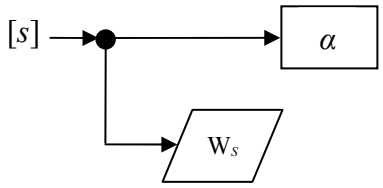


Figure 3.1: Upon arrival, a group of data items *s* is recorded to a window over an input stream.

represents flow of data between two operations, $e_2 \subseteq (d \times w)$ represents flow of data from a window to an operation, $e_3 \subseteq (w \times d)$ represents flow of data from an operation to a window, $e_4 \subseteq (w \times m)$ represents flow of data from an operation to a merge point, $e_5 \subseteq (m \times w)$ represents flow of data from a merge point to an operation.

3.1.2 Components

The network represents operations through rectangles and windows through parallelograms. A window *d* is an ordered sequence of a group of data items arriving at different times. Whenever a group of data item arrives in a stream, it is recorded to a window over an input stream as shown in figure 3.1. More details on windows are provided in section 3.2.

The network also has two special operations called merge and write. Merge is represented through a black square called a merge point *m* whereas write is denoted by a large dot called a writer. A writer records contents of every group of data items into the windows. Section 3.1.4 provides more details on merge and write.

Edges are used to represent data flows between operations, windows and merge points. For example when a writer records the contents of a data item into a window, then an edge

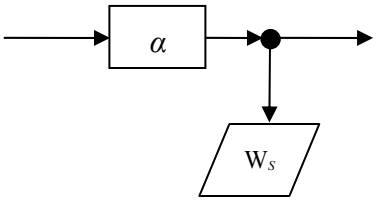


Figure 3.2: An edge from a writer to a window represents a write.

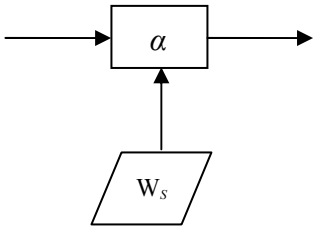


Figure 3.3: An edge from a window to an operation represents a read.

represents the data flow from that writer to the particular window as shown in figure 3.2. Similarly when an operation reads from a window then an edge represents the data flow from that window to the particular operation as shown in figure 3.3.

3.1.3 Computation

An operation in a data stream processing network accepts a group of data items as input and always produces output. The output from an operation is also a group of data items. The output from an operation becomes input for other operations.

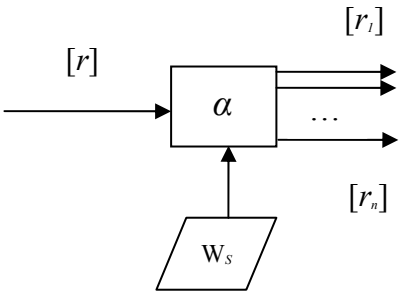


Figure 3.4: An operation with a group of data items as input and produces groups of data items as output.

This allows for parallel computation in the network as the output of an operation becomes an input for different operations at the same time. In circumstances where an operation produces an empty group of data items as output, the empty group is provided as an input to the rest of the operations in a sequence until the empty group gets recorded to the window containing the final results.

3.1.4 Operations

Figure 3.4 shows basic computation performed by an operation that accepts a group of data items *r* as input then reads from a window w_s and produces a group of outputs. Every output contains two indicators. The first indicator specifies the starting of the output whereas the second specifies completion of an output. There can be circumstances where an operation produces an empty group of data items as output but still has the two indicators.

The data stream processing network also contains the following two special operations:

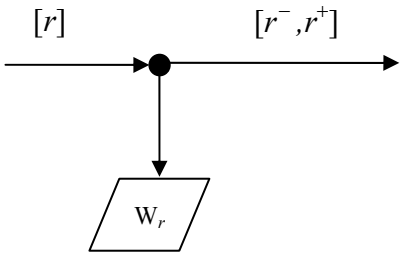


Figure 3.5: A writer records contents of a group of data items into a window.

- **Write**

The contents of every group of data items are recorded into the windows by the special operation called write. Figure 3.5 shows an example where a group of data item [r] upon arrival is recorded to a window over an input stream by the writer. The insertion of a new group of data items - i.e. positive data items r^+ - results in deletion of an old group of data items - i.e. negative data items r^-. An operation in that sequence then processes a group of positive data items r^+ and negative data items r^-, which is basically a modification of the window caused by the arrival of new group of data items.

- **Merge**

Merge is the only operation in the network which accepts more than one group of data items as input. Consider a situation where two groups of data items [r_1] and [r_2] go through the merge point. After going through merge operation, the two groups are processed as [r_1] [r_2]. The order in which two groups arrived at merge point, the merging is done similar to

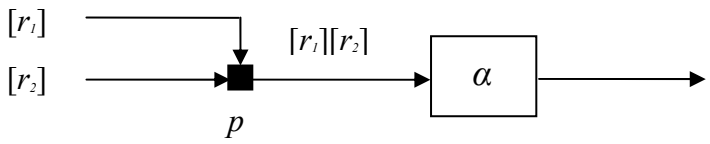


Figure 3.6: A situation where merge is performed on two groups of data items.

that order. Figure 3.6 shows $[r_1]$ and $[r_2]$ going through the merge point and then processed by operation α as $[r_1]$ $[r_2]$.

3.2 Windows

An ordered sequence of a group of data items arriving at different times is called a window. The windows used in this work are sliding windows. These windows are of fixed length and are composed of two sliding end points. The window moves as a group of data items arrives and is appended to it. This movement results in the deletion of an old group of data items from the window. The new group is referred to as a positive group of data items whereas the deleted group is referred to as a negative group of data items.

As soon as a group of data item arrives in a stream, it is recorded to a window over an input stream. There are also windows for intermediate and final results. For example if a group of data items arrives in stream r, it is recorded to the window for input data stream w_r. The intermediate result of processing of that group of data items is recorded into window w_{temp}. The final result of the group of data items is recorded into window w_{out}.

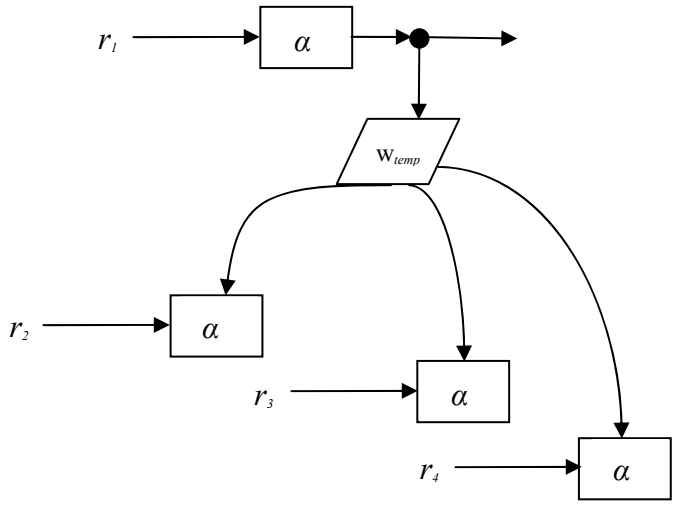


Figure 3.7: A situation where many operations read from a window.

3.2.1 Multiple Operations on a Window

It is also possible that many operations read from a window at the same time. For example figure 3.7 shows the output of data item r_1 is recorded to a window which is then read by data items r_2, r_3 and r_4.

3.3 Visualization of the Network

Figure 3.8 and 3.9 provide visualizations of sample data stream processing networks. We get these networks by translating an expression into a data stream processing network. Section 3.3.2 describes the method which is used to translate an expression with two inputs and one output into a data stream processing network.

3.3.1 Sample Networks

For the sample networks we have taken into account relational algebra and arithmetic expressions. We consider a class of expressions where an operation in the expression

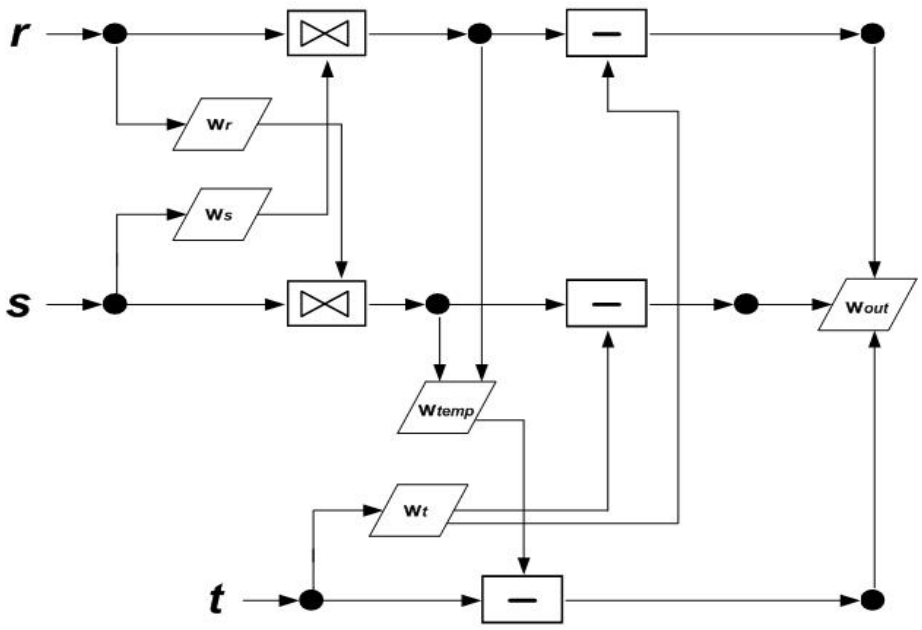


Figure 3.8: A data stream processing network for a relational algebra expression $[(r \bowtie s) - t]$.

accepts a maximum of two inputs and produces just one output. The two inputs include the input data item and the contents recorded in an output window.

The expression used for figure 3.8 is $[(r \bowtie s) - t]$, which is a relational algebra expression. This expression contains two operations i.e. join and minus and there are three operands *r*, *s* and *t*. Each operand in the expression is considered as a single stream. In terms of the sample network, group of data items arriving in stream *r*, *s* and *t* are termed as d_r, d_s and d_t respectively. All these groups are recorded in their respective windows w_r, w_s and w_t as soon as they arrive

According to the expression there is a join between the operands *r* and *s* and then minus is used between the result of join and the operand *t*. When a group of data item d_r arrives at

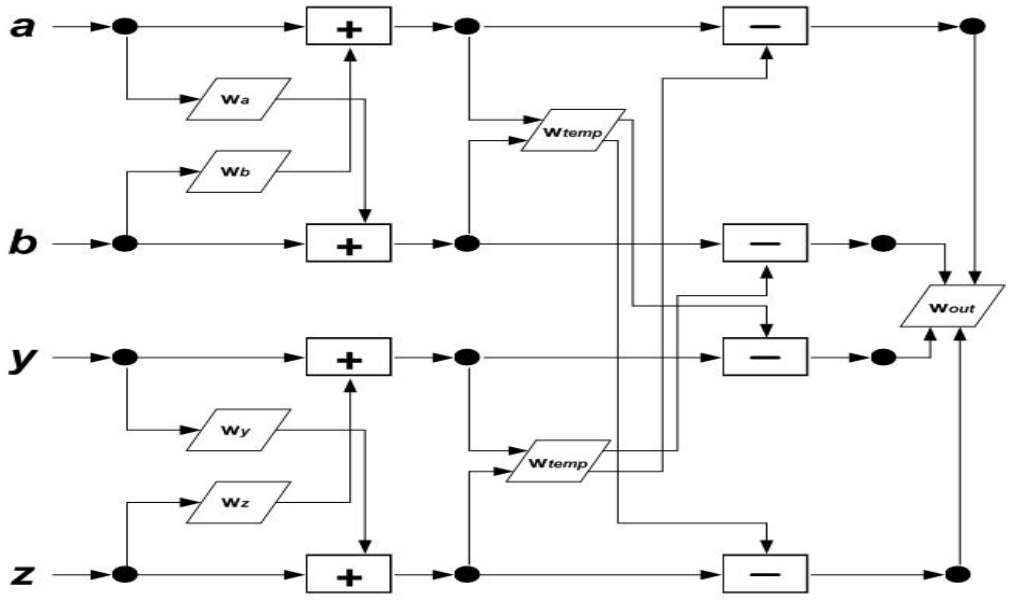


Figure 3.9: A data stream processing network for an arithmetic expression $[(a + b) - (y + z)]$.

stream *r*, join operation of d_r obtains the contents of d_s from w_s and performs join. The result of the join is then recorded in window w_{temp}. Now the minus operation of d_r obtains the contents of d_t from w_t and performs minus. The final result of the expression is recorded in window w_{out}. The same procedure is repeated at stream *s* when a group of data items d_s arrive and then final results of the expression are recorded to w_{out}. When a group of data items d_t arrive at stream *t* then minus operation of d_t obtains the result of join of d_r or d_s from w_{temp} and performs minus. The final result is then recorded in window w_{out}.

On the other hand, the expression used for figure 3.9 is an arithmetic expression $[(a + b) - (y + z)]$. It consists of three operations and four operands *a*, *b*, *y* and *z*. The three operations include two additions and one subtraction. The network constructed for this expression would be different to the sample network of figure 3.8, as the arithmetic expression for this

network contains an extra operand and operation. There will be two temporary windows used, the first will contain the result of ($a + b$) and the second will have the result of ($y + z$).

According to the expression there are two plus signs between the operands r, s and operands y, z. The minus sign is then used between the results of two additions. When a group of data item d_a arrives at stream a, plus sign of d_a obtains the contents of d_b from w_b and performs addition. The result of addition is then recorded in window w_{temp}. Now the minus sign of d_a obtains the result of ($y + z$) from w_{temp} and performs subtraction. The final result of the expression is then recorded in window w_{out}. The same process is repeated when groups of data items d_b, d_y and d_z arrive at stream b, y and z respectively.

In a data stream processing network, if processing of groups of data items is performed in a simultaneous manner then there are circumstances where processing needs to be synchronized for obtaining correct results. In the next chapter, we present examples that necessitate synchronization for processing groups of data items in a data stream processing network.

3.3.2 Translation

The following method is used to translate an expression with two inputs and one output into a data stream processing network. Let e be an expression and T_e be a syntax tree for that expression with nodes representing operators and leaf nodes representing operands for the operators.

Each operand i.e. leaf node of T_e is considered as a stream s_i where $s_i \in \{s_1, \ldots, s_n\}$. Also consider $A_1, \ldots, A_n$ be a sequence of operations for every stream s_i from parent node to root

node of T_e. The input data item of stream s_i is first processed by A_1 operation from the sequence of operations and then rest of the operations in the sequence process the output of preceding operation.

For every stream s_i, do the following:

1. Add a writer and a window for the input data item d_i appended to the stream s_i.

2. Provide operation A_1 with (d_i, ws_{i+1}) where d_i is the first argument and ws_{i+1} is the second argument.

3. Add a writer for the out put of every operation A until root node operation of A_n and an intermediate window (w_{temp}) for arguments that contribute towards that window.

4. For all other A's i.e. A_2 till root node A_n in the sequence provide A with ($\{w_{temp}\}$, ws_{i+n}) where w_{temp} the previous output and ws_{i+n} the other leaf node as the second argument.

5. Add a writer and an output window (w_{out}) for every data stream s_i after root node operation A_n.

6. For step 1 and 2 place an edge from the writer towards the window at the input stream and again an edge from the writer towards the operatoin A_1. Place edges from A_1, … , A_n for every stream s_i showing data flow among operations. For step 3 place an edge from the writer of every s_i to the intermediate window w_{temp}. For any operation A_1, … , A_n place an edge from window w to the operation if that operation reads from any window. For step 5 place an edge from the writer to the window (w_{out}).

Chapter 4

Synchronization Problems

This chapter identifies the circumstances that necessitate synchronization in data stream processing networks. The chapter starts with an overview of sequential and simultaneous processing of data items in a data stream processing network. The next section presents motivating examples. The chapter also shows how the processing of data items in a data stream processing network can be expressed in the terms of database transactions.

4.1 Data Processing Techniques

In the sequential technique, groups of data items are processed one at a time. A group of data item is processed only when its predecessor group completes processing. If groups of data items are processed one by one - i.e. sequentially then processing rate of groups of data items will be extremely slow. As there is no switching of processing between operations of different groups of data items, this is why serial processing does not have any problems regarding synchronization.

On the other hand, the simultaneous technique involves processing all groups of data items at the same time. The processing of a group of data items starts as soon as it arrives in a stream. If simultaneous processing involves a single processor then the processor shares time between operations of different groups of data items. It switches processing from one

operation to the other of different groups of data items. In that case there are places where processing needs to be synchronized in order to get correct results.

Processing long sequences of data items in a sequential manner is not feasible as processing rate becomes extremely slow. Instead it should be done in simultaneous way which makes processing more effective by processing groups of data items at the same time. For the rest of the thesis we will focus on simultaneous processing.

4.2 Motivating Examples

In a data stream processing network, if processing of groups of data items is performed in a simultaneous manner then there are circumstances that require synchronization for obtaining correct results. In the following examples we present situations that occur if synchronization is not considered.

4.2.1 Example 1

Consider two groups of data items d_r and d_s appended in more or less the same period of time to the streams *r* and *s*. Firstly the two groups are recorded in the respective windows w_r and w_s. Then a join operation on d_r and w_s is executed at the same time as the join operation on d_s and w_r. If d_r and d_s were already recorded in windows w_r and w_s then this leads towards computing the results of join d_r and d_s twice as shown in figure 4.1.

The same possibility may happen twice in figure 4.2 which contains four groups of data items including d_a, d_b, d_y and d_z. Groups of data items d_a and d_b are appended to streams *a* and *b* whereas groups of data items d_y and d_z are appended to streams *y* and *z* in more or

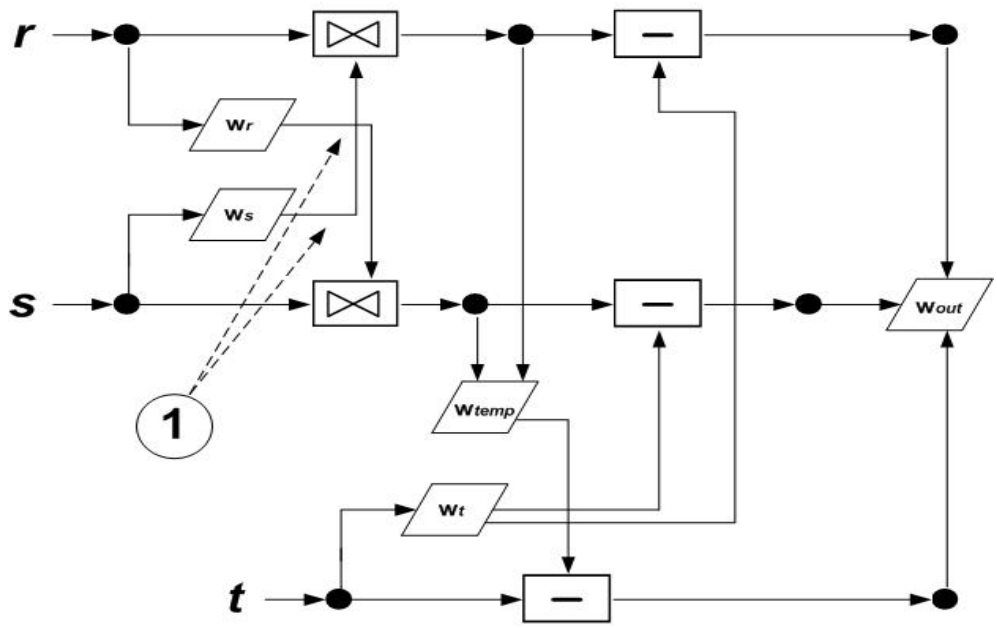


Figure 4.1: Data stream processing network for a relational algebra expression of example 1.

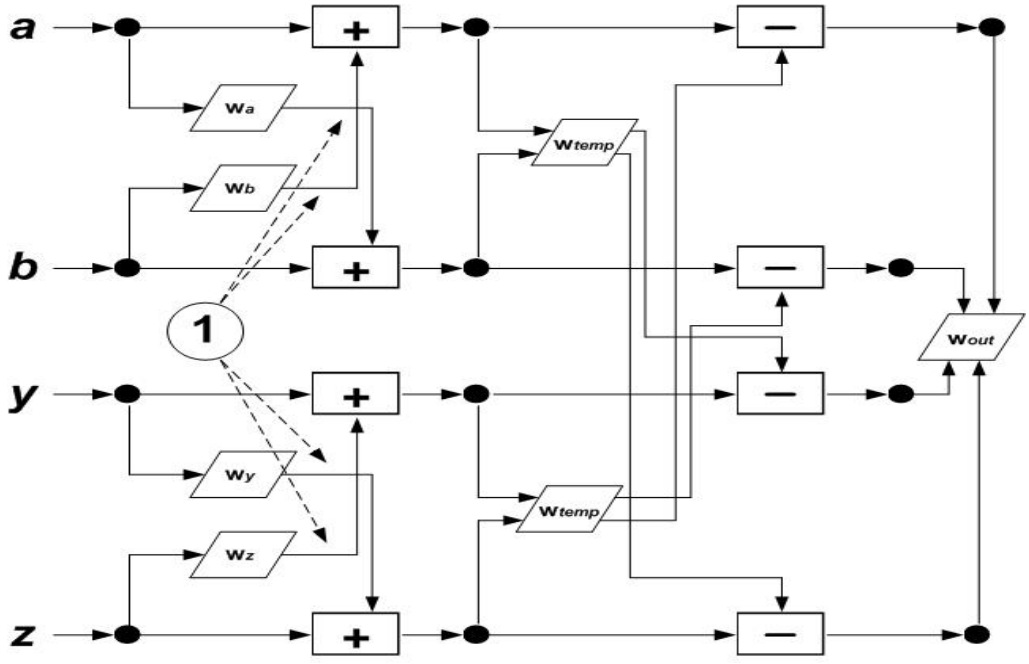


Figure 4.2: Data stream processing network for an arithmetic expression of example 1.

less the same period of time. All groups of data items are recorded to their respective windows w_a, w_b, w_y and w_z.

The first posibility may occur when join operation on d_a and w_b is executed at the same time as the join operation on d_b and w_a assuming d_a and d_b were already recorded in windows w_a and w_b. The second posibility may take place when join operation on d_y and w_z is executed at the same time as the join operation on d_z and w_y assuming d_y and d_z were already recorded in windows w_y and w_z.

4.2.2 Example 2

Consider a case where two groups of data items d_r and d_s are recorded in window w_{temp} after going through a sequence of operations. Now the order in which the two groups are recorded may not be in accordance with the stream arrival. The same possibility may also occur at w_{out}, where the final result from each stream is recorded as shown in figure 4.3.

In figure 4.4, both temporary windows w_{temp} and the window w_{out} have the same risk where results may not be recorded in accordance with the stream arrival.

4.2.3 Example 3

Consider two groups of data items d_r and d_s are appended to streams r and s and start executing. As execution starts, one more group d_t arrives and is appended to stream t. According to the expression $[(r \bowtie s) - t]$, a join operation is performed on d_r and d_s, whereas d_t requires reading the results of d_r and d_s after the join is performed. The result of the join gets recorded in w_{temp}, which is then read by d_t. Now there is a possibility that the

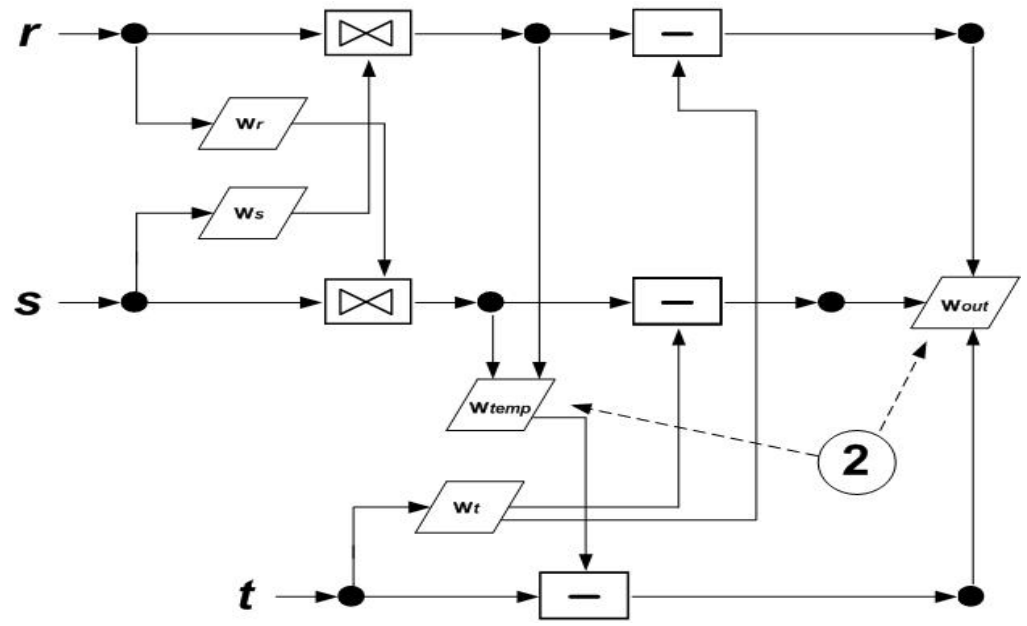


Figure 4.3: Data stream processing network for a relational algebra expression of example 2.

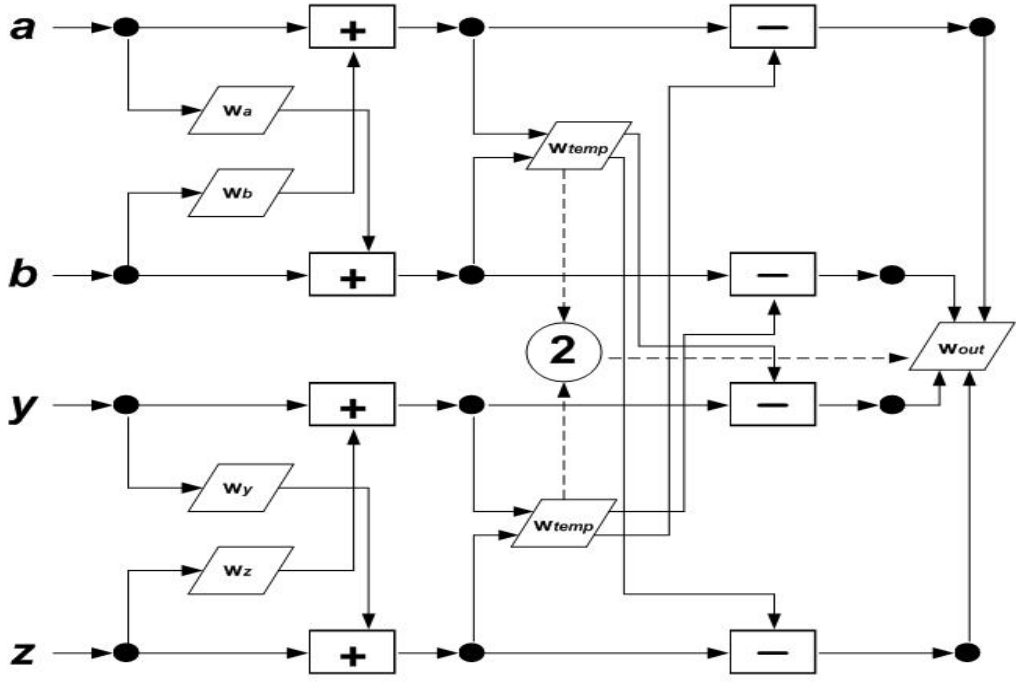


Figure 4.4: Data stream processing network for an arithmetic expression of example 2.

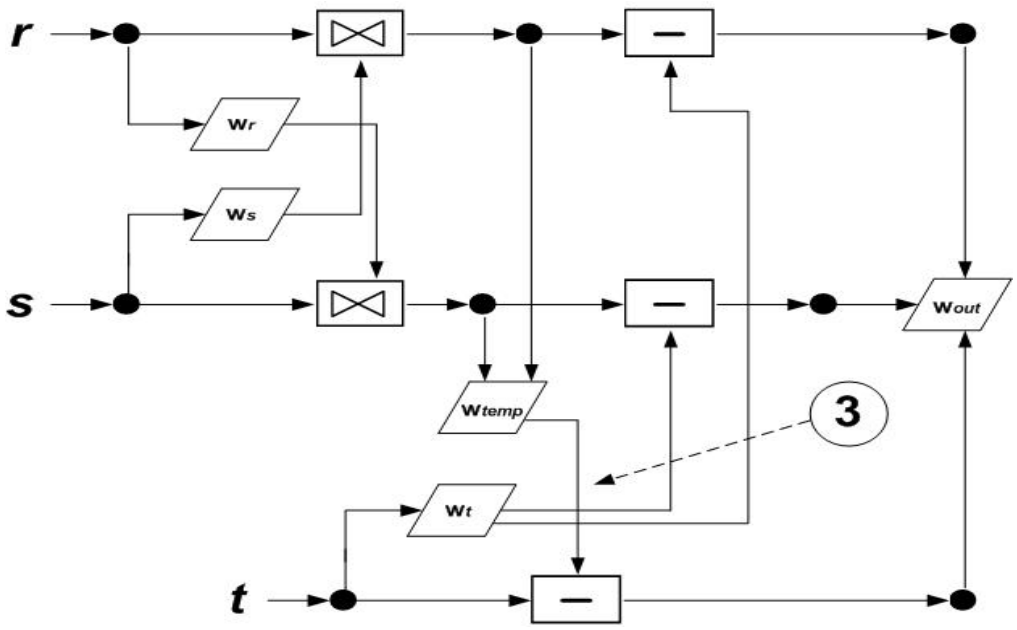


Figure 4.5: Data stream processing network for a relational algebra expression of example 3.

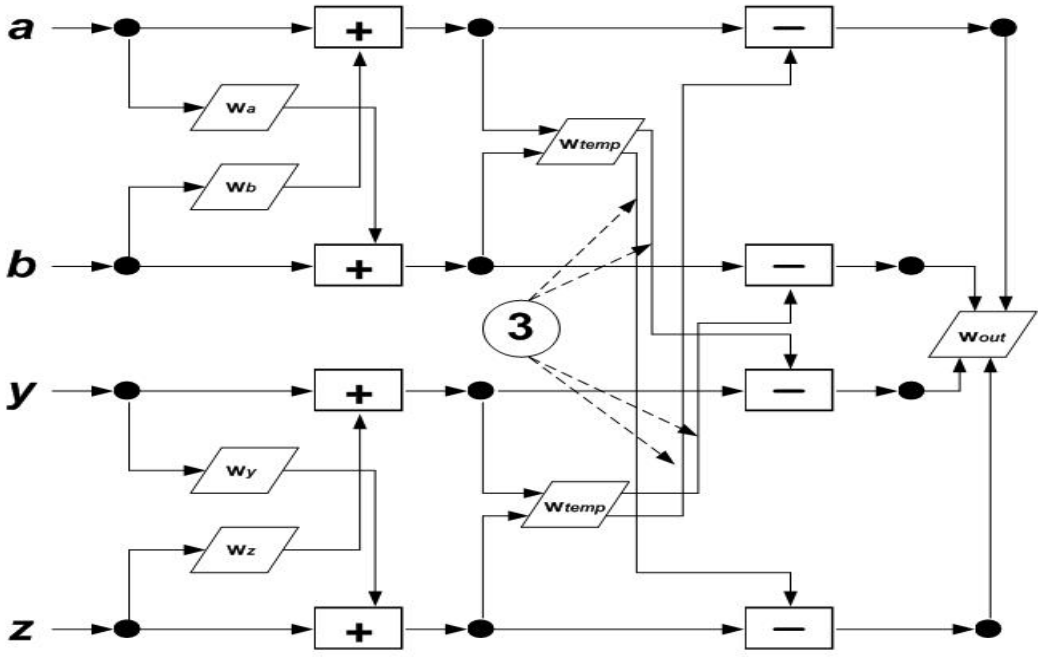


Figure 4.6: Data stream processing network for an arithmetic expression of example 3.

read operation of d_t may read w_{temp} before the result of the join is recorded, as shown in figure 4.5.

In figure 4.6, both temporary windows contain the same possibility. The temporary window where result of join between d_a and d_b is to be recorded - may be read by either d_y or d_z before the result of the join is recorded. The same can happen with the temporary window containing result of join between d_y and d_z that it may be read by either d_a or d_b before the result of join gets recorded.

4.3 Transactional Interpretation of Data Stream Processing

A database transaction is a collection of operations for performing a specific task (Bernstein et al. 1987)**.** If a sequence of input data items is processed concurrently then synchronization of data streams is similar to the synchronization of database transactions.

In our case, a body of transaction is sequence of reads and writes performed at different windows as shown in figure 4.7 and 4.8. A transaction starts when a group of data items gets appended to a stream and is recorded to a window on the input data stream. The group then goes through a sequence of reads and writes and finally gets recorded to the window with the final results. This becomes an end of a transaction. In some cases when an operation produces an empty group of data items then that empty group is provided as an input to the rest of the operations in that sequence. When the empty group of data items is recorded to the window containing final results, then in that case it becomes an end of the transaction. This transformation of execution of groups of data items to database transactions is called as transactional interpretation of data stream processing.

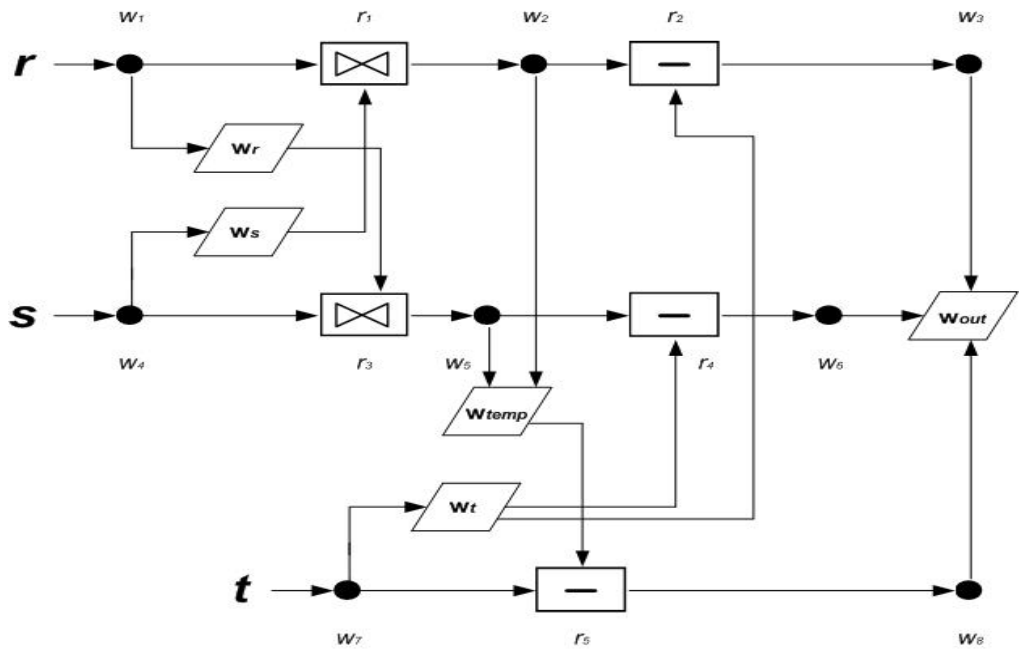


Figure 4.7: Data stream processing network for a relational algebra expression with sequences of transactional operations.

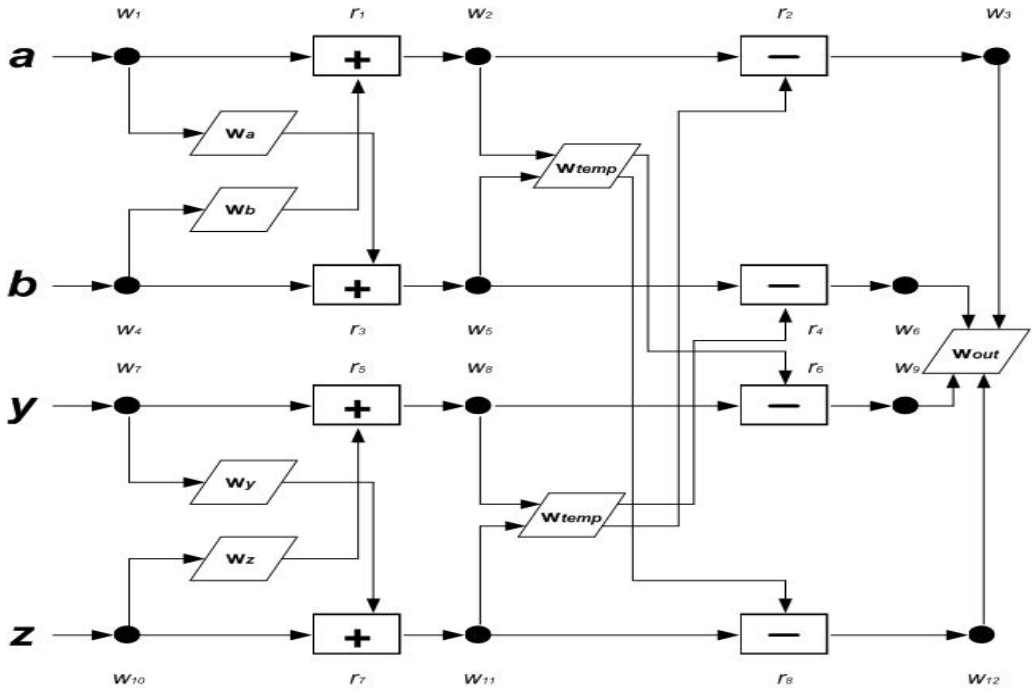


Figure 4.8: Data stream processing network for an arithmetic expression with sequences of transactional operations.

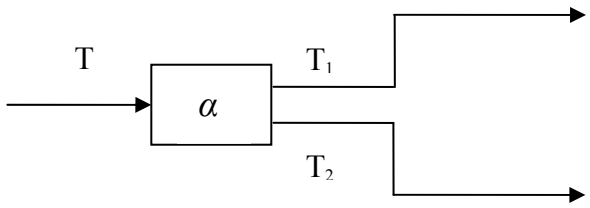


Figure 4.9: A situation where a transaction splits into two sub transactions.

For example in figure 4.1, if a group of data items d_r arrives, gets appended to stream *r* and is recorded into the window over the input queue then that's where a transaction starts. According to the expression $[(r \bowtie s) - t]$**,** a join is performed on d_r and w_s, but before that d_r requires reading the contents of w_s. After a join is performed, d_r records the results into the temporary window and so on until d_r comes to the resultant window w_{out}. The moment d_r records the final result in w_{out} is where a transaction comes to an end. All the read and write operations on windows triggered by processing of d_r constitute the body of transaction. This way processing of a group of data items in a stream is interpreted as a transaction.

4.3.1 Merge and Split

In terms of transactions, merge is a set of two individual transactions T_i and T_j. The two transactions perform their operations individually until they arrive at some point *p*. At point *p* these two transactions start performing their operations as T_iT_j. The merging is performed according to the order in which the two transactions arrived at point *p*.

Split case on the other side is of a single transaction which at some point splits into two sub transactions. Consider the transaction T of figure 4.9 with a sequence of read and write

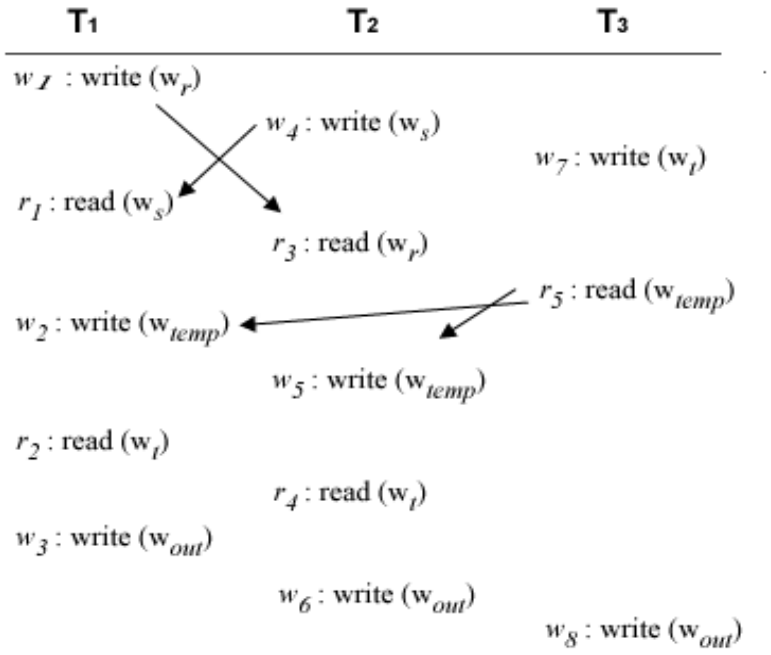


Figure 4.10: Schedule for data stream processing network of figure 4.4.

operations. The transaction splits into two sub transactions T_1 and T_2 after some processing. Each split transaction is given a special number to make them distinct from each other. The sub transactions run sequentially in some order and perform their operations.

4.4 Revised Motivating Examples

On the basis of information on how we get these transactions out of our network, we would now take the motivating examples from section 4.2 and explain them in terms of transactions. Consider each stream in figure 4.7 as a transaction; if all these transactions run simultaneously on a single processor then we will encounter the conflicts presented in the schedule of figure 4.10.

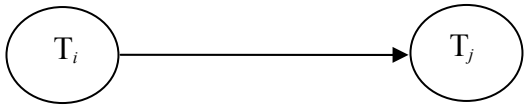


Figure 4.11: Serialization graph for T_i and T_j, where T_i is accessed by T_j.

A schedule is used for showing the order in which transactions are processed. The schedule of figure 4.10 consists of three transactions T_1, T_2 and T_3 for streams *r*, *s* and *t* respectively. As defined earlier, each transaction is a sequence of reads and writes where T_1 consists of operations w_1, r_1, w_2, r_2, and w_3, T_2 consists of operations w_4, r_3, w_5, r_4, and w_6 and T_3 consists of operations w_7, r_5 and w_8.

A concurrent execution is considered to be correct if its schedule is serializable to a serial schedule i.e. running all transactions simultaneously will produce the same results as if these transactions would have been executed serially in some order. The serializability in a schedule can be evaluated by using a graph known as a serialization graph. In a serialization graph, each transaction of the schedule is represented as a node. An edge is created between two nodes when a data item is accessed by two different transactions, at least one of which is in write mode. The edge is directed towards the node representing the transaction which accessed the data item.

Consider a schedule with two transactions T_i and T_j, where T_i is accessed by T_j and at least one transaction in write mode. In the serialization graph an edge will be created from the node T_i towards node T_j as shown in figure 4.11.

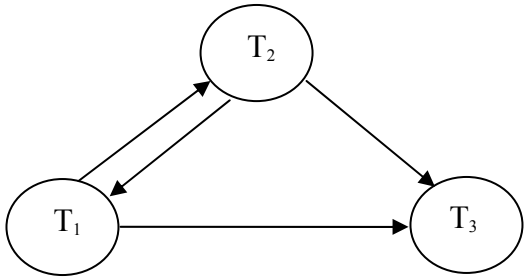


Figure 4.12: Serialization graph for schedule of figure 4.10 contains a cycle.

Now if at some point T_j is accessed by T_i then an edge is placed from node of T_j towards the node of T_i. This results in creation of a cycle between the two nodes. If the serialization graph of a schedule contains a cycle then execution is not considered as serializable. More details on serializability and its related issues can be found in Bernstein et al. (1987).

The serialization graph for the schedule of figure 4.10 is shown in figure 4.12. The first edge gets created when T_2 reads contents of T_1. When T_1 reads T_2 then we get a cycle in the serialization graph. We get more edges when T_3 is accessed by T_1 and T_2.

Chapter 5

Synchronization Strategy and Correctness

This chapter presents a set of rules to synchronize processing of data items in a data stream processing network. The chapter starts with the description of a collector and a regulator. The next section defines a set of synchronization rules. A section applies the synchronization rules to motivating examples of chapter 4. The chapter concludes with a proof of correctness.

5.1 Synchronization Assistants

In order to synchronize execution of groups of data items in a data stream network, the concept of two assistants is introduced. These assistants comprise of a container called collector and a process called regulator.

5.1.1 Collector

A collector is a special purpose container which collects timestamps of data items. It is attached to windows and merge points. Initially a collector contains no timestamp. When a group of data items arrive and is allocated a timestamp, then that timestamp is also placed into collectors attached to windows. This includes windows with intermediate and final results where a group of data items is to be recorded. The timestamp is also placed into collectors attached to merge points from which that group of data items goes through. A

collector attached to a window or merge point keeps a timestamp of a group of data items until the output of that group is not recorded into that window or the group doesn't go through that merge point.

5.1.2 Regulator

A regulator is a process that performs checks at temporary windows and at merge points. A check is performed before an operation reads a temporary window or when a group of data items reaches a merge point. The purpose of performing a check on a window is to make sure that a window is only read by an operation of a group of data items when all previously arrived groups with smaller timestamps have recorded their contents into that window. On the other side, a check is performed at a merge point to allow a group to pass through the merge point only when previously arrived groups with smaller timestamps have gone through that merge point.

When a regulator receives a request by an operation to read the contents of a window then it takes the timestamp of the group of data items executing that operation and performs a check on the collector of the window which is to be read. Similar action is taken by the regulator on the collector at the merge point, when a group arrives at a merge point. By performing a check, the regulator searches for a smaller timestamp in the collector. A collector contains timestamps of already arrived groups which are still going through some processing. The timestamps in the collector are compared to the timestamp of a group which requires reading the contents of a window or timestamp of a group at the merge point.

At both places - i.e. at a window or a merge point - if evaluation of a check returns true then the regulator assumes that all groups of data items with smaller timestamps have been processed. The regulator then allows a group to read the contents of a window or a group to pass through the merge point. If evaluation of a check returns false then the regulator doesn't allow a group to read the window or pass through the merge point.

The regulator then puts the timestamp of that group in a separate queue named a waiting queue. A group of data items whose timestamp is placed into the waiting queue is resumed for processing by the regulator only when previously arrived groups with smaller timestamps have been processed. In such a case, the regulator takes the timestamp of the group out of the waiting queue and sends a request to the operation to resume its processing.

The regulator also removes the timestamp of groups of data items from the collector. As soon as the contents of a group are recorded to a window, its timestamp is deleted by the regulator from the collector attached to that window. On the other hand, when a group of data items pass through a merge point, its timestamp is also deleted from the collector attached to that merge point.

5.2 Synchronization Rules

The following set of synchronization rules is proposed to synchronize processing of groups of data items:

Rule 1

Upon arrival every group of data items is provided with a unique timestamp. The timestamp is a number assigned in an increasing sequence and remains same throughout the execution.

Rule 2

A group of data items can only "see" other groups of data items with smaller timestamps i.e. all previously arrived groups are visible to a group. For example if an operation reads from a window then it will always read groups with smaller timestamp than timestamp of the group performing the operation.

Rule 3

Every window and merge point is supplied with a collector and a regulator excluding windows at input data streams and windows with only write operations.

Rule 4

A group of data items is not released for processing until its timestamp gets recorded into the collectors attached to windows and merge points. This includes windows where contents of that group are to be recorded and include merge points which that group has to go through.

Rule 5

The following are the checks performed by the regulator when an operation attempts to read a window or a group reaches at a merge point:-

At a temporary window

The following check is performed before execution of a read operation at a window:

if there exists a timestamp in collector < timestamp of group reading window

then put timestamp of the group in a waiting queue

else execute read operation

At a merge point:

When a group of data items arrives at a merge point, the following check is performed:-

if there exists a timestamp in collector < timestamp of group at merge point

then put timestamp of the group in a waiting queue

else allow group to pass through merge point

5.3 Solution

We apply the synchronization rules of section 5.2 to the motivating examples of chapter 4:

5.3.1 Solution 1

The situation described in the example of section 4.2.1, where results were computed twice, is eliminated because of rule 2. One of the groups from d_r and d_s that arrived earlier cannot

see the other group due to a higher timestamp. Because of this a join is performed on windows with different contents and no computation is performed twice.

5.3.2 Solution 2

In the example of section 4.2.2, the task was to arrange the order of the results recorded in intermediate or final windows according to stream arrival. This would be quite simple if the arrival time of the group of data items is known. This is already known because we use timestamps.

For the situation described in section 4.2.2, results recorded in w_{temp} and w_{out} are not according to stream arrival but can be easily reordered according to timestamps.

5.3.3 Solution 3

In order to solve the problem of section 4.2.3, the regulator refers to the appropriate collector to perform the check. If in the collector, the regulator finds smaller timestamps then this indicates that there are still groups of data items being processed and the read operation cannot execute at this moment. The regulator puts the group performing the read operation in a waiting queue and resumes processing only when all groups with smaller timestamp are recorded into the window.

For the example in section 4.2.3, the regulator checks for smaller timestamps in the collectors of both d_r and d_s. If the regulator finds smaller timestamps then this means that results have not been recorded. The regulator puts d_t in a waiting queue until the processing of both d_r and d_s gets completed and their values are recorded into the temporary window w_{temp}.

5.3.4 Merge and Split

In section 4.3.1 of chapter 4, the merging was performed according to the order in which the two transactions arrived at a merge point. This instead should always be done according to the arrival order of the two transactions. This leads towards having no problems regarding synchronization.

In case of split of a transaction, we characterize the sub transactions with the main transaction as a parent-child relation. The scope of the child transactions remains within the parent transaction. After splitting the sub transactions run sequentially in some order and results in having no problems regarding synchronization. The order for the sub transactions does not really matter as they will have the same timestamp.

5.4 Correctness

In this section, we prove that applying synchronization rules always results in a correct execution. As in chapter 4 we interpreted processing of data items in a data stream network in terms of database transactions. Now we will use the method to check execution of transactions for their correctness and will apply it to the execution performed through synchronization rules.

For correct execution of transactions, execution should always be conflict serializable i.e. in simultaneously running transactions the conflicting operations must be processed in the same order as they would have been processed in a serial execution. We will prove that set of synchronization rules will produce conflict serializable execution.

The synchronization rules proposed in section 5.2 are for synchronizing a data stream network, whereas the method we apply to check the correctness of the rules belongs to database transactions. In order to prove correctness for the rules, we modify the rules in accordance to database transactions. We then apply the synchronization rules to database transactions to witness the effect made on execution of transactions.

5.4.1 Transactional Interpretation of Rules

The synchronization rules from section 5.2 are translated in terms of transactions in order to synchronize processing of database transactions. There have been certain changes made to the rules in order to apply them on transactions. For instance, the rules made for synchronizing a data stream network had a concept of a window, used for storage of data items. In the rules for synchronizing processing of database transactions we use data containers for storage purpose. The concept of merge point is deleted as it is not valid for database transactions. Regulators and collectors will be placed only at data containers. Checks by regulators will only be performed at collectors attached to data containers. The following are the modified set of synchronization rules:

Rule 1

Every transaction is provided with a unique timestamp. The timestamp of a transaction remains the same throughout execution and is assigned in an increasing sequence.

Rule 2

A transaction can only “see” data items of other transactions with smaller timestamps i.e. data items of all the previously arrived transactions are visible to a transaction.

Rule 3

Every data container will be supplied with a collector and a regulator excluding data containers with only write operations.

Rule 4

A transaction is not released for processing until its timestamp gets recorded into the collectors attached to data containers. This includes data containers where that transaction is to be recorded.

Rule 5

The following check is performed before execution of a read operation at a data container:

if there exists a timestamp in collector < timestamp of transaction reading container

then put timestamp of transaction in a waiting queue

else execute read operation

5.4.2 Theorem

The synchronization rules listed in section 5.4.1 always produces conflict serializable execution.

Proof

As mentioned in section 4.4 of chapter 4, if the serialization graph of a schedule contains a cycle then execution is not considered as serializable. In order to prove the above theorem

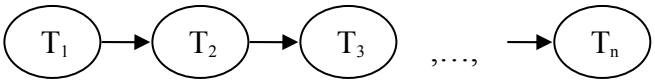
we have to show that there will be no cycles created in the serialization graph when the synchronization rules are applied to execution of a data stream processing network.

Suppose T is a sequence of transaction which we get through transforming an execution of a group of data items in a data stream processing network

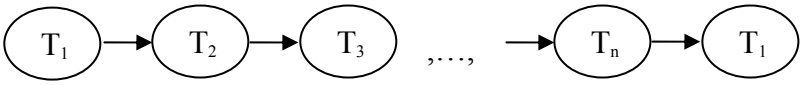
i.e. $T = T_1, T_2, T_3, \ldots, T_n$

where the time stamps $t_1 < t_2 < t_3$ and so on

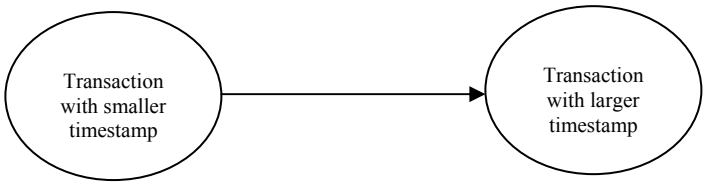
We know that, an edge is formed in a serialization graph when a data item is accessed by two different transactions; at least one of transaction is in write mode (as mentioned in section 4.4 of chapter 4). If T_2 reads the contents of T_1 then an edge will be created from node T_1 pointing towards node T_2 and so on.



Assume that the serialization graph for these transactions contains a cycle. Now for creation of a cycle there will be a moment when an edge goes from T_n to T_1.



Therefore, this contradicts rule 2 from the modified set of rules, which makes every transaction only able to see data items of previously arrived transactions. This rule means that edges can only be created from the node of a transaction with a smaller timestamp to a node with a larger timestamp.



Hence there will be no edges created from a node with a larger timestamp towards a node with a smaller timestamp. This results in having no cycles created in the serialization graph and proves that by using synchronization rules we get conflict serializable execution for groups of data items in a data stream processing network. □

Chapter 6

Conclusion

Synchronization of data stream processing has a major impact on performance of systems where processing of data streams is executed simultaneously. This work contributes towards synchronization of data stream processing as it identified circumstances that necessitated synchronization. The processing of data items within those circumstances was then synchronized by applying a set of rules.

A data stream processing network was defined for illustrating processing of data items. The network was constructed on the basis of a directed graph with a set of nodes and edges. The network provided visualization of how computations were performed in a data stream network. A method was provided for translating an expression into a data stream network and is one of the contributions of this work.

An overview of sequential and simultaneous processing concluded with identifying that simultaneously processed data items required synchronization. Motivating examples were presented for showing consequences if synchronization was not considered. One of the examples presented a situation where a computation was performed twice leading towards wrong results. The other situation presented was the possibility that contents recorded to a window may not be recorded according to stream arrival order. The more complicated

situation presented through the example was where a window was read before results were recorded.

In this thesis we presented the solution as a set of rules based on timestamp ordering. The rules were applied to the circumstances provided in the motivating examples. For synchronizing execution of data items, the concept of a container called collector and a process called regulator was introduced and used within the rules. The regulators and collectors are placed in such a way that if an operation required reading the contents from a window then it was to request the regulator placed at that window for reading the contents. The regulator after receiving a request is used to perform a check for knowing the current status of a collector. If the collector had no smaller timestamps than of the data item performing the operation, then it allowed the operation to read the contents, otherwise the request was rejected. This is how processing was efficiently synchronized and any possibility of wrong results was removed.

One of the main contributions of this work is expressing the processing of data items in a data stream processing network in terms of database transactions. The method used to check execution of transactions for their correctness has been applied to our results, and we proved that using a set of rules will always produce conflict serializable executions.

A summary of my contributions of this work towards “Synchronization of Data Stream Processing” include:

1. A new class of networks called data stream processing networks is defined as a formal model for processing data items.
2. A method is defined for translating an expression into a data stream network.

3. Processing of data items in a data stream processing network is expressed in terms of database transactions.
4. Identification of circumstances that require synchronization.
5. A solution is presented as a set of rules that govern processing of groups of data items.
6. The concept of two synchronization assistants called the collector and the regulator is introduced.
7. Proof of correctness is provided for the strategy used to solve the problems.

References

Abadi, D. et al. (2003): Aurora: A Data Stream Management System. *Proc. ACM SIGMOD International Conference on Management of data*, 663 – 663.

Abadi, D. et al. (2005): The Design of the Borealis Stream Processing Engine. *Proc. Of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA.

Arpaci–Dusseau, R. H. et al. (1999): Cluster I/O with River: Making the fast case common. *In Sixth Workshop on I/O in Parallel and Distributed Systems*, Atlanta, USA, 10 – 22.

Arsu, A. et al. (2002): Characterizing Memory Requirements for Queries Over Continuous Data Streams. *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 221-232.

Avnur, R. and Hellerstein, J. (2000): Eddies: Continuously Adaptive Query Processing. *Proc. ACM SIGMOD International Conference on Management of data*, 261 – 272.

Babcock, B. et al. (2002): Models and Issues in Data Stream Systems. *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1-16.

Babcock, B. et al. (2003): Maintaining Variance and K- Medians Over Data Stream Window. *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 234 – 243.

Babcock, B. et al. (2003): STREAM: The Stanford Stream Data Manager, *IEEE Bulletin of the Technical Committee on Data Engineering*, 19 – 26.

Babu, S. et al. (2004): Adaptive Ordering of Pipelined Stream Filters. *Proc. ACM SIGMOD International Conference on Management of data*, Paris, France.

Babu, S. and Bizarro, P. (2005): Adaptive Query Processing in the Looking Glass. *Proc. Of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA.

Babu, S. and Widom, J. (2001): Continuous Queries Over Data Streams. *Proc. ACM SIGMOD Record*, 30(3): 109 – 120.

Babu, S. and Widom, J. (2004): StreaMon: An Adaptive Engine for Stream Query Processing. *Proc. ACM SIGMOD International Conference on Management of data*, Paris, France.

Bernstein, P. A. et al. (1987): Concurrency control and recovery in database systems. *Serializability Theory*. 25 – 45. Addison – Wesley Publishing Company.

Bose, S. (2004): Data Stream Management for Historical XML Data. *Proc. ACM SIGMOD International Conference on Management of data*, Paris, France.

Carney. et al. (2002): Monitoring Steams a New Class of Data Management Applications. *Proc. International Conference on Very Large Data Bases*, Hong Kong, China.

Chan, J. (2000): NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *Proc. ACM SIGMOD International Conference on Management of data*, 379 – 390.

Chandrasekaran, S. et al. (2003): TelegrpahCQ: Continuous Dataflow Processing. *Proc. ACM SIGMOD International Conference on Management of data*, 665 – 665.

Cherniack, M. et al. (2003): Scalable Distributed Stream Processing. *Proc. Of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA.

Cranor, C. et al. (2003): The Gigascope Stream Database, *IEEE Bulletin of the Technical Committee on Data Engineering*, 27 – 32.

Das, A. et al. (2003): Approximate Join Processing Over Data Streams. *Proc. ACM SIGMOD International Conference on Management of data*, 40 – 51.

Deshpande, A. (2004): An Initial Study of Overheads of Eddies. *Proc. ACM SIGMOD Record*, 33(1).

Deshpande, A. et al. (1998): Caching Multidimensional Queries using Chunks. *Proc. ACM SIGMOD International Conference on Management of data*, 259 – 270.

Getta, J. R. and Vossough, E. (2004): Optimization of Data Stream Processing. *Proc. ACM SIGMOD Record*, 33(3): 34 – 39.

Golab L. and Özsu M. T. (2003): Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2): 5 – 14.

Ives, Z. et al. (1999): An Adaptive Query Execution System for Data Integration. *Proc. ACM SIGMOD International Conference on Management of data*, 299 – 310.

Jain, A. et al (2004): Adaptive Stream Resource Management Using Kalman Filters. *Proc. ACM SIGMOD International Conference on Management of data*, Paris, France.

Lee, Ken C.K. et al. (2004): QUAY: A Data Stream Processing System using Chunking. *Proc. of International Database Engineering and Applications Symposium*.

Lee, Ken C.K. et al. (2002): Semantic Database Broadcast for a Mobile Environment and Adaptive Chunking. *IEEE Trans. on Computers*, 51(10): 1253 – 1268.

Levy, A. (2000): Special Issue on Adaptive Query Processing. *IEEE Bulletin of the Technical Committee on Data Engineering*, 23(2): 2.

Madden, S. et al. (2002): Continuously Adaptive Continuous Queries Over Streams. *Proc. ACM SIGMOD International Conference on Management of data*, Madison, Wisconsin, 49 – 60.

Naughton, J. F. et al. (2001): The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2): 27 – 33.

Rastogi R. (2002): Special Section on Online Analysis and Querying of Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3): 513 – 514.

Sulliovan, M. et al. (1996): Tribeca: A Stream Database Manager for Network Traffic Analysis, *Proc. International Conference on Very Large Data Bases*, 594.

Stonebraker, M. et al. (2003): The Aurora and Medusa Projects, *IEEE Bulletin of the Technical Committee on Data Engineering*, 3 – 10.

Tatbul, N. et al. (2003): Load Shedding in a Data Stream Manager. *Proc. International Conference on Very Large Data Bases*, Berlin, Germany.

Terry, D. B. et al. (1992): Continuous Queries Over Append-Only Databases. *Proc. ACM SIGMOD International Conference on Management of data*, San Diego, USA, 321 – 330.

Tian, F. and DeWitt, D. J. (2003): Tuple Routing Strategies for Distributed Eddies. *Proc. International Conference on Very Large Data Bases*, Berlin, Germany.

Tucker, P. A. et al. (2003): Applying Punctuation Schemes to Queries Over Data Streams. *IEEE Bulletin of the Technical Committee on Data Engineering*, 33 – 40.

Vossough, E. (2004): Processing of Continuous Queries Over Infinite Data Streams. Ph.D. thesis. University of Wollongong.

Xing, Y. et al (2005): Dynamic Load Distribution in the Borealis Stream Processor. *IEEE Proc. of 21st International Conference on Data Engineering*.

Zhu, Y. et al. (2004): Dynamic Plan Migration for Continuous Queries Over Data Streams. *Proc. ACM SIGMOD International Conference on Management of data*, 431 – 442.