UPPSALA
UNIVERSITET

*Digital Comprehensive Summaries of Uppsala Dissertations
from the Faculty of Science and Technology* 836

Scalable Parallelization of Expensive Continuous Queries over Massive Data Streams

ERIK ZEITLER

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2011

ISSN 1651-6214
ISBN 978-91-554-8095-0
urn:nbn:se:uu:diva-152255

Dissertation presented at Uppsala University to be publicly examined in Auditorium Minus, Museum Gustavianum, Akademigatan 3, Uppsala, Tuesday, September 20, 2011 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Zeitler, E. 2011. Scalable Parallelization of Expensive Continuous Queries over Massive Data Streams. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 836. 35 pp. Uppsala. ISBN 978-91-554-8095-0.

Numerous applications in for example science, engineering, and financial analysis increasingly require online analysis over streaming data. These data streams are often of such a high rate that saving them to disk is not desirable or feasible. Therefore, search and analysis must be performed directly over the data in motion. Such on-line search and analysis can be expressed as continuous queries (CQs) that are defined over the streams. The result of a CQ is a stream itself, which is continuously updated as new data appears in the queried stream(s). In many cases, the applications require non-trivial analysis, leading to CQs involving expensive processing. To provide scalability of such expensive CQs over high-volume streams, the execution of the CQs must be parallelized.

In order to investigate different approaches to parallel execution of CQs, a parallel data stream management system called SCSQ was implemented for this Thesis. Data and queries from space physics and traffic management applications are used in the evaluations, as well as synthetic data and the standard data stream benchmark; the Linear Road Benchmark. Declarative *parallelization functions* are introduced into the query language of SCSQ, allowing the user to specify customized parallelization. In particular, declarative stream splitting functions are introduced, which split a stream into parallel sub-streams, over which expensive CQ operators are continuously executed in parallel.

Naïvely implemented, stream splitting becomes a bottleneck if the input streams are of high volume, if the CQ operators are massively parallelized, or if the stream splitting conditions are expensive. To eliminate this bottleneck, different approaches are investigated to automatically generate parallel execution plans for stream splitting functions. This Thesis shows that by parallelizing the stream splitting itself, expensive CQs can be processed at stream rates close to network speed. Furthermore, it is demonstrated how parallelized stream splitting allows orders of magnitude higher stream rates than any previously published results for the Linear Road Benchmark.

Erik Zeitler, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-8095-0
urn:nbn:se:uu:diva-152255 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-152255)

To my grandparents
Anna and Hans Wilhelm
Hannelore and Rudolf

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I E. Zeitler, T. Risch. (2006) Processing high-volume stream queries on a supercomputer. *Proc. ICDE Workshops 2006*, pp 147–151.
I am the primary author of this paper.

II E. Zeitler, T. Risch. (2007) Using stream queries to measure communication performance of a parallel computing environment. *Proc. ICDCS Workshops 2007*, pp 65–74.
I am the primary author of this paper.

III G. Gidófalvi, T. B. Pedersen, T. Risch, E. Zeitler. (2008) Highly scalable trip grouping for large-scale collective transportation systems. *Proc. EDBT 2008*, pp 678–689.
I contributed to 60% of the implementation work and to 30% of the writing.

IV E. Zeitler, T. Risch. (2010) Scalable Splitting of Massive Data Streams. *Proc. DASFAA 2010 part II*, pp 184–198.
I am the primary author of this paper.

V E. Zeitler, T. Risch. (2011) Massive scale-out of expensive continuous queries. Accepted for publication at *VLDB 2011.*
I am the primary author of this paper.

Reprints of the papers were made with permission from the publishers. All papers are reformatted to the one-column format of this book.

Other Related Publications

VI T. Risch, S. Madden, H. Balakrishan, L. Girod, R. Newton, M. Ivanova, E. Zeitler, J. Gehrke, B. Panda, M. Riedewald: Analyzing data streams in scientific applications. In A. Shoshani, D. Rotem (eds.): *Scientific Data Management: Challenges, Existing Technology, and Deployment*. Chapman & Hall/CRC Computational Science 2009, pp 399–429.

Contents

1 Introduction 11

2 Background 15
- 2.1 Data Stream Management Systems 15
- 2.2 Parallel Data Stream Management 16
- 2.3 Distributed Databases 17
- 2.4 Parallel Batch Systems 18
- 2.5 Amos II 19

3 Overview of contributions 21

4 Future Work 26

5 Summary in Swedish 28
- 5.1 Stående frågor över dataströmmar 28
- 5.2 Forskningsfrågor 29
- 5.3 Sammanfattning av studierna 31

6 Acknowledgements 33

7 Bibliography 35

Abbreviations and Symbols

Amos	Active Mediator Object System
CPU	Central Processing Unit
DBMS	Database Management System
DSMS	Data Stream Management System
b	Broadcast percentage
bfn	Broadcast function
C	(CPU) cost
CQ	Continuous query
cc	Consume cost (Paper IV)
ce	Emit cost (Paper IV)
cm	Merge cost (Paper V)
cp (Paper IV)	Process cost
cp (Paper V)	Poll cost
cr	Read cost (Paper V)
cs	Split cost (Paper IV)
E	Emit capacity (Paper IV)
f_l	Fanout at tree level l (Paper IV)
Φ	Stream rate (Paper IV – V)
Φ_D	Desired stream rate (Paper V)
Φo_i	Rate of output stream *i* (Paper IV)
$\Phi o^{(l)}$	Total output stream rate at tree level l (Paper IV)
$\Phi_{PARASPLIT}$	Maximum stream rate of *parasplit* (Paper V)
Φ_{PQ}	Maximum stream rate of *PQ* (Paper V)
Φ_{PR}	Maximum stream rate of *PR* (Paper V)
Φ_{PS}	Maximum stream rate of *PS* (Paper V)
$\Phi_{PS}^{(1)}$	Maximum stream rate of *PS* with $q = 1$ (Paper V)
Gbps	Gigabit per second
GPU	Graphics Processing Unit
η	Efficiency (Paper V)
l	Splitstream tree level (Paper IV)
λ_l	Cumulative fanout at tree level l (Paper IV)
L	Number of expressways in the LRB
LR()	LRB stream function implementation (Paper IV)
LRB	Linear Road Benchmark
Mbps	Megabit per second
MPI	Message Passing Interface

MRT	maximum response time
n	Parallelism (Paper III, section 4)
$O(\cdot)$	Complexity is order of $\cdot$
p	PS parallelism (Paper V)
PQ	Query processor in *parasplit* (Paper V)
PR	Window router in *parasplit* (Paper V)
PS	Window splitter in *parasplit* (Paper V)
pset	Processing set (in BlueGene)
q	PQ parallelism (Paper V)
r	Routing percentage
r_l	Routing percentage at tree level l (Paper IV)
rfn	Routing function
RP	Running Process (Paper I)
S_i	Stream i (Paper IV)
So_j	Output stream j (Paper IV)
$So^{(l)}_j$	Output stream j at tree level l (Paper IV)
SCSQ	Super Computer Stream Query processor
SCSQL	Super Computer Stream Query Language
scsq-lr	SCSQ LRB implementation (Paper IV – V)
scsq-plr	Parallelized SCSQ LRB implementation (Paper IV – V)
SP	Stream process
TCP	Transmission Control Protocol
TG	Trip Grouping algorithm (Paper III)
u	Number of input streams (Paper V)
w	Width of parallelization (Paper IV)
W	Physical window size (Paper V)

1 Introduction

On-line decision-making over streaming data requires processing of *continuous queries* (CQs). CQs are used in applications such as science, engineering, and financial analysis. Unlike conventional database queries that are defined over tables, CQs are defined over live streams of values. A conventional database query executes once and returns a table of tuples reflecting the current state of the tables. Each row in a database table is called a tuple. Analogously, an item in a data stream is also called a tuple. Unlike a conventional database query that results in a table, the result of a continuous query is a stream. This result stream is updated as new data appears in the input stream(s). The data streams are often of such a high rate that saving them to disk is not desirable or feasible. Furthermore, results of CQs have to be delivered as soon as possible, putting requirements on the response time. In many cases, the applications require non-trivial analysis, leading to CQs involving expensive processing.

When new tuples arrive in the input stream, the CQ is executed over these tuples. If the CQ is expensive, result tuples will not be delivered immediately. Depending on the cost of the CQ, delays are incurred until result tuples are delivered. If the response time is larger than the rate of the input stream tuples, the delays accumulate, effectively preventing the system from keeping up with the input stream rate. A classic method for keeping up with the input stream rate is *load shedding*, i.e. dropping the tuples of the input stream that cannot be processed in time [38]. However, if data loss is not tolerated, load shedding is not an option, and the execution of queries becomes a scalability problem. One approach to provide scalability of CQs with expensive operations over high-volume streams is to parallelize the execution of the CQs. Input streams must be split into parallel sub-streams, over which expensive query operators are continuously executed.

The problem of parallelizing CQ execution with expensive operations is addressed in this Thesis, which consists of five papers. The following overall research questions are studied. These research questions are established from the originally formulated research questions stated in *Paper I*.

1. How can scalability of continuous query execution involving expensive computations be ensured for large stream data volumes?
2. How should user-defined computations, and models to distribute these, be included without compromising the scalability?

3. How does the hardware environment influence the system architecture and its algorithms? For example, how can the communication subsystems be utilized optimally?

To answer the above research questions, we implemented a parallel Data Stream Management System (DSMS) prototype, called SCSQ (Super Computer Stream Query processor). A DSMS is a general software system that processes CQs over data streams. In SCSQ, CQs are specified in a query language that includes types and operators for streams and vectors. Vector processing operators enable queries to contain numerical computations over the input data streams. Composite types are allowed, which enables useful constructs such as vectors of streams. Furthermore, the query language is extended with *stream processes* (SPs) and *parallelization functions*, which allow the user to specify customized parallelization and distribution of queries. SCSQ has been implemented to execute in a variety of hardware environments, including desktop PCs, Linux clusters, and IBM BlueGene.

SCSQ was evaluated using data and queries from the following applications:

- Digital telescopes of the kind that has been developed in the LOFAR [31] and Lois projects [32] (*Paper II* and *Paper VI*). Thousands of receivers spread over vast land areas digitize radio waves from outer space into data streams. Scientists search and analyze physical phenomena in these streams using CQs. The challenge is to execute these CQs over streams of high volume from a large number of receivers.
- Automatic online spatio-temporal trip grouping in metropolitan areas with the purpose to save transportation cost (*Paper III*). The challenge is to continuously discover trip groupings with high savings when the number of requests per second is high.
- The Linear Road Benchmark (LRB) [4] (*Paper IV – V*). The LRB simulates an expressway system with variable tolling, which depends on the current traffic conditions. The system must compute toll rates and discover accidents using continuous queries over position reports that are emitted from the vehicles travelling in the expressway system. All queries must deliver results within the allowed Maximum Response Time (MRT). The challenge is to process as many expressways as possible.

Developing and evaluating SCSQ for these applications also led to the following more specific research questions:

4. If the input stream splitting requires both routing and broadcasting of tuples, how can the stream splitting scale with increasing stream rate?
5. If the input stream splitting itself is expensive, how can the stream splitting be automatically parallelized, with additional resource consumption within reasonable bounds?

Questions 4 and 5 are specializations of questions 1 and 2.

Table 1 shows the relationship between each research problem and the papers. The contributions of the papers are summarized briefly below the table. A more elaborate summary of the contributions can be found in Chapter 3.

Table 1. Relationship between research questions (1 – 5) and papers (I – V).

	1	2	3	4	5
I	×	×	×		
II			×		
III	×	×			
IV	×	×		×	
V	×	×	×	×	×

The main contribution of *Paper I* is the definition of the research questions **one**, **two**, and **three**, and the outline of the first prototype of SCSQ, which was implemented in LOFAR's heterogeneous parallel computing environment featuring an IBM BlueGene super computer and a number of Linux clusters.

Paper II enhances the SCSQ prototype in the heterogeneous parallel computing environment. Multiple hardware systems had to be utilized optimally by SCSQ. We develop primitives for efficient stream communication and parallel stream processing. Scheduling of the parallel stream processes turned out to be important for high stream rate in such an environment. These results provide an answer to research question **three**.

The work in Paper I – II forms the basis for Paper VI, which summarizes the architecture of SCSQ and further discusses how SCSQ utilizes the hardware of a parallel computing environment.

Our implementation of stream communication and query distribution in SCSQ enabled us to study various practical applications of parallel stream processing. In *Paper III*, a system for continuous automatic booking of large-scale car sharing was implemented in SCSQ (Trip Grouping algorithm; TG) in order to save travel costs in metropolitan areas. A parallelization study showed that naïve round-robin splitting of the input data stream decreases the travel cost savings. When splitting the input stream using spatial methods, the savings improved substantially compared to the naïve splitting. This shows that custom splitting of input data streams is important. To facilitate advanced stream splitting, SCSQL is extended with *postfilters* that allow very flexible specifications of whether each individual result tuple should be sent to zero, one or more other stream processes. Paper III provides answers to research questions **one** and **two**.

To propel the development of SCSQ, we made an implementation of the LRB, called *scsq-lr* [41]. In *Paper IV*, different methods are evaluated for parallelizing custom input stream splitting. The overall strategy was to generate a tree of stream processes, where the input stream arrives at the root of the tree, and the parallel sub-streams are available at the leaves. The expensive query operators are continuously executed in parallel over the streams from the leaf nodes. We showed that such tree-shaped stream splitting scales significantly better than a naïve splitting performed in a single stream process. Furthermore, our performance for the LRB (64 expressways) is enhanced by one order of magnitude in comparison to previously published results [17]. Paper IV provides answers to research questions **one**, **two**, and **four**.

The fundamental limitation of tree-shaped data stream splitting is the fact that all tuples must pass the root, in which operators for the custom stream splitting are executed on each tuple in the stream. Furthermore, passing tuples between the SPs in the tree is computationally expensive. The cost of stream splitting and communication turns the root into a bottleneck. To eliminate this bottleneck, we developed a fully parallelized stream splitting method in *Paper V*, where custom stream splitting is performed on parallel sub-streams. Furthermore, to cut the communication cost, we introduced physical windows, effectively amortizing the communication cost over all tuples in the window. We call this parallelized stream splitting approach *parasplit*. We showed that stream splitting – and hence parallel stream processing – could be performed at network bound speeds using *parasplit*. Furthermore, we showed that the computational overhead incurred by executing all the processes in *parasplit* was moderate. Lastly, our performance for the LRB (512 expressways) is enhanced by an additional order of magnitude compared to the results in Paper IV. In summary, Paper V provides answers to all research questions.

The next chapter gives an overview of the enabling technologies used to develop SCSQ, and summarizes related work. Chapter 3 elaborates the contributions, and outlines the evolution of SCSQ. Lastly, Chapter 4 provides directions for future work.

2 Background

This chapter discusses Data Stream Management Systems (DSMSs) and technologies that are related to this Thesis, including distributed databases and parallel batch systems. In addition, the chapter introduces the Amos II system, which SCSQ extends.

2.1 Data Stream Management Systems

Figure 1 shows the important building blocks of a DSMS.

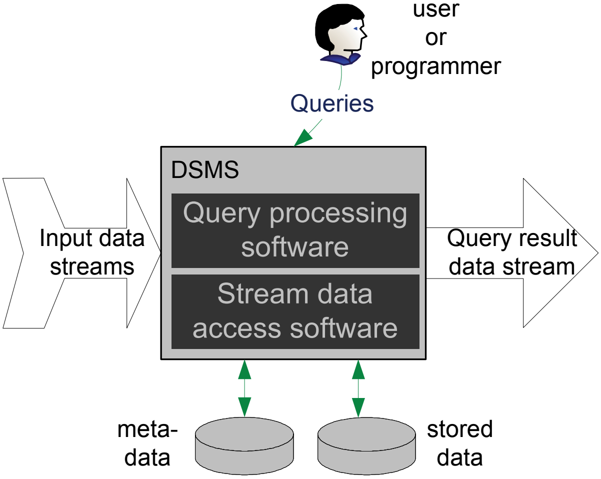


Figure 1. A Data Stream Management System

Like a Database Management System (DBMS), a DSMS compiles and optimizes user queries into query plans. Unlike a DBMS, a DSMS has the capability to process not only data at rest in tables, but also data in motion, illustrated by the input data streams in the figure. Queries that involve streams are called Continuous Queries (CQs). Unlike one-time queries to regular databases, CQs keep delivering results continuously in an output stream, and can continue to do so for an indefinite amount of time. A CQ is terminated either explicitly by the user or by a stop condition in the query. When optimizing one-time queries, the query optimizer may use meta-data and statistics on the tables. In the same fashion, a CQ optimizer may use meta-data and statistics on the data streams. An executing CQ plan continuously reads input data streams and may access stored data. A lot of research effort has been put into semantics and languages for CQs, as well as processing, optimization, and execution of CQs [22]. Many of these research efforts are made by building and extending DSMS prototypes [1] [11] [14] [29] [33].

When executing an expensive CQ over streams of high rate, it is important that the CQ keeps up with the rate of the input stream(s). One strategy to keep up with the stream rate in overload situations is load shedding [38] [15]. This is not an option if data loss is not tolerated. If the input stream is bursty, it may be feasible to balance the load over time by writing some tuples to disk during overload, and process them later during quieter periods [30]. This strategy is called *state spill*. If the input stream rate is constantly high and if the application needs the DSMS to respond in time, state spill is not an option. In this case, parallelization of the execution is a way to keep up with the input stream rate. How this is done is explored in this Thesis.

2.2 Parallel Data Stream Management

Two main strategies for parallelization of continuous queries can be identified: Partitioning the query plan (operator parallelism), and partitioning the data (data parallelism). Plan partitioning involves assigning query operators to compute nodes [26]. In adaptive CQ plan partitioning, query plans are partitioned by dynamically migrating operators between processors [8]. A variant of adaptive query plan partitioning is called *Eddies*, which routes tuples to the operator that currently has the smallest load [5] [39]. However, a fundamental problem of CQ plan partitioning is the fact that heavyweight stream operators are bottlenecks. For example, the heaviest stream operator of a partitioned query proved to be a bottleneck in [26]. The goal of data-partitioned parallelization is to eliminate bottlenecks associated with expensive operators by parallelizing those operators and partitioning the data such that each operator processes a portion of the data. Partitioning a data stream requires the input stream to be split into parallel sub-streams over which CQ operators are executed in parallel. DSMS operators for splitting a stream have been discussed in [12], and have been implemented and evaluated in [3] and [9] for moderate numbers of parallel sub-streams. To partition a stream of high volume into a large number of parallel sub-streams, scalable *splitstream* functions are introduced in this Thesis.

A naïve data-partitioning strategy is to route input stream tuples to the query processors in a round-robin fashion. This approach is often sub-optimal, as was shown in [27], where a query-aware input data stream partitioning was proposed and evaluated. However, in [27], the execution and scalability of input stream splitting was not studied. A recent study identifies the problem of scaling up the number of parallel sub-streams when splitting an input stream into parallel sub-streams [3]. Recent work in distributed event based stream processing has also observed the scalability problem of partitioning an event stream into a number of sub-streams using non-trivial stream splitting predicates [9]. This Thesis is set apart from previous work by proposing two approaches for parallelizing the stream splitting itself, namely tree-based parallelization (*exptree* and *maxtree* in Paper IV), and lattice-based parallelization (*parasplit* in Paper V). We show that *parasplit* enables stream processing at network bound rates by massive scale-out of customized routing and broadcasting.

Although automatic parallelization of CQs was shown to be possible for a certain class of aggregation and join queries in [27], it is very difficult to automatically induce a data parallel strategy in general. This is especially difficult if the CQs are not declarative. Therefore, many DSMSs and DBMSs require the user to provide additional information to assist the parallelization of the queries.

Both SPADE [3] and StreamInsight [28] have stream splitting operators that allow routing and broadcasting of streams, which are used when parallelizing the stream processing. The stream programming language WaveScript [34] represents a program by a graph of stream operators that is partitioned into sub-graphs and executed in a distributed environment. GSDM [25] distributes stream computations by generating parallel execution plans with tree-shaped stream splitting, through parameterized code generators. These code generators are called *distribution templates*. The user selects a parallelization strategy by choosing a distribution template. By contrast, SCSQ provides declarative parallelization functions in the query language. Stream splitting is specified using routing and broadcast functions. As parallelization functions are declarative, they are optimizable and automatically parallelizable. This fact is exploited when we parallelize the execution of splitstream into *exptree*, *maxtree*, and *parasplit.*

When transferring stream tuples between compute nodes in a distributed DSMS, the marshalling cost is substantial. This tuple transfer cost is reduced by grouping tuples into windows (also known as signal segments, or *SigSegs*) [21]. Similarly, SCSQ utilizes physical windows, which was shown to be important for maintaining network bound stream rates in Paper V.

2.3 Distributed Databases

In distributed databases, fast and scalable data processing is facilitated by scaling out storage. Fragmentation and replication [35] are key technologies for this scale-out. The purpose of fragmentation is to partition data over distributed storage nodes in a balanced way, whereas replication aims to provide fast access or high availability by storing each tuple in more than one node. The user provides fragmentation and replication conditions as metadata. Analogous to fragmentation and replication conditions of distributed databases, our splitstream functions provide customized routing and broadcasting of stream tuples (Paper IV – V). Unlike distributed databases, the extreme stream rates for DSMSs require scaling out not only the CQs, but also the execution of routing and broadcast functions.

2.4 Parallel Batch Systems

A well-known example of an infrastructure for large-scale parallel data processing is MapReduce [16], which was implemented at Google to support parallel processing on large-scale computational clusters of large numbers of distributed data sets. MapReduce allows a programmer to *map* any function over each data item in a distributed file system, and to compute any *reduce* (aggregate) function over each data item resulting from the mapping. This can be seen as a form of parallelized group-by. By contrast, SCSQ has a general streaming query language, allowing streams to be both split, transformed and queried in a scalable way.

More recently developed systems allow more flexible parallelization schemes than does MapReduce. For example, Dryad [24] provides a procedural language to construct graphs of processes and communication channels. In contrast to Dryad, SCSQ does not require the user to explicitly construct process graphs, since the process graphs of SCSQ are automatically generated by the parallelization functions.

Map-Reduce-Merge [45] provides an SQL-like query language on top of MapReduce, which significantly eases the programming burden on the user. Like Map-Reduce-Merge, SCOPE [10] provides a scripting language and execution environment for analysis of large data sets on large clusters. However, neither Map-Reduce-Merge nor SCOPE allows on-line stream processing.

MapReduce, SCOPE, and Dryad are all batch systems that do not process streams on-line. Also, the Computational Grid [18] is a basic infrastructure for batch processing on distributed clusters. The purpose of a batch system is to provide multiple users with the functionality to process entire data sets at rest within reasonable time, while maximizing total system throughput for all users. As all data files of a batch system are available all the time, a batch system has the freedom to access each data item more than once, while streams typically must be processed in one pass due to their infinite nature. Furthermore, batch computations produce files, while the result of a CQ is a stream. Thus, batch systems do not continuously produce output streams while input data is processed, and the output is normally delayed until all processing is complete. The scheduling of computations in batch systems is also allowed to be delayed to improve total system throughput. By contrast, on-line stream processing using CQs requires the result stream tuples to be delivered just after new data has arrived on the input stream.

Recently, Streaming MapReduce was introduced [13] with pipelining extensions that gave MapReduce the capability to process parallel data streams. Like conventional MapReduce, Streaming MapReduce is based on a procedural programming model not using any general query language. Furthermore, the problem of scalable stream splitting is not handled by Streaming MapReduce.

2.5 Amos II

SCSQ is implemented using the Amos II kernel [36]. Amos II is a functional and extensible main memory DBMS, with a main-memory storage manager, query processor, and a type system. Queries are compiled and optimized using a cost-based optimizer, which translates the queries into procedural execution plans in ObjectLog, which is an object-oriented dialect of Datalog. Queries are optimized using statistical estimates of the cost of executing each generated query execution plan expressed in a query execution algebra. A query interpreter interprets the optimized algebra to produce the result. To minimize memory requirements during the interpretation of queries over large data sets, the execution plans are interpreted in an iterative tuple-by-tuple style, materializing data only when favorable. This approach of minimal materialization lends itself very well to execution of CQs, and is therefore utilized in SCSQ.

SCSQ extends Amos II in the following ways:

- Stream query coordinators start parallel processes dynamically (Paper I – II).
- SPs provide mechanisms for iteration over streams in a distributed environment (Paper I – III).
- Primitives for network stream connections provide an infrastructure for communicating stream processes (Paper II).
- Numerical vectors represented in binary form, and functions operating over these vectors, provide efficient processing of stream tuples (Paper II and Paper IV – V).
- Postfilters extend stream processes by reducing and transforming their output streams (Paper III).
- Query language parallelization functions provide declarative parallelization of CQs (Paper IV – V).

- Physical windowing functions provide network bound data stream rates between stream processes (Paper V).
- Performance tools allow profiling of parallelized query execution (all papers).

3 Overview of contributions

The first SCSQ prototypes were made to execute in a high performance computing environment, containing an IBM BlueGene super computer, and a number of Linux clusters. In such a massively parallel environment, several communication subsystems co-exist and need to be utilized optimally for parallel processing of streams of high rate. Therefore, efficient stream communication primitives are a crucial part of SCSQ. In **Paper II**, SCSQ itself was used to investigate the communication performance of a BlueGene cluster environment. To enable this investigation, the query language of SCSQ, called SCSQL, was extended with Stream Processes (SPs), allowing the user to specify parallelization of queries. Furthermore, query language functions were introduced that allowed the user to specify the location of processes in a heterogeneous and distributed environment. We showed how to use SPs and functions for process location to determine properties of the communication subsystems of a heterogeneous high performance computing environment. The scheduling of SPs was shown to have a significant impact on the communication performance. Thus, careful scheduling of SPs is important to achieve high stream rate in such an environment. These results provides answer to research question **three**.

Using SCSQ, we carried out extensive studies of two applications of parallel stream processing: Trip grouping for large-scale collective transportation systems, and the Linear Road Benchmark (LRB). Both these applications featured expensive CQs, which were executed over input streams of high rate. To keep up with increasing input stream rates, the CQ execution had to be parallelized. In both applications, the input stream was split into a number of parallel sub-streams, each sub-stream having a lower rate than the input stream. CQ operators were executed over each sub-stream. The output streams of the parallel CQ operators were further processed or merged depending on the application.

In **Paper III**, a streamed Trip Grouping algorithm (TG) was devised that enables on-line ride-sharing in a metropolitan area. TG was implemented and executed using SCSQ, and its execution was parallelized. In the parallelization experiments, it became evident that naïvely splitting the input stream in a round-robin fashion leads to sub-optimal trip grouping results. Instead, by splitting the input stream using spatial partitioning methods, the trip grouping quality improved. This demonstrates the usefulness of user-defined splitting of data streams.

Parallel computations were defined as sets of parallel sub-queries, where each sub-query executed on one SP. The output of an SP is sent to one or more other SPs, which are called *subscribers* of that SP. To enable non-trivial stream splitting, SCSQ's stream process function *SP*() was extended with an optional functional argument, called a *postfilter*. The postfilter is expressed in SCSQL, and can be any function that operates on the output stream of its SP. For each output tuple from the SP, the postfilter function is called once per subscriber. Hence, the postfilter can transform and filter the output of an SP to determine whether a tuple should be sent to a subscriber. In the parallelization experiments, one SP was splitting the incoming stream of trip requests using a postfilter.

Figure 2 shows how the SPs communicate when TG is parallelized. The input stream *S* is split by SP_S into *q* parallel streams. Spatial partitioning methods were used in the postfilter function of SP_S. Each stream S_0 … S_{q-1} is processed by an SP running TG. The result streams from all SP_0 … SP_{q-1} are merged into the result stream *R* in SP_U using a union-all. We showed experimentally that splitting the input stream according to spatial partitioning methods was superior to naïve round-robin stream splitting. The results of the parallelization experiments of Paper III provided insight into research questions **one** and **two**.

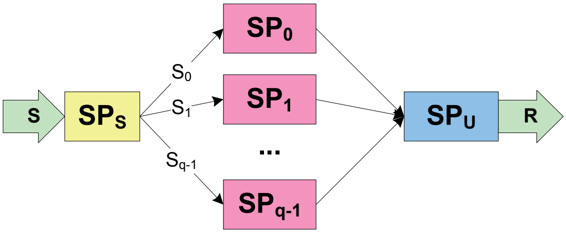


Figure 2. Parallelization of TG using SPs.

For Paper IV – V, we made an implementation of the LRB, called *scsq-lr* [41], and studied how to parallelize that implementation. LRB simulates a traffic system of expressways with variable tolling that depends on the utilization of the roads and the presence of accidents. Vehicles undertake journeys in an expressway system consisting of *L* expressways while emitting position reports. The input stream to the implementation contains such position reports and parameterized queries, whereas the expected output stream of the implementation contains responses to a number of continuous and historical queries, which are specified in the benchmark. The implementation must respond correctly to these queries within the allowed maximum response time (MRT). The number of expressways that an implementation is able to respond to within the MRT is called the *L*-rating of the implementation.

Most of the CPU time of *scsq-lr* was spent computing statistical aggregates for toll calculation. These aggregates are local to each expressway. Thus, the key to efficient parallelization lies in partitioning the input stream into *L* parallel sub-streams, one for each expressway, and executing one instance of *scsq-lr* over each sub-stream. This strategy was employed in *scsq-plr*, as reported in **Paper IV**. When employing this parallelization strategy, a small fraction (0.5%) of the tuples in the input stream requires an aggregate to be computed across all parallel *scsq-lr* nodes. As a consequence, these tuples must be broadcasted to all parallel sub-streams. Each parallel *scsq-lr* emitted a partial result of this aggregate, so these *L* partial results must be aggregated. Thus, the input stream is split such that most tuples are routed to exactly one of the sub-streams, whereas a small fraction of the tuples is broadcasted to all sub-streams.

The cost of splitting the input stream using the postfilter functions developed in Paper III is $O(q)$, where q is the number of output streams. For the LRB, $q=L$. Thus, using postfilters for splitting a stream into *L* parallel streams is too expensive when scaling *L*. To improve the scalability for high parallelism, a new class of functions was introduced, called *parallelization functions*. Parallelization functions are declarative, and can be parallelized automatically. Figure 3 illustrates the three basic parallelization functions: *splitstream*, *mapstream*, and *mergestream*. The function *splitstream* distributes and replicates tuples of the input stream by executing a routing function *rfn* and a broadcast function *bfn*. The functions *rfn* and *bfn* are provided by the user. The function *mapstream* applies a CQ on each stream in a collection of streams, while *mergestream* merges or joins a collection of streams into a single output stream. As *splitstream* turned out to be a bottleneck, we focused on parallelizing the execution of *splitstream* in Paper IV.

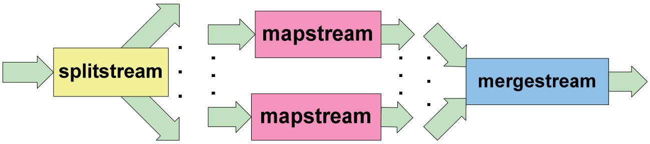


Figure 3. Splitstream, mapstreams, and mergestream.

We made a naïve implementation of *splitstream* called *fsplit*, which executed in a single process. We devised a cost model for *fsplit*, showing that it becomes a bottleneck especially if a large percentage of the tuples are broadcasted. This bottleneck was alleviated by parallelizing the execution of *fsplit* using tree-shaped parallel execution plans. A theoretically optimal execution strategy called *maxtree* was developed based on the cost model for *fsplit*. However, *maxtree* required knowledge of the routing and broadcast percentages, as well as the costs of *rfn* and *bfn*. Therefore, another kind of parallel execution plan called *exptree* was implemented, which did not require knowledge of any of these percentages or costs. Although not theoretically optimal, the performance of *exptree* was shown to be comparable to that of *maxtree*. Lastly, *autosplit* was introduced, which features a simple heuristic that generates an *exptree* or an *fsplit* depending on whether *bfn* is present in the call to *splitstream*. In a final experiment, *autosplit* was used as a splitstream function in a parallel implementation of the LRB. An L-rating of L=64 was achieved, which was an order of magnitude higher than any previously published result.

In summary, the implementation of parallelization functions in Paper IV provides answers to research questions **one** and **two**. Distributing the execution of *splitstream* provides answer to research question **four**.

The fundamental limitation of the tree-shaped execution plans introduced in Paper IV is the fact the input stream must pass the root of the splitstream tree, where *rfn* and *bfn* are executed for each tuple. Therefore, the maximum stream rate of a splitstream tree is sensitive to the cost of executing *rfn* and *bfn*. In particular, it was shown in Paper IV that the maximum stream rate of a tree with the *rfn* and *bfn* used to parallelize the LRB input stream corresponded to 65 expressways. The data rate of 65 expressways is 73 Mbps, which is much less than the bandwidth of a gigabit Ethernet interface. Thus, the CPU cost of executing *rfn* and *bfn* prohibited higher stream rates.

In **Paper V**, we showed how to handle expensive *rfn* and *bfn* by introducing *parasplit*, which is a new way of parallelizing the execution of splitstream. The execution plan generated by *parasplit* had the shape of a lattice instead of a tree. The maximum stream rate of *parasplit* was shown to be superior to that of all splitstream trees. The execution of *rfn* and *bfn* was parallelized into a number of parallel processes, effectively making *parasplit* insensitive to the cost of *rfn* and *bfn*, as well as to the broadcast percentage. When implementing *parasplit*, the cost of marshalling and de-marshalling tuples of the input stream dominated the cost, turning the communication cost into a bottleneck. We introduced physical windows, effectively amortizing the communication cost over all tuples in the window. By setting the window size large enough for the communication system used, the marshalling bottleneck was eliminated.

An execution plan of *parasplit* is shown in Figure 4. First, the *window router PR* reads physical windows containing tuples represented in binary form from the input stream *S*. Each physical window is randomly routed with equal probability to one of the *p* parallel sub-streams S_i, $i = 0 \ldots p - 1$. Second, each window splitter PS_i unpacks the tuples of the physical windows of its sub-stream S_i received from *PR*, and executes *rfn* and *bfn* on each tuple so that each tuple is distributed to zero, one or more continuous *query processors* PQ_j, $j = 0 \ldots q - 1$. Third, each query processor PQ_j merges all received streams T_{ij}, $i = 0 \ldots p - 1$, into a local stream U_j. Expensive CQ operators are then applied in the query processors on each local stream U_j.

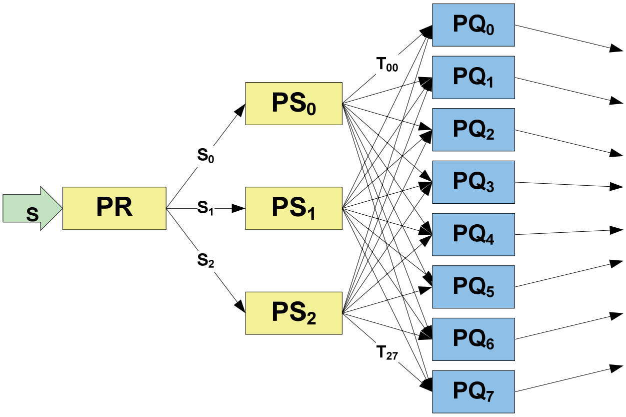


Figure 4. Execution plan of *parasplit*, showing *p*=3 and *q*=8.

The maximum stream rate of *parasplit* was not sensitive to the cost of *rfn* and *bfn*, as the execution of these functions was parallelized. The maximum stream rate of *parasplit* was shown to be network bound instead of CPU bound. Furthermore, we showed that the computational overhead incurred by executing all the processes in *parasplit* was moderate. Thus, Paper V provides answers to all research questions **one**, **two**, **three**, **four**, and **five**.

4 Future Work

When we started to study scalable parallelization of expensive continuous queries over massive data streams, we focused on research questions **one**, **two**, and **three**. In the process of looking for answers to these questions, we found that the scalability of input data stream splitting was crucial, leading us to formulate the additional research questions **four** and **five**. Although this Thesis provides answers to these five research questions, there are several new research questions to study, as outlined below.

Parasplit splits streams at network bound rates, which was experimentally evaluated in a cluster of up to 70 compute nodes with eight cores each, connected by a 1Gbps switched network. Future work includes investigating the behavior of *parasplit* for higher network bandwidths and larger number of compute nodes, to identify unforeseen scalability problems.

The query plan of *parasplit* is optimized, parallelized, and scheduled when the CQ is started. Although this approach was shown to work well in our evaluations, it would be worthwhile to extend it with methods for adaptive parallelization and scheduling of execution over streams after the CQ has been started, as in [29] and [2].

For CQs involving selective predicates, it should be investigated how to push down some selection predicates into *rfn*, effectively saving communication cost by increasing omit percentage *o* in the window splitters of *parasplit*.

Stream join processing has been extensively studied in previous research. However, none of the existing research has investigated stream join processing for large numbers of input streams of high volume. For instance, the studies in [6] and [20] were limited to binary joins, and the experiments of [27] were restricted to eight-way joins (involving four compute nodes). Windowed multi-way join operators were studied for up to six parallel input streams in [42], and in [43], distributed windowed stream join was studied for adaptively partitioned windows. The experimental results in [43] were shown for three-way joins. In sensor networks, merge and join of many streams of moderate rates have been studied [37]. It would be highly interesting to investigate how to facilitate scalable stream join processing for hundreds or even thousands of streams of high volume.

Moreover, we want to extend our energy efficiency studies of parallel stream processing. Paper V estimates the energy efficiency of *parasplit* by comparing the CPU time spent in executing stream splitting predicates to the

CPU time of the parallelized *parasplit*. This efficiency measure shows how much extra work is incurred by parallelizing the stream splitting predicates. The unit of this efficiency measure is a percentage, as CPU seconds are divided by CPU seconds. Future work includes investigating whether GPUs [23] and other hardware acceleration [44] techniques can be utilized to improve energy efficiency of general parallel stream processing.

Various utility measurements that capture the user value versus the execution cost of the DSMS should also be investigated. In the case of the LRB, possible utility measures are expressways per CPU second, expressways per unit electric energy [40], or expressways per ownership and operations cost.

High Availability [7] is another aspect of parallel execution of CQs that has not been studied in this Thesis. The current implementation of SCSQ cannot guarantee operational performance, as there are no mechanisms implemented to compensate for hardware or software slowdowns or unavailability. Hence, methods that provide high availability for highly parallel stream processing systems should be developed. Furthermore, the energy efficiency of such methods should be investigated.

Lastly, the parallelization functions of SCSQ may well provide an execution environment for inference in near real-time, such as data stream mining [19] and event processing [9]. In data stream mining applications, combining high volumes of data at rest with high volumes of data in motion is an important capability. Therefore, future work includes investigating scalable approaches to integrating parallel databases with SCSQ.

5 Summary in Swedish

Den här avhandlingen handlar om skalbar parallellisering av kostsamma stående frågor över massiva dataströmmar. För att förstå vad detta innebär behövs lite bakgrund.

Tillämpningar inom bland annat naturvetenskap, teknik, finansiell analys och datavetenskap ställer ökande krav på att nya data ska analyseras genast så fort de blir tillgängliga. Mätvärden, nyhetsflöden, marknadsinformation och loggfiler innehåller data som ständigt uppdateras. Sådana datakällor kallas *dataströmmar*. När nya data anländer med hög hastighet är det i regel inte önskvärt eller möjligt att lagra dataströmmens innehåll på disk för senare analys, som i vanliga databaser. Istället måste sökning och bearbetning utföras direkt på den levande dataströmmen. Under det senaste decenniet har databasforskningen utvecklat metoder för sökning och bearbetning av sådana dataströmmar. Ansatsen är att bearbetningen ska kunna uttryckas med så kallade *stående frågor*, som förklaras härnäst.

5.1 Stående frågor över dataströmmar

Traditionella databashanterare, såsom Oracle och MySQL, utgörs av mjukvara som lagrar data, vanligen i form av tabeller. Varje rad i en sådan tabell kallas tupel (på engelska *tuple*). En databashanterare har ett frågegränssnitt där användaren formulerar frågor i ett *frågespråk*, vanligen SQL. Frågorna uttrycker *sökningar* och *bearbetningar* av innehållet i dessa tabeller. Svaret på en fråga – som i sig är en tabell – beror av de lagrade tabellernas innehåll. Databashanterarens mjukvara översätter användarens frågor till *frågeplaner*. En frågeplan är ett program som kör de operatorer som behövs för att besvara frågan. Att översätta en fråga till en plan kan göras på ofattbart många olika sätt, och det är viktigt att databassystemet kan upprätta smarta planer så att svarstiderna blir korta – för alla vet hur jobbigt det är att vänta på en dator. Det allmänna problemet att generera effektiva planer för många olika sorters frågor kallas *frågeoptimeringsproblemet* och har studerats under lång tid inom databasforskningen. Även om tabellerna innehåller mycket data, eller om mer data fylls på i tabellerna, är det viktigt att frågorna fortfarande besvaras inom rimlig tid. Effektiv bearbetning av datamängder, även när de ökar i storlek, är ett centralt problem inom datavetenskapen som kallas *skal-*

barhetsproblemet. Frågeoptimering och skalbarhet är centralt även vid sökning och bearbetning av dataströmmar.

Till skillnad från konventionella databasfrågor som är definierade över tabeller, är en *stående fråga* (på engelska *continuous query*) definierad över strömmar av data som ständigt ändras. Ett värde i en dataström kallas tupel, analogt med en rad i en tabell. Medan konventionella databasfrågor returnerar ett resultat som beror av tabellernas innehåll vid tillfället när frågan ställdes, är resultatet av en stående fråga i sig en ström av tupler som uppdateras efterhand som nya tupler anländer i de sökta strömmarna (indataströmmarna). En stående fråga kan köra obegränsat länge.

Många stående frågor innehåller avancerad sökning och bearbetning som kräver mycket datorkraft, d.v.s. är *kostsam* att utföra. Samtidigt kräver tillämpningarna korta svarstider. Därför är frågeoptimeringsproblemet centralt även för dataströmhantering: Målet är att generera en plan av operatorer som levererar resultatströmmen med kortast möjliga svarstid. Svarstiden för en stående fråga definieras som tiden från att data anländer i indataströmmarna tills de eftersökta data levererats i resultatströmmen. Skalbarhetsproblemet är också viktigt: Dataströmhanteraren måste leverera en resultatström med minimalt dröjsmål även om bearbetningen är kostsam och tidskrävande, eller om nya data anländer med hög hastighet i indataströmmen. Dataströmmar där nya data anländer med hög hastighet kallas *massiva*.

5.2 Forskningsfrågor

Ett sätt att snabba upp kostsam sökning och bearbetning av dataströmmar är att *parallellisera* databehandlingen, genom att utnyttja många datorers samlade beräkningskraft samtidigt. I den här avhandlingen, som består av fem studier, har vi undersökt hur stående frågor kan bearbetas parallellt över dataströmmar med hög hastighet. Frågeställningarna för avhandlingen formulerades ursprungligen i vår första studie, *Paper I*. Från denna inledande studie kan följande övergripande frågeställningar kristalliseras:

1. Hur kan stående frågor med kostsamma bearbetningar utföras skalbart över snabba dataströmmar?
2. Hur ska dataströmhanteraren hantera och parallellisera specialiserade databearbetningar på ett skalbart sätt?
3. Hårdvarumiljön utgörs av datorerna som står till systemets förfogande. Hur påverkar hårdvarumiljön systemets uppbyggnad och dess algoritmer? Hur ska t.ex. kommunikationssystemet utnyttjas optimalt?

För att studera forskningsfrågorna har vi utvecklat en prototyp för parallell dataströmhanterig som vi kallar SCSQ (Super Computer Stream Query processor, uttalas 'siss-kju:). I dess frågespråk SCSQL (Super Computer Stream

Query Language) kan stående frågor uttryckas över dataströmmar. Typsystemet i SCSQL innehåller bl.a. strömmar och vektorer samt funktioner över dessa. Funktioner för vektorbearbetning har använts för att utföra beräkningar över strömmarnas innehåll. SCSQL tillåter även sammansatta datatyper, vilket är användbart för att konstruera t.ex. vektorer av strömmar i ett frågespråk som tillhandahåller funktioner över bl.a. strömmar och vektorer. Dessutom innehåller SCSQL *strömprocesser* (SP:er) och *parallelliseringsfunktioner*, där användaren specificerar *icke-procedurellt* hur de stående frågorna ska parallelliseras, d.v.s. utan att behöva ange i detalj hur och var de ska köras. SCSQ fungerar i olika hårdvarumiljöer, t.ex. persondatorer, Linux-kluster och superdatorer såsom IBM BlueGene. I våra studier har SCSQ utvärderats med hjälp av data och frågor från följande tillämpningar:

- Digitala stjärnkikare av den typ som utvecklats i LOFAR- och Lois-projekten (*Paper II* och *Paper VI*). Tusentals radiomottagare spridda över stora landområden fångar upp och digitaliserar radiovågor från yttre rymden och omvandlar dessa till dataströmmar. Forskare eftersöker och analyserar fysikaliska fenomen i dessa strömmar med hjälp av stående frågor. Utmaningen är att fortlöpande utföra kostsamma sökningar och bearbetningar av mycket stora datamängder från ett stort antal mottagare.
- Automatisk bokning av samåkningar i storstadsområden för att minska transportkostnader (*Paper III*). Utmaningen är att fortlöpande planera samåkningar när antalet samtidigt begärda resor är mycket stort.
- Linear Road Benchmark (LRB) (*Paper IV* och *Paper V*). LRB är ett stresstest för dataströmhanteringssystem, som simulerar ett trafiksystem för motorvägar med ett dynamiskt vägtullssystem, vars tull beror på trafikläget. Dataströmhanteringssystemet måste fortlöpande beräkna tull och upptäcka olyckor baserat på stående frågor över positionsdata från samtliga fordon och vägavsnitt. All bearbetning måste dessutom ske inom tillåten svarstid (engelska *Maximum Response Time*, MRT). Utmaningen är att kunna hantera data från så många motorvägar som möjligt.

Inom dessa studier har vi vidareutvecklat SCSQ och fått inblick i följande specifika frågeställningar:

4. Om uppdelningen av indataströmmen kräver att vissa data mångfaldigas, hur kan vi säkerställa skalbarhet i uppdelningen när strömhastigheten ökar?
5. Om uppdelningen av indataströmmen är kostsam, hur kan uppdelningen automatiskt parallelliseras samtidigt som den ökade resursförbrukningen hålls inom rimliga gränser?

Frågorna 4 och 5 är specialiseringar av frågorna 1 och 2. Tabell 1 på sidan 13 visar hur studierna täcker forskningsfrågorna.

5.3 Sammanfattning av studierna

I *Paper I* definieras forskningsfrågorna, som vi redogjorde för i avsnittet ovan. I *Paper II* beskrivs den första prototypen av SCSQ, som kördes i en parallelldatormiljö med en IBM BlueGene superdator och ett antal Linux-kluster där flera hårdvarusystem måste utnyttjas optimalt av dataströmhanteraren. Vi utvecklade primitiver för effektiv strömkommunikation och parallell strömbearbetning (strömprocesser; SP:er). Vi såg att schemaläggningen av strömprocesser i parallelldatormiljön hade avgörande betydelse. Därför måste strömprocesserna placeras noga i en sådan miljö för hög strömhastighet. Dessa resultat gav svar på forskningsfråga **tre**.

Arbetet i *Paper I* och *Paper II* ligger till grund för *Paper VI*, som sammanfattar SCSQs arkitektur och diskuterar hur SCSQ utnyttjar kommunikationssystemet i en parallelldatormiljö.

Med primitiver på plats för strömkommunikation och frågedistribution, använde vi SCSQ för att studera olika praktiska tillämpningar inom parallell strömbearbetning. I *Paper III* implementerades ett system i SCSQ för fortlöpande automatisk planering av stora mängder samåkningar (*trip grouping algorithm*; TG) med syfte att minska resekostnader i storstadsområden. Indataströmmen bestod av begärda resor. I ett första experiment delades denna ström upp genom att de parallellt arbetande processerna turades om att ta emot de begärda resorna. Det visade sig att denna enkla strömuppdelning försämrade besparingarna. Besparingarna blev större när indataströmmen delades upp med spatiala metoder jämfört med när den uppdelades på enklaste sätt. Detta visar att användardefinierad uppdelning av indataströmmar är en viktig teknik. För att möjliggöra avancerad strömuppdelning utökades SCSQL med *postfilter*, som transformerar och filtrerar resultatströmmen från en strömprocess och därigenom avgör hur tupler ska skickas vidare. *Paper III* ger svar på forskningsfrågorna **ett** och **två**.

För att ytterligare driva utvecklingen av SCSQ framåt implementerade vi LRB i SCSQ. Vår implementation kallas *scsq-lr*. I *Paper IV* utvärderades olika metoder att parallellisera användardefinierad uppdelning av dataströmmar. Som övergripande strategi för att dela upp strömmarna genererades träd av parallella strömprocesser, där varje strömprocess utförde en del av uppdelningsarbetet. De parallella kostsamma strömbearbetningarna kördes på delströmmarna från trädets löv. I studien visade vi att en sådan trädformad strömuppdelning skalar betydligt bättre än om uppdelningen utförs av en enda strömprocess. Med denna ansats uppnådde vi en tiopotens högre prestanda för LRB (64 motorvägar) än dittills publicerade resultat. Sammanfattningsvis ger *Paper IV* svar på forskningsfrågorna **ett**, **två** och **fyra**.

Ett problem med trädformad strömuppdelning är att indataströmmen måste passera trädets rot, där den användardefinierade strömuppdelningen utförs på strömmens alla data. Ett annat problem är kommunikationskostnaden: Det krävs mycket datorkraft för att skicka tupler mellan strömprocesserna i trädet. Kostnaderna för strömuppdelning och kommunikation gör att roten blir en flaskhals. För att eliminera denna flaskhals utvecklade vi en fullständigt parallelliserad strömuppdelningsmetod i *Paper V*, där den den användardefinierade strömuppdelningen utförs parallellt på delar av strömmen. Detta resulterar i en komplicerad graf-formad parallell exekveringsplan, som vi kallar *parasplit*. För att minska kommunikationskostnaden klumpade vi samman tuplerna till fysiska fönster (på engelska *physical windows*) i *parasplit*. Vi visade att strömuppdelning med *parasplit* – och därmed parallell strömbearbetning – kan utföras i en hastighet som ligger nära nätverkets maximala hastighet. Vi visade även att den ytterligare datorkraft som måste skjutas till för att köra alla processer i *parasplit* var måttlig. Med *parasplit* uppnådde vi åter en tiopotens högre prestanda för LRB (512 motorvägar) än vårt tidigare resultat i *Paper IV*. På så sätt ger *Paper V* svar på samtliga forskningsfrågor.

Vi började med att ställa forskningsfrågorna **ett**, **två** och **tre**. När vi arbetade med dessa frågor upptäckte vi att det var kritiskt för prestanda att indataströmmen kunde delas upp på ett skalbart sätt. Således uppstod forskningsfrågorna **fyra** och **fem**. I våra fem studier I – V har vi givit några svar på forskningsfrågorna, och vet således nu lite mer om skalbar parallellisering av kostsamma stående frågor över massiva dataströmmar. Emellertid har ytterligare nya forskningsfrågor uppkommit under arbetets gång, som alltjämt återstår att lösa. Dessa nya frågor skisseras i Kapitel 4, *Future Work*.

6 Acknowledgements

First and foremost I would like to thank Professor Tore Risch for supervising me. Thank you for helping me focus the project, and for sharing your knowledge and enthusiasm during our frequent discussions. I appreciate your willingness to assist in software engineering and scientific writing.

Tore is also acknowledged for running Uppsala Database Lab (UDBL) at the Department of Information Technology, Uppsala University. UDBL not only produces research papers and PhDs – UDBL also produces working software systems. The system-oriented approach to database research has made my project very inspiring. Furthermore, I appreciate the social activities of our lab, such as the hiking trips and the dinners at Tore's and Brillan's home. It has been a privilege to be part of UDBL.

I am thankful to present and past lab members – from all over the world – for interesting discussions and for sharing with me the PhD student experience; Kjell Orsborn, Milena Ivanova, Johan Petrini, Ruslan Fomkin, Sabesan, Silvia Stefanova, Győző Gidófalvi, Lars Melander, Minpeng Zhu, Cheng Xu, Andrej Andrejev, Thành Trương Công, Robert Kajić, Mikael Lax, and Sobhan Badiozamany. I am thankful to Győző for the collaboration on scalable trip grouping. Furthermore, I had the pleasure of supervising three master students; Mårten Svensson, Stefan Kegel, and Fredrik Edemar, whose contributions have accelerated my project. Thank you!

Colleagues at the IT department are acknowledged for contributing to the quality of the work environment. The head of the computing science division Lars-Henrik Eriksson and the head of the IT department Håkan Lanshammar deserve a special mention. Thank you for running our department! The computer support group is acknowledged for all their help. The administrative staff is acknowledged for all their help, and for being such great company at the coffee breaks. The staff at restaurant Rullan is acknowledged for making such great food. Ulrik Ryberg deserves a special mention for his spirited comments delivered with a smile every day. Finally; Johan, Kjell, and Lars – I am happy that we had those long discussions about everything except work.

The experiments were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). Jonas Hagberg, Lennart Karlsson, Jukka Komminaho, and Tore Sundqvist at UPPMAX are

acknowledged for assistance concerning technical aspects. Thank you for all your help!

Uppsala University Library is acknowledged for their electronic subscriptions of ACM, IEEE, Springer, etc. These resources have been important for me. Jesper Andersson at Publishing and Graphic Services of Uppsala University Library is acknowledged for kind assistance in the publishing process of this Thesis.

I am happy that I have had the opportunity to perform a PhD project at Uppsala University, with its inspiring environment and its strong traditions of freedom of thought. I believe that freedom of thought is important in the pursuit of truth through mercy and nature (*Veritas gratiae [et] naturae*). I am thankful to those who maintain our freedom of thought.

VINNOVA (iStreams project, 2007-02916), ASTRON, and the Swedish Foundation for Strategic Research (SSPI project, grant RIT08-0041) are acknowledged for financial support. Anna Maria Lundins stipendiefond of Smålands nation and Liljewalchs resestipendium are acknowledged for travel grants.

In fall 2007, I had the pleasure of doing an internship at Google in Mountain View, California. This internship gave me further experience in practical software development. I am thankful to Jim Dehnert, Carole Dulong, and Silvius Rus for Google style management. I am thankful to my fellow interns for sharing the experience with me, and to the Uppsala University IT department alumni showing me the Bay Area. Zoran Radović needs a special mention for having me stay at his place in San José.

Before I applied for a position at UDBL, I performed a master thesis project at KDDI R&D Labs in Japan, supervised by Keiichiro Hoashi. It was under Hoashi-san's supervision I realized that I wanted to do more research. After completion of my master thesis project, Dan Ekblom suggested me to apply at UDBL by telling me that *"databaser är en framtidsbransch"* (database technology is a future industry).

Music has been an important source of inspiration during these years. I am thankful to Erik Hellerstedt and Uppsala Chamber Choir, Fredrik Ell and The Opera Factory, and Stefan Parkman and the Academy Chamber Choir of Uppsala for Monteverdi, Mozart, Mendelssohn, and Mäntyjärvi.

I am grateful to my friends and my family for encouragement and generous support – and for *ridendo dicere verum* (telling the truth through humour). In particular, I am grateful to my parents Ingrid and Sven Georg, my brother Johan and his fiancée Jorunn: Thank you for all the long seminars about life in general and research in particular.

Finally, Susanna, *con amore*: Thank you for always being there.

This work is dedicated to the memory of my grandparents Anna and Hans Wilhelm, Hannelore and Rudolf, for their generosity, and their never ending confidence and encouragement.

7 Bibliography

1. D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A.S. Maskey, A.Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik: The Design of the Borealis Stream Processing Engine. *Proc. CIDR 2005*.
2. D. Alves, P. Bizarro, P. Marques: Flood: elastic streaming MapReduce. *Proc DEBS 2010*.
3. H. Andrade, B. Gedik, K. L. Wu, P. S. Yu: Scale-Up Strategies for Processing High-Rate Data Streams in System S. *Proc. ICDE 2009*.
4. A. Arasu, M. Cherniack, E. Galvez, D. Maier, A.S. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts: Linear Road: A Stream Data Management Benchmark. *Proc. VLDB 2004*.
5. Ron Avnur and Joseph M. Hellerstein: Eddies: continuously adaptive query processing. *Proc. SIGMOD 2000.*
6. Y. Bai, H. Thakkar, H. Wang, C. Zaniolo: Optimizing Timestamp Management in Data Stream Management Systems. *Proc. ICDE 2007*.
7. M. Balazinska, H. Balakrishnan, S. R. Madden, M. Stonebraker: Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.* 33, 1, Article 3 (March 2008), 44 pages.
8. M. Balazinska, H. Balakrishnan, M. Stonebraker: Contract-Based Load Management in Federated Distributed Systems. *Proc. NSDI 2004*.
9. L. Brenna, J. Gehrke, M. Hong, D. Johansen: Distributed event stream processing with non-deterministic finite automata. *Proc. DEBS 2009*.
10. R. Chaiken R. Chaiken, B. Jenkins, P.Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB 2008*.
11. S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, V. Raman, F. Reiss, and M.A. Shah: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *Proc. CIDR 2003*.
12. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, S. Zdonik: Scalable distributed stream processing. *Proc. CIDR 2003*.
13. T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, R. Sears: Online aggregation and continuous query support in MapReduce. *Proc. SIGMOD 2010*.
14. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk: Gigascope: a stream database for network applications. *Proc. SIGMOD 2003*.
15. A. Das, J. Gehrke, M. Riedewald: Approximate join processing over data streams. *Proc. SIGMOD 2003*.
16. J. Dean, S. Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI 2004*.
17. P. M. Fischer, K. S. Esmaili, and R. J. Miller: Stream schema: providing and exploiting static metadata for data stream processing. *Proc. EDBT 2010*.

18. I. Foster, C. Kesselman, S. Tuecke: The Anatomy of the Grid – enabling virtual scalable organizations. *International Journal of High Performance Computing Applications* 15(3): 200–222 (2001).
19. M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy: Mining data streams: a review. *SIGMOD Rec.* 34, 2 (June 2005), pp 18–26.
20. B. Gedik, R.R. Bordawekar, P.S. Yu: CellJoin: a parallel stream join operator for the cell processor. *VLDB Journal (2009)* 18:501–519.
21. L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden: XStream: A Signal-Oriented Data Stream Management System. *Proc. ICDE 2008.*
22. L. Golab, M.T. Özsu: Data Stream Management. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2010.
23. N. K. Govindaraju, N. Raghuvanshi, D. Manocha: Fast and approximate stream mining of quantiles and frequencies using graphics processors. *Proc. SIGMOD 2005.*
24. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, Volume 41, 59–72, 2007.
25. M. Ivanova, T. Risch: Customizable Parallel Execution of Scientific Stream Queries. *Proc. VLDB 2005.*
26. N. Jain L. Amini, H. Andrade, R. King, Y. Park, P. Selo, C. Venkatramani: Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. *Proc. SIGMOD 2006.*
27. T. Johnson, M. S. Muthukrishnan, V. Shkapenyuk, O. Spatscheck: Query-Aware Partitioning for Monitoring Massive Network Data Streams. *Proc. SIGMOD 2008.*
28. S.J. Kazemitabar, U. Demiryurek, M. Ali, A. Akdogan, C. Shahabi: Geospatial stream query processing using Microsoft SQL Server StreamInsight. *Proc. VLDB 2010.*
29. B. Liu, Y. Zhu, M. Jbantova, B. Momberger, E.A. Rundensteiner: A dynamically adaptive distributed system for processing complex continuous queries. *Proc. VLDB 2005.*
30. B. Liu, Y. Zhu, E.A. Rundensteiner: Run-time operator state spilling for memory intensive long-running queries. *Proc. SIGMOD 2006.*
31. LOFAR, LOw Frequency Array, http://www.lofar.org. Accessed May 2011.
32. LOIS – the LOFAR Outrigger In Scandinavia, http://www.lois-space.net. Accessed May 2011.
33. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma: Query Processing, Resource Management, and Approximation in a Data Stream Management System. *Proc. CIDR 2003.*
34. R. Newton, L. Girod, M. Craig, S. Madden, G. Morrisett: WaveScript: A Case-Study in Applying a Distributed Stream-Processing Language. CSAIL Technical Report MIT-CSAIL-TR-2008-005, January 2008.
35. M.T. Özsu, and P. Valduriez: Principles of Distributed Database Systems, Second Edition. Prentice-Hall, 1999.
36. T. Risch, and V. Josifovski: “Distributed Data Integration by Object-Oriented Mediator Servers”, in *Concurrency and Computation: Practice and Experience J.* 13(11), John Wiley & Sons, September 2001, pp 933–953.
37. SIGMOD Records, Special sections on sensor network technology & sensor data management, 32 (4), December 2003 and 33 (1), March 2004.
38. N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker: Load shedding in a data stream manager. *Proc. VLDB* 2003.

39. F. Tian, D.J. DeWitt: Tuple Routing Strategies for Distributed Eddies. *Proc. VLDB 2003*.
40. D. Tsirogiannis, S. Harizopoulos, M. A. Shah: Analyzing the energy efficiency of a database server. *Proc. SIGMOD 2010*.
41. Uppsala University Linear Road implementations, http://www.it.uu.se/research/group/udbl/lr.html. Accessed May 2011.
42. S.D. Viglas, J.F. Naughton, J. Burger: Maximizing the output rate of multi-way join queries over streaming information sources. *Proc. VLDB 2003*.
43. S. Wang, E.A. Rundensteiner: Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. *Proc. EDBT 2009*.
44. L. Woods, J. Teubner, G. Alonso: Complex event detection at wire speed with FPGAs. *Proc. VLDB 2010*.
45. H. Yang, A. Dasdan, R.L. Hsiao, D.S. Parker: Map-reduce-merge: simplified relational data processing on large clusters. *Proc. SIGMOD 2007*.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 836

Editor: The Dean of the Faculty of Science and Technology

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2011

Distribution: publications.uu.se
urn:nbn:se:uu:diva-152255

Paper I

DOI=10.1109/ICDEW.2006.118

Processing high-volume stream queries on a supercomputer

Erik Zeitler and Tore Risch
Department of Information Technology, Uppsala University, Sweden
{erik.zeitler, tore.risch}@it.uu.se

Abstract– Scientific instruments, such as radio telescopes, colliders, sensor networks, and simulators generate very high volumes of data streams that scientists analyze to detect and understand physical phenomena. The high data volume and the need for advanced computations on the streams require substantial hardware resources and scalable stream processing. We address these challenges by developing data stream management technology to support high-volume stream queries utilizing massively parallel computer hardware. We have developed a data stream management system prototype for state-of-the-art parallel hardware. The performance evaluation uses real measurement data from LOFAR, a radio telescope antenna array being developed in the Netherlands.

1. Background

LOFAR [13] is building a radio telescope using an array of 25,000 omni directional antenna receivers whose signals are digitized. These digital data streams will be combined in software into streams of astronomical data that no conventional radio telescopes have been able to provide earlier. Scientists perform computations on these data streams to gain more scientific insight.

The data streams arrive at the central processing facilities at a rate of several terabits per second, which is too high for the data to be saved on disk. Furthermore, expensive numerical computations need to be performed on the streams in real time to detect events as they occur. For these data intensive computations, LOFAR utilizes an IBM BlueGene supercomputer and conventional clusters.

High-volume streaming data, together with the fact that several users wanting to perform analyses suggests the use of a data stream management system (DSMS) [9]. We are implementing such a DSMS called SCSQ (Super Computer Stream Query processor, pronounced *cis-queue*), running on the BlueGene computer. SCSQ scales by dynamically incorporating more computational resources as the amount of data grows. Once activated, continuous queries (CQs) filter and transform the streams to identify events and reduce data volumes of the result streams delivered in real time. The area of stream data management has gained a lot of interest from the database research community recently [1] [8] [14]. An important application area for stream-oriented databases is that of sensor networks where data from large numbers of small sensors are collected and queried in real time [21] [22]. The LOFAR antenna array will be the largest sensor network in the world. In difference to conventional sensor networks where each sensor produces a limited amount of very simple data, the data volume produced from each LOFAR receiver is very large.

Thus, DSMS technology needs to be improved to meet the demands of this environment and to utilize state-of-the-art hardware. Our application requires support for computationally expensive continuous queries over data streams of very high volumes. These queries need to execute efficiently on new types of hardware in a heterogeneous environment.

2. Research problem

A number of research issues are raised when investigating how new hardware developments like the BlueGene massively parallel computer can be optimally utilized for processing continuous queries over high-volume data streams. For example, we ask the following questions:

1. How is the scalability of the continuous query execution ensured for large stream data volumes and many stream sources? New query execution strategies need to be developed and evaluated.
2. How should expensive user-defined computations, and models to distribute these, be included without compromising the scalability? The query execution strategies need to include not only communication but also computation time.
3. How does the chosen hardware environment influence the DSMS architecture and its algorithms? The BlueGene CPUs are relatively slow while the communication is fast. This influences query distribution.
4. How can the communication subsystems be utilized optimally? The communication between different CPUs depends on network topology and the load of each individual CPU. This also influences query distribution.

3. Our approach

To answer the above research questions we are developing a SCSQ prototype. We analyze the performance characteristics of the prototype system in the target hardware environment in order to make further design choices and modifications. The analyses are based on a benchmark using real and simulated LOFAR data, as well as test queries that reflect typical use scenarios. These experiments provide test cases for prototype implementation and system re-design. In particular, performance measurements provide a basis for designing a system that is more scalable than previous solutions on standard hardware.

The CQs are specified declaratively in a query language similar to SQL, extended with streaming and vector processing operators. Vector processing operators are needed in the query language since our application requires extensive numerical computations over highvolume streams of vectors of measurement data. The queries involve stream theta joins over vectors applying non-trivial numerical vector computations as join criteria. To filter and transform streams before merging and joining them, the system supports sub-queries parameterized by stream identifiers. These sub-queries execute in parallel on different nodes.

A particular problem is how to optimize high-volume stream queries in the target parallel and heterogeneous hardware environment, consisting of BlueGene compute nodes communicating with conventional shared-nothing Linux clusters. Pre- and post-processing computations are done on the Linux clusters, while parallelizable computations are likely to be more efficient on the BlueGene. The distribution of the processing should be automatically optimized over all available hardware resources. When several different nodes are involved in the execution of a stream query, properties of the different communication mechanisms (TCP, UDP, MPI) substantially influence the query execution performance.

4. The hardware environment

Figure 1 illustrates the stream dataflow in the target hardware environment. The users interact with SCSQ on a Linux *front cluster* where they specify CQs. The *input streams* from the antennas are first pre-processed according to the user CQs in the Linux *back-end cluster*. Next, *BlueGene* processes the CQs over these pre-processed streams. The output streams from BlueGene are then post-processed in the front cluster and the result stream is finally delivered to the user. Thus, three parallel computers are involved and it is up to SCSQ to transparently and optimally distribute the stream processing between these.

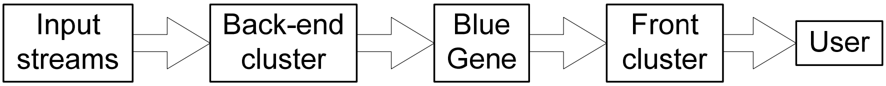


Figure 1. Stream data flow in the target hardware environment.

The hardware components have different architectures. The BlueGene features dual PowerPC 440d 700MHz (5.6 Gflops max) *compute nodes* connected by a 1.4 Gbps 3D torus network, and a 2.8 Gbps tree network [3].

Each compute node has a local 512 MB memory. The compute nodes run the *compute node kernel* (CNK) OS, a simple single-threaded operating system that provides a subset of UNIX functionality. Each compute node has two processors, of which normally one is used for computation and the other one for communication with other compute nodes. MPI is used for communication between BlueGene compute nodes, whereas communication with the Linux clusters utilizes *I/O nodes* that provide TCP or UDP. One important limitation of CNK is the lack of support for server functionality (no listen(), accept() or select() are implemented). Furthermore, two-way communication is expensive and should be avoided for time-critical code. Each I/O-node is equipped with a 1 Gbit/s network interface. In LOFAR's BlueGene, there are 6144 dual processor compute nodes, grouped in processing sets, or *psets*, consisting of 8 compute nodes and one I/O node. This I/O-rich configuration enables high volumes of incoming and outgoing data streams.

The Linux front and back-end clusters are IBM JS20 computers with dual PowerPC 970 2.2GHz processors.

5. The SCSQ system

Figure 2 illustrates the architecture of the SCSQ components running on the different clusters.

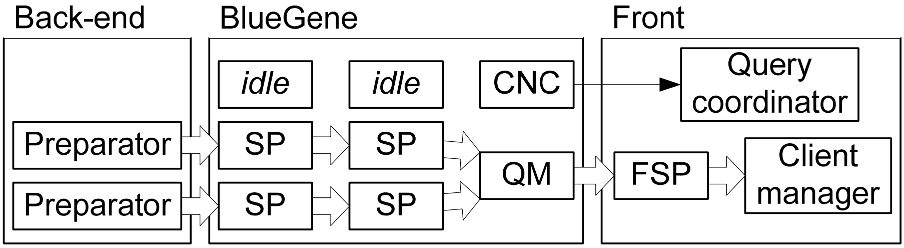


Figure 2. The SCSQ components. Double arrows indicate data streams.

On the front cluster, the user application interacts with a SCSQ *client manager*. The client manager is responsible for i) interacting with the user application, ii) sending CQs and meta-data, such as client manager identification, to the *query coordinator* for compilation.

The query coordinator is responsible for i) compiling incoming CQs from client managers, ii) starting one or more *front stream processors* (FSP) to do the post processing of the streams from the BlueGene, and iii) posting instructions to the BlueGene components for execution of CQs. When the query coordinator receives a new CQ from a client manager, the query coordinator initiates new FSPs for post-processing of that CQ. It also maintains a request queue of CQs and other instructions to be processed by the Blue-

Gene. This queue is regularly polled by the BlueGene *compute node coordinator* (CNC) (single arrow in Figure 2).

The CNC is responsible for i) retrieving new CQs and instructions from the query coordinator, ii) assigning and coordinating *stream processors* on the compute nodes, and iii) monitoring the execution of all stream processors. The BlueGene processors to be used by SCSQ are initiated once when the system is set up. The CNC is always executing on a single node while all other nodes are stream processors waiting for instructions from the CNC. When the CNC retrieves a new CQ, it assigns one idle stream processor to be the new *query master* for that query.

A query master is responsible for i) compiling and executing its stream query, ii) delivering the result to an FSP on the front cluster previously initiated by the query coordinator, iii) starting new stream processors of subqueries if needed, iv) communicating with the backend cluster to retrieve input data, and v) monitoring the execution of its stream query. When a query master receives a CQ it is compiled and then the execution is started. If the query master determines that additional stream processors are needed for some stream subqueries, it dynamically requests the CNC to assign new ones. The query master then sends the subqueries to the new stream processors for execution. Each stream processor may in turn start new subqueries when so required. Stream queries may be terminated either by explicit user intervention or by some stop condition in the query. Therefore, the stream processors also exchange control messages to initialize and terminate stream queries. Control messages are also used to regulate the stream flow between the processors

The only difference between a stream processor and a query master is that the query master delivers its result to an FSP in the front cluster using TCP, while a stream processor delivers its result stream through MPI to the stream processor or query master that initiated it

Nodes participating in the processing of a stream are called *working nodes*. Stream processors, query masters, and FSPs are all working nodes.

When a working node needs measurements from an input stream it initiates TCP communication for that stream through its *preparator*. A preparator is a working node running on the back-end cluster wrapping one or more input streams.

The set-up of a stream query generates a distributed query execution tree, as illustrated by the double arrows in Figure 2.

We have implemented the first SCSQ prototype and are evaluating it. All BlueGene and front node functionality for execution of single user queries have been implemented. We have used this implementation to analyze bandwidth properties of the I/O nodes and strategies for efficient buffering in the MPI and TCP communication subsystems.

The implementation of SCSQ nodes is based on Amos II (Active Mediator Object System) [18] [19], which is modified to allow execution of continuous queries over streams in the target hardware environments. The SCSQ modules are extensible by linking user-defined functions written in compiled C. On the front and back-end clusters, dynamic linking is allowed. However, only static linking is allowed on BlueGene. As a consequence, all user-defined stream operators written in C must be statically linked with the stream processor executable for the BlueGene. To configure dynamically the stream processors at run-time we utilize a built-in Lisp interpreter to communicate code between the front cluster and the BlueGene. All time-critical code running on the BlueGene is written in C and statically linked.

6. Related work

The SCSQ implementation is related to research in DSMSs, parallel and distributed databases, continuous query processing, and database technology for scientific applications.

A promising approach to achieve the high performance, flexibility, and expressiveness required is to develop a distributed DSMS running on highly connected clusters of main memory nodes [2] [7] [12], which is extensible through user-defined data representations and computational models [10]. Most of the DSMS, e.g. [6] [8] [14] [15] [20], are designed for rather small data items and a relatively small cost of the stream operators per item. In contrast, SCSQ is intended for a very high total stream volume, large data item sizes, and computationally expensive scientific operators and filters.

The use of extensible database technology where database queries call user-defined functions in the database engine have been shown very useful for astronomical applications [17]. Parallelization of user-defined functions has been studied in [16].

Distributed execution of expensive user-defined stream query functions has been studied in the recently proposed Grid Stream Data Manager (GSDM) [10] [11], an object-relational DSMS for scalable scientific stream query processing. GSDM features a framework for predefined and customized parallelization schemes, which distribute the execution of user-defined stream query functions over the Grid. Like SCSQ, GSDM is intended for scalable on-line analysis using expensive user-defined stream query functions over high-volume scientific data streams from instruments and simulations.

However, unlike all other DSMSs, SCSQ will be optimized for a heterogeneous target hardware environment including a BlueGene supercomputer.

7. Ongoing work

Query execution scalability is achieved by developing query processing strategies able to utilize an increasing number of compute nodes while optimally utilizing the communication facilities.

To generate local query execution plans on each stream processor we employ query optimization strategies based on heuristics and a simple cost model.

Queries are distributed based on the need to execute sub-queries in parallel. Currently, each stream processor can execute only one sub-query. Any stream processor can at run-time request idle stream processors from the CNC to execute sub-queries. This allows dynamic reconfiguration of the distributed query execution plan.

The performance monitoring subsystem in each stream processor measures the performance of different phases of stream query execution. It is currently used to evaluate the characteristics of different execution strategies. However, the same mechanism will also be used to optimize the stream query distribution itself. Since our system allows dynamic reassignment of stream processors we will use the performance monitoring subsystem for adaptive run-time query re-optimization. This is necessary since sudden bursts in the measured signals may require execution plans to be dynamically reconfigured.

To analyze the system and understand the issues that are relevant to the LOFAR application we are developing a benchmark. The benchmark includes real and simulated data as well as queries from the radio astronomy application domain. We are initially concentrating on queries that detect transients among a large number of incoming streams. We scale the number of incoming streams and optimize throughput and latency as the data volume grows. Therefore, we scale not only the data volume but also the computation time in our experiments.

A stream oriented communication protocol between stream processors is developed based on MPI. We measure the characteristics for different communication methods between the stream processors. The communication latency and bandwidth depend on the topology and the load of the nodes. For example, nodes far apart have long latency but may have a high bandwidth, since there are many communication links between them that can be used in parallel. On the other hand, highly loaded intermediate nodes slow down communication [5]. These characteristics will influence query decomposition and distribution.

The query execution performance depends on the utilization of each stream processor. The utilization of a stream processor depends on the relation between its stream rate and computational load. Each stream processor is buffering its incoming and outgoing streams. The buffer utilization of a stream processor indicates the load balance between communication and processing. Each stream processor monitors its buffer utilization and adapts the flow rate by sending control messages regularly. In an overflow situation, different policies can be devised, for example: load shedding by dropping incoming data [23], simplifying aggregation operators [4], sending control messages that slow down sub query stream processors, or asking CNC for more stream processors.

It is also important to analyze the performance of queries involving expensive operators. We investigate the scalability over large numbers of high-volume input streams that are merged by computationally expensive stream combination functions from the benchmark. The goal is to understand how to distribute the streams and computations optimally in the heterogeneous target hardware environment.

7. Ongoing work

This work is supported by LOFAR.

7. References

1. Daniel J. Abadi et al, "Aurora: a new model and architecture for data stream management", *The VLDB Journal*, Springer, 12(2) 2003, pp 120–139.
2. Daniel J. Abadi et al, "The Design of the Borealis Stream Processing Engine", in *The Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA 2005, pp 277–289.
3. George Almási et al, "Implementing MPI on the BlueGene/L Supercomputer", *Lecture Notes in Computer Science, Volume 3149*, Jan 2004, pp 833–845.
4. Brian Babcock, Mayur Datar, Rajeev Motwani, "Load Shedding for Aggregation Queries over Data Streams", in *Proc. of the International Conference on Data Engineering (ICDE 2004)*, Boston, USA, pp 350–361.
5. Gyan Bhanot et al, "Optimizing task layout on the Blue Gene/L supercomputer", *IBM Journal of Research and Development*, Volume 49, Number 2/3, 2005, pp 489–500.
6. Donald Carney et al, "Monitoring Streams – A New Class of Data Management Applications", in *Proc. Of the 28th Int'l Conf. on Very Large Databases (VLDB'02)*, Hong Kong, China, 2002, pp 215–226.
7. Mitch Cherniack et al, "Scalable distributed stream processing", in *The First Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar*, CA 2003.

8. Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk, “Gigascope: A Stream Database for Network Applications”, in *Proc. Of the ACM SIGMOD Conference on Management of Data*, San Diego, CA 2003, pp 647–651.
9. Lukasz Golab and M. Tamer Özsu, “Issues in data stream management”, SIGMOD Record, 32(2), 2003, pp 5–14.
10. Milena Ivanova and Tore Risch, ”Customizable Parallel Execution of Scientific Stream Queries”, in Proc. *Of the 31st Int’l Conf. on Very Large Databases (VLDB’05)*, Trondheim, Norway 2005, pp 157–168.
11. Milena Ivanova, ”Scalable Scientific Stream Query Processing”, in *Uppsala Dissertations from the Faculty of Science and Technology 66*, Acta Universitatis Upsaliensis, Uppsala 2005, http://user.it.uu.se/~udbl/Theses/MilenaIvanovaPhD.pdf.
12. Bin Liu et al, “A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries”, in *Proc. Of the 31st Int’l Conf. on Very Large Databases (VLDB’05)*, 2005, pp 1338–1341.
13. LOFAR, http://www.lofar.nl/.
14. Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman, “Continuously adaptive continuous queries over streams”, in *Proc. Of the ACM SIGMOD Conference on Management of Data*, Madison, Wisconsin 2002, pp 49–60.
15. Rajeev Motwani et al, “Query processing, approximation, and resource management in a data stream management system”, in *The First Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA 2003.
16. Kenneth W. Ng and Richard R. Muntz, “Parallelizing user-defined functions in distributed object-relational DBMS”, in *International Database Engineering and Applications Symposium (IDEAS)*, Montreal, Canada 1999, pp 442–450.
17. María A. Nieto-Santisteban et al, “When Database Systems Meet the Grid”, in *The Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA 2005, pp 154–161.
18. Tore Risch and Vanja Josifovski, “Distributed Data Integration by Object-Oriented Mediator Servers”, in *Concurrency and Computation: Practice and Experience J.* 13(11), John Wiley & Sons, September 2001, pp 933–953.
19. Tore Risch, Vanja Josifovski, and Timour Katchaounov “Functional Data Integration in a Distributed Mediator System”, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Data Management – Modeling, Analyzing and Integrating Heterogeneous Data*, Springer 2003, pp 211–238.
20. Elke A. Rundensteiner et al, “CAPE: A Constraint-Aware Adaptive Stream Processing Engine”, in Nauman Chaudhry, Kevin Shaw, and

Mahdi Abdelguerfi (eds.): *Stream Data Management*, Advances in Database Systems Series, Springer 2005, pp 83–111.

21. Special Section on Sensor Network Technology and Sensor Data Management, *SIGMOD Record*, 32(4), December 2003.
22. Special Section on Sensor Network Technology and Sensor Data Management (Part II), *SIGMOD Record*, 31(1), March 2004.
23. Nesime Tatbul, et al, "Load Shedding in a Data Stream Manager", in Proc. *Of the 29th Int'l Conf. on Very Large Databases (VLDB'03)*, Berlin, Germany, pp 309–320

Paper II

DOI=10.1109/ICDCSW.2007.88

Using stream queries to measure communication performance of a parallel computing environment

Erik Zeitler and Tore Risch
Department of Information Technology, Uppsala University, Sweden
{erik.zeitler, tore.risch}@it.uu.se

Abstract– We have developed a data stream management system that supports declarative stream queries running over high data volumes in a supercomputing environment. To enable specification of massively parallel computations our query language provides processes as query language objects. The queries call process construction functions that execute stream subqueries assigned to a CPU. Such queries can be used to define query functions that parallelize computations. The CPU assignment is normally automatic, but can also be influenced by the user. We show how this enables performance measurements of different communication topologies in a heterogeneous hardware environment containing a Linux cluster and a BlueGene

1. Introduction

LOFAR [13] is currently building a radio telescope using an array of 25,000 omni-directional antenna receivers whose signals are digitized into data streams of very high rate. Scientists perform computations on these data streams to gain scientific insight. The LOFAR antenna array will be the largest sensor network in the world. The receivers produce raw data streams that arrive at the central processing facilities at a rate, which is too high for the data to be saved on disk. Furthermore, advanced numerical computations are performed on the streams in real time to detect astronomical events as they occur. For these data-intensive computations, LOFAR utilizes an IBM BlueGene supercomputer combined with conventional Linux clusters.

To enable stream processing in heterogeneous and massively parallel environments of LOFAR's kind we have developed a data stream management system called SCSQ (Super Computer Stream Query processor, pronounced *sis-queue*) [22]. SCSQ transparently executes on a variety of hardware platforms and operating systems, including MS Windows, Linux, and BlueGene. To support transparent streaming in a heterogeneous environment consisting of clusters with different communication subsystems, SCSQ features internal drivers that currently support MPI and TCP for carrying streams.

Continuous queries (CQs) are declaratively specified in a query language, SCSQL (pronounced *sis-kel*). To maximize throughput of streams and computations it is important to parallelize CQs into continuous subqueries, each executing as a separate process on a CPU. To enable a customized parallelization, SCSQL provides *stream processes* (SPs) as first-class objects in queries. The user associates subqueries with SPs. Massively parallel computations are defined in terms of sets of subqueries, executing on sets of stream processes.

Properties of the different CPUs, communication mechanisms, and operating systems substantially influence query execution performance. These properties are stored in a database, which is used by the query optimizer when assigning an SP to a CPU.

In implementing the query optimizer, it is crucial to understand how different strategies to distribute computation and communication influence the execution performance. It is particularly important for our application to maximize the bandwidth of the data streams from the receivers into the compute nodes of the BlueGene. The incoming streams are critical paths of the application since a sub-optimal input data rate will slow down the entire stream processing chain. In this paper, we use SCSQL queries in order to measure the streaming bandwidth of different communication topologies between a back-end Linux cluster and the BlueGene, as well as between compute nodes inside the BlueGene. SCSQ optionally allows the user to influence the choice of CPU to which an SP is assigned. We use this facility to specify different communication topologies in SCSQL.

In summary, we present the following contributions:

- The introduction of stream processes enables specification of massively parallel computations in the query language SCSQL.
- We show how SCSQL can be used to measure streaming bandwidth inside a BlueGene using different communication topologies. The results from these measurements provide a basis for automatic CPU allocation strategies inside BlueGene.
- Analogously, inbound streaming bandwidth from a Linux cluster to BlueGene is measured using different communication topologies specified in SCSQL to provide a basis for automatic set-up of inbound streaming communication.

Before presenting the results, we give an overview of the SCSQ system and the heterogeneous hardware environment in which the experiments were performed. An introduction to the query language SCSQL is also given, and we show how to formulate mapreduce [8] and radix fft [12] queries using SCSQL.

2. The SCSQ system

In this section, we first describe the LOFAR hardware environment that is used for our experiments. Then, we present the overall SCSQ architecture and finally we describe the features of SCSQL that are used in the experiments.

2.1. Hardware environment

Figure 1 illustrates the stream dataflow in the LOFAR hardware environment. Users interact with SCSQ on a Linux front-end cluster. Another Linux back-end cluster first receives the streams from the sensors where they are pre-processed. Next, the BlueGene processes these streams. The output streams from the BlueGene are then post-processed in the front-end cluster and the result stream is finally delivered to the user. Thus, three computer clusters are involved.

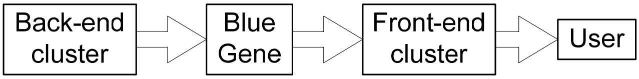


Figure 1. Stream data flow in the LOFAR environment.

The hardware components are characterized by different architectures. The BlueGene features dual PowerPC 440d 700MHz (5.6 Gflops max) compute nodes connected by a 1.4 Gbps 3D torus network, and a 2.8 Gbps tree network. The time it takes for a compute node to send data to another one depends on the relative locations of these nodes in the torus, and how loaded the nodes between them are. Each compute node has a local 512 MB memory. The compute nodes run the compute node kernel (CNK) OS [15], a simple single-threaded operating system that provides a subset of UNIX functionality. One important limitation of CNK is the lack of support for server capabilities (no *listen()*, *accept()* or *select()*). Each compute node has two CPUs, of which normally one is used for computation and the other one for communication with other compute nodes. A native MPI implementation is used for communication between BlueGene compute nodes, whereas communication with the Linux clusters utilizes I/O nodes that provide TCP or UDP. Each I/O-node is equipped with a 1 Gbit/s network interface. I/O nodes are only used for communication, and cannot be used for computations. In LOFAR's BlueGene, there are 6144 dual processor compute nodes, grouped in processing sets of 8 compute nodes and one I/O node.

The Linux front and back-end clusters are IBM JS20 computers with dual PowerPC 970 2.2GHz processors. Each computer in the back-end cluster has a 1 Gigabit Ethernet interface connected via a switch to the BlueGene.

2.2. SCSQ architecture

Figure 2 illustrates a query that is set up for execution in the hardware environment. SCSQ users interact with the client manager, in which they specify CQs using SCSQL. The execution of a CQ forms a directed acyclic graph of running processes (RPs), each executing the subquery specified in one SP.

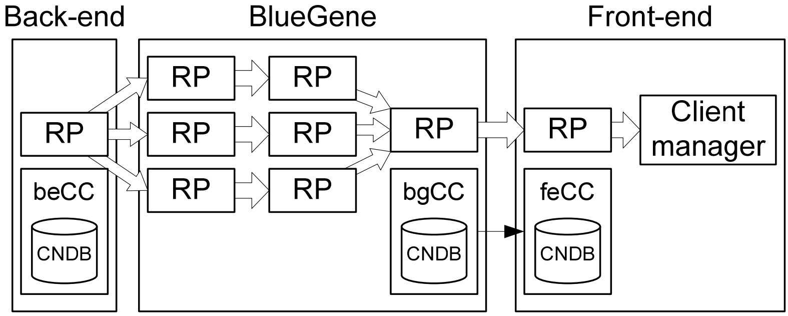


Figure 2. Set-up of a CQ for execution in SCSQ. Wide arrows indicate data streams.

The execution of CQs may be stopped either by explicit user intervention or by a stop condition in the query that makes the stream finite. When a CQ is stopped, its RPs are terminated. RPs regularly exchange control messages, which are used to regulate the stream flow between them and to terminate execution upon a stop condition.

When a user submits a CQ, it is optimized and started in the client manager. When the client manager identifies an SP, the sub-query of that SP is registered with the coordinator of the cluster where the sub-query is to be executed (*feCC*, *bgCC*, or *beCC* in Figure 2). Then, the coordinator starts an RP to execute the sub-query. In addition, an RP can dynamically start new RPs by requesting them from the cluster coordinator of the cluster where the new RP is started.

Since the BlueGene lacks server functionality, sub-queries from the client manager to be executed on the BlueGene are registered with the *feCC*. The *bgCC* retrieves new sub-queries from the *feCC* by polling. As BlueGene compute nodes can execute only one process, each new RP in BlueGene is assigned to a new compute node.

Each cluster coordinator maintains an internal *compute node database* (CNDB) containing the properties and status of the possibly thousands of compute nodes in its cluster. A *node selection algorithm* in the cluster coordinator starts the new RP on a suitable compute node by querying its CNDB. Currently, a naïve node selection algorithm is used, returning the next available node.

2.3. Running Processes

An RP has the components shown in Figure 3. It is responsible for i) compiling its subquery into a local *Stream Query Execution Plan*, SQEP and interpreting it, ii) delivering the result to other RPs, its *subscribers*, iii) dynamically requesting new RPs from a coordinator if needed, iv) retrieving its input data from other RPs, its *producers*, and v) monitoring the execution of its SQEP.

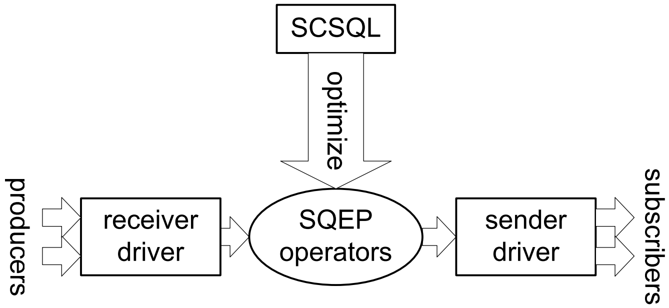


Figure 3. A SCSQ running process.

The operators in the SQEP are executed when data arrives. Incoming data is buffered in a *receiver driver* and de-marshaled (materialized) into objects. Streams of materialized data objects are delivered to the operators of the SQEP. The objects resulting from the operators are passed on to the *sender driver*, which marshals them and sends the buffer contents to subscribers. Objects are dynamically de-allocated when no longer needed by any operator. The sender and receiver drivers can use various network protocols for carrying the streams. We have implemented stream carrier protocols based on MPI and TCP. SCSQ supports the use of MPI on any MPI enabled cluster. MPI is always used inside the BlueGene as that is the only allowed protocol, while TCP is always used when communicating between clusters. The MPI sender and receiver drivers contain double buffers so that one buffer can be processed while the other one is read or written.

2.4. SCSQL

SCSQL is a query language similar to SQL, but extended with streams and *stream processes* as first-class objects. Stream processes allows dynamic parallelization of continuous queries, which is used in this paper to measure the performance of a massively parallel and heterogeneous computing environment. This section introduces SCSQL.

All data in SCSQ is represented by *objects* in SCSQL. The relation between first-class objects in SCSQL is illustrated in Figure 4. A *stream* is an object that represents (possibly unbounded) sequences of any kind of objects. The result of a continuous subquery is a stream. Continuous subqueries are assigned to *stream processes*. Users of SCSQL define parallel and distributed stream computations by assigning continuous subqueries to stream processes.

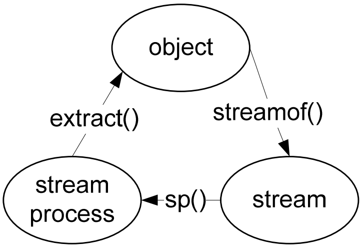


Figure 4. The relation between streams, stream processes and objects in SCSQL.

The function *sp(s, c)* assigns the subquery *s* to a new stream process to be run in cluster *c*. The function *extract(p)* requests elements (objects) from the subquery assigned to stream process *p*. If *p* ever terminates, *extract(p)* also terminates. The function *streamof(e)* transforms the output of any expression *e* to a stream. This is useful when a stream output is desired from functions that do not naturally return streams, e.g. *count()*, which returns a single integer.

To enable easy handling of sets of parallel stream processes, the function *spv(**s**, c)* assigns each subquery in the set ***s*** to a new stream process on some compute node in the cluster *c*, and returns a set (bag) of handles to the assigned stream processes.

The function *merge(**p**)* generalizes *extract()* by requesting elements from each stream process in ***p***. *merge()* terminates when (if ever) the last stream process in ***p*** terminates.

The use of the data types representing streams and stream processes allows specification of parallel and distributed CQs with different topologies. *merge()* provides stream combinations, while variables bound to sets of stream processes provide parallel execution.

For example, the distributed grep mapreduce [8] query using 1000 parallel *grep* calls is specified in SCSQL as follows:

```
merge(spv(
     select grep("pattern", filename(i))
     from integer i
     where i in iota(1,1000)));
```

Line 1 contains the reduce predicate. In this case there is no reduction, so there is no function outside the merge. On line 2, the subquery performs a grep for a pattern on the i^{th} filename in a table. Each subquery executes in a separate process. Line 4 specifies the degree of parallelism, in this case 1000 processes. *iota(n,m)* generates all integers from *n* to *m*. In this example, *iota()* is used to generate 1000 duplicates of the select stream, and to provide a key to the *filename()* table.

Splitting of streams is specified by referencing common variables bound to stream processes, as illustrated by the following query function, which implements the radix2 parallelization of FFT [12] for a stream source named *s*.

```
create function radix2(string s)
                       ->stream
as select radixcombine(merge({a,b}))
from sp a, sp b, sp c
where a=sp(fft(odd (extract(c))))
  and b=sp(fft(even(extract(c))))
  and c=sp(receiver(s));
```

The *receiver()* function returns a stream of 1D arrays of signal data. *odd(x)* and *even(x)* obtain odd and even elements from array *x*, respectively. *radixcombine()* combines the results from the partial FFT algorithms working in parallel.

Optionally, the SCSQL user can constrain the allowed compute nodes for the node selection algorithm by specifying a *node allocation* query as an extra argument to *sp()* and *spv()*. This query returns a stream of allowable compute nodes in preferred allocation order, called the *allocation sequence*. The allocation sequence is passed to the node allocation algorithm of the cluster coordinator when it allocates the RP for an SP. The node selection algorithm will choose the first available node in the allocation sequence. (In case the stream contains no available node, the query will fail.) Thus, allocation sequences allow the user to restrict and prioritize the node selection order.

In the next section we show how we utilize allocation sequences to enforce different communication topologies. This helps us determine how to achieve maximum streaming bandwidth. The gained knowledge will be used to improve the node selection algorithm.

3. Streaming performance

Using SCSQL and its allocation sequence option, we set up different communication topologies and measure how they influence the streaming bandwidth. The following experiments are performed:

1. The streaming bandwidth between RPs executing on compute nodes inside the BlueGene is measured. For good performance of *extract()* and *merge()*, SCSQ buffers incoming elements in the receiver driver. Different buffer settings for MPI streams inside the BlueGene are evaluated. Furthermore, explicit node selections are used to measure different communication topologies inside the BlueGene.
2. The bandwidth is measured for communicating streams from RPs in the back-end cluster to RPs inside the BlueGene. The impact on the bandwidth of different node selections in the back-end cluster and in the BlueGene is measured. As the system uses TCP for communication between the back-end cluster and BlueGene, we rely on the buffering of the TCP stack in this case.

In all experiments, the streams contain arrays of numerical data, as required by the application. The bandwidth is computed by measuring the total time to communicate a finite stream of 3MB arrays between stream processes. Small array sizes increase processing overhead, and we are primarily interested in communication performance, hence the large array size. Each experiment was performed five times in order to achieve low variance in the measurements.

3.1. Intra-BG streaming

The following experiments are performed:

1. We measure the bandwidth of point-to-point communication between two RPs, which execute on different BlueGene compute nodes.
2. We measure the bandwidth of stream merging from two RPs to a third one.

In the experiments, we vary the buffer sizes of the communication subsystem. We also compare the usage of double and single buffering.

Figure 5 illustrates the set-up of the point-to-point measurement. *a* generates a stream of large arrays and *b* counts the total number of arrays in the finite stream extracted from a. The result of the count is sent to the front-end. Since only one number is transmitted from *b* to the client manager, the total time measured is dominated by the time for streaming the data from *a* to *b*.

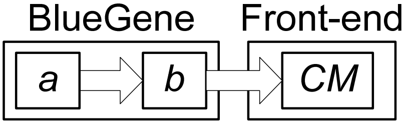


Figure 5. Intra-BG point to point streaming.

The sending RP *a* (i) generates the arrays, (ii) marshals them into a send buffer and (iii) transmits the send buffers when they are full. The receiving

RP *b* (iv) receives buffers, (v) de-marshals the buffer contents, (vi) allocates new arrays, (vii) counts them, and (viii) de-allocates them. Only the result of the count is streamed to the front-end. The query is expressed in SCSQL as follows:

```
select extract(b)
from sp a, sp b
where b=sp(streamof(count(extract(a)))
           'bg',0) and
   a=sp(gen_array(3000000,100),'bg',1);
```

gen_array() generates the finite stream of 100 arrays of size 3MB each. The calls to *sp()* assign the streams to new stream processes on a compute nodes in the 'bg' (BlueGene) cluster. The function *count()* counts the number of elements in a bag. The function *streamof()* makes a stream of the output of *count()*. Allocation sequences are specified in the third arguments of the *sp()* calls as single node identifier values (0 and 1), since we want to exactly specify the selected node here. The selected node cannot be busy in this query since we know what nodes are allocated and where they are located in the BlueGene.

Figure 6 shows the bandwidth of intra-BG point-to-point streaming. As can be seen, the optimal buffer size is 1000 bytes for both single and double buffering. The drop-off above the 1000-byte buffer size is probably due to cache misses. The performance degradation for buffers smaller than 1000 bytes buffer size is because 1K is the smallest message size that can be exchanged in the BlueGene 3D torus. Furthermore, we observe that double buffering pays off for large buffers. A number of bumps are clearly seen in the double-buffer curve. No explanation for this phenomenon can be found, but it is nevertheless statistically significant.

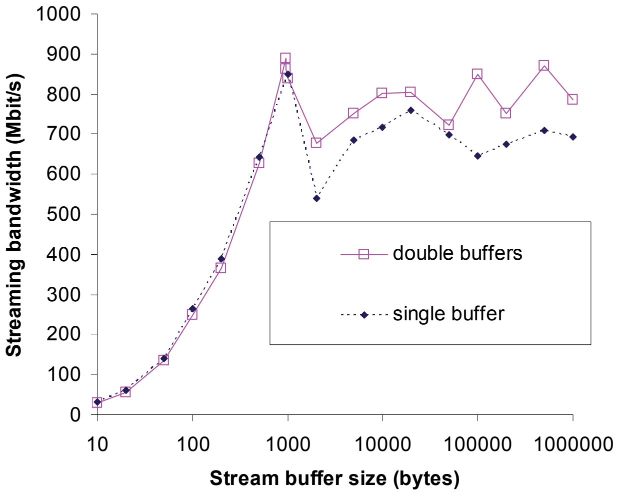


Figure 6. Point to point streaming performance.

For stream merging, we measure the throughput when two RPs send data to a third one. When messages are sent between non-adjacent nodes in BlueGene, they must be routed through the communication co-processors of the nodes in between. Communication will be slower if these co-processors are busy.

Since the enumeration of compute nodes in the BlueGene 3D torus is known, it is easy to specify the two communication topologies in Figure 7 using the allocation sequence feature of SCSQL. In both cases *c* merges data from the streams of *a* and *b*. The two experiments are defined in SCSQL by varying *x* and *y* in the following query:

```
1  Select extract(c)
2  from sp a, sp b, sp c
3  where c=sp(count(merge({a,b})), 'bg',0)
4  and a=sp(gen_array(3000000,100),'bg',x)
5  and b=sp(gen_array(3000000,100),'bg',y);
```

count() counts the total number of arrays in the merged streams *a* and *b*. The explicit node selections 0, *x*, and *y* on lines 3–5 specify the exact BlueGene node numbers where the RPs execute.

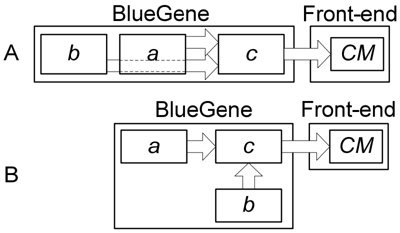


Figure 7. Alternative BlueGene node selections for stream merging.

Figure 7A shows a sequential node selection, where MPI messages from *b* to *c* are routed through the communication co-processor of *a*. Here, *x*=1 and *y*=2 to select compute nodes arranged as in figure 7A. Figure 7B shows a *balanced node selection*, where messages from *a* and *b* are sent directly to *c* over individual communication channels. Here, *x*=1 and *y*=4 to select compute nodes arranged as in figure 7B.

Figure 8 shows the total streaming input bandwidth at node *c* for stream merging using the two node selection strategies. Both single and double buffering is evaluated. Analogously to the results of the point-to-point streaming experiment shown in Figure 6, we expected double buffering to pay off for large buffers.

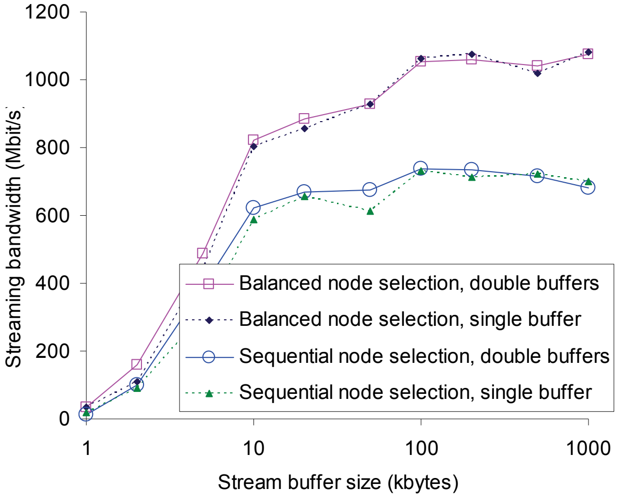


Figure 8. Alternative BlueGene node selections for stream merging.

We observe the following:

1. The streaming bandwidth depends highly on the compute nodes to which the RPs are allocated. This is because of the topology of the BlueGene 3D torus interconnection network.
2. The benefit of double buffering is less significant than that of point-to-point communication.
3. Finally, an interesting observation is that buffers smaller than 10K are much slower for stream merging than for point-to-point communication.

The reason for better performance for large buffers when merging streams is that the single-threaded communication co-processor of *c* must handle data streams from both *a* and *b*. In *c*, it switches between receiving messages from *a* and *b*. Less frequent switching improves communication. By contrast, for point-to-point communication, all messages come from the same source, so the co-processor does not pay any switching penalty. Thus, sending larger but fewer messages is beneficial for stream merging while the opposite holds for point-to-point communication.

3.2. BG inbound streaming

We conducted experiments for six different ways to inject data streams into the BlueGene, named Query 1 through Query 6. The inbound streaming bandwidth of each query is measured for different numbers of parallel input streams by altering a query variable *n*. In all experiments, the total number of arrays in all the finite streams produced in the back-end cluster is counted. The output data from the query is a single integer. Thus, the time to execute the query is dominated by time for streaming the data from the back-end cluster into the BlueGene.

Query 1 investigates the streaming bandwidth when all streams 1 through *n* are sent from a single node in the back-end cluster through a single I/O node into a single compute node in the BlueGene. Query 2 differs from Query 1 in that several compute nodes in the back-end cluster are injecting data into BlueGene. This is to investigate whether parallelization over several compute nodes in the back-end cluster will improve the streaming bandwidth compared to that of Query 1. Queries 3 and 4 transfer all data over one single I/O node but parallelize the receiving over several compute nodes in BlueGene. This is to see whether parallelizing the receiving compute nodes will improve the streaming bandwidth in comparison to all streams being received on a single compute node. Queries 5 and 6 are analogous to Queries 3 and 4 but parallelize the data injection into BlueGene over several I/O nodes. By intuition, query 6 can be expected to achieve the highest streaming bandwidth of all queries, since parallel back-end compute nodes inject data through parallel I/O channels.

The distribution pattern of Query 1 is shown in Figure 9. All streams are produced on a_1 through a_n, executing on the same compute node in the back-end cluster. All streams are sent to *b* inside BlueGene, which merges and counts them. *c* extracts the count from *b* unchanged and sends it to the client manager in the front cluster. The reason to include *c* in this query is to make all experiments comparable.

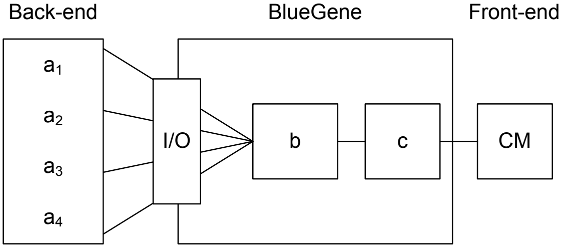


Figure 9. Execution distribution of Query 1.

Query 1 is formulated in SCSQL as follows:

```
 1  select extract(c) from
 2  bag of sp a, sp b, sp c,
 3  integer n
 4  where c=sp(extract(b),'bg')
 5  and   b=sp(count(merge(a)), 'bg')
 6  and   a=spv(
 7    (select gen_array(3000000,100)
 8    from integer i where i in iota(1,n)),
 9              'be', 1)
10  and n=4;
```

The explicit node selection on line 9 assigns all back-end SPs to compute node 1 in the back-end cluster.

The execution distribution of Query 2 is shown in Figure 10. In this query, a_1 through a_n execute on different compute nodes in the back-end cluster. All streams are sent to *b* inside the BlueGene, which merges them and counts the total number of arrays. *c* passes on the count unchanged as before.

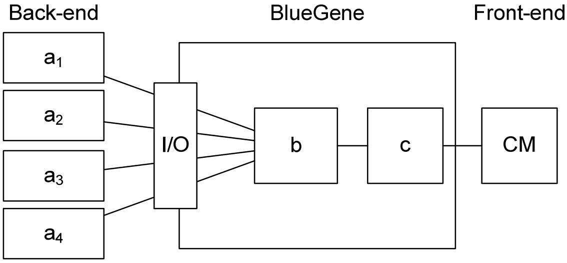


Figure 10. Execution distribution of Query 2.

Query 2 is formulated in SCSQL as follows:

```
 1  select extract(c) from
 2  bag of sp a, sp b, sp c,
 3  integer n
 4  where c=sp(extract(b), 'bg')
 5  and   b=sp(count(merge(a)), 'bg')
 6  and   a=spv(
 7    (select gen_array(3000000,100)
 8    from integer i where i in iota(1,n)),
 9              'be', urr('be'))
10  and n=4;
```

Only the last argument to *spv()* in line 9 differs from Query 1. Here we want to assign each SP in a to different compute nodes. The node allocation function *urr(cl)* retrieves a stream from the CNDB of cluster *cl* of compute node identifiers where each identifier represents a new available node in the cluster in a round-robin fashion. This allocation sequence stream is later shipped back to the cluster coordinator by the *spv()* call to be used by the node selection algorithm. By shipping stream handles we avoid unnecessary data shipping.

The execution distribution of Query 3 in Figure 11 parallelizes the aggregation over several RPs, each one running on a separate receiving BlueGene compute node. Compute nodes belonging to the same pset use the same I/O node for inbound communication. All streams from the back-end cluster are sent to the BlueGene compute nodes through a single I/O node by specifying b_1 through b_n to belong to the same *pset*.

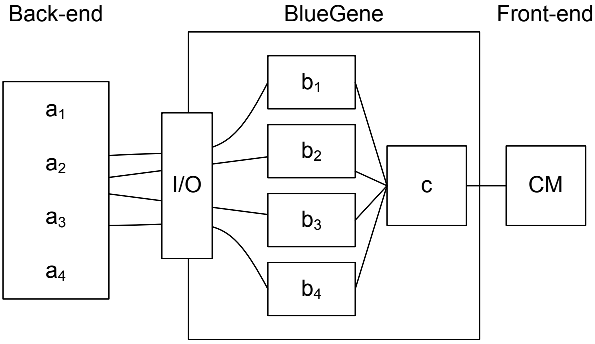


Figure 11. Execution distribution of Query 3.

Query 3 is defined by the following SCSQL query.

```
select extract(c) from
bag of sp a, bag of sp b, sp c,
 3  integer n
 4  where c=sp(streamof(sum(merge(b)),
 5             'bg'))
 6  and   b=spv(
 7    (select streamof(count(extract(p)))
 8     from sp p
 9     where p in a),
10              'bg', inPset(1))
11  and a=spv(
12   (select gen_array(3000000,100)
13    from integer i where i in iota(1,n)),
14              'be', 1)
15  and n=4;
```

On line 10, the processor selection function *inPset(k)*, which returns a stream of compute node identifiers in *pset* number *k*, forces all SPs to belong to the same *pset*.

The execution distribution of Query 4, shown in Figure 12, differs from Query 3 in that the back-end RPs run on different compute nodes.

Query 4 is defined by the following SCSQL query.

```
 1  select extract(c) from
 2  bag of sp a, bag of sp b, sp c,
 3  integer n
 4  where c=sp(streamof(sum(merge(b))),
 5  'bg')
 6  and   b=spv(
 7    (select streamof(count(extract(p)))
 8    from sp p
 9    where p in a),
10              'bg', inPset(1))
11  and a=spv(
12   (select gen_array(3000000,100)
13    from integer i where i in iota(1,n)),
14              'be', urr('be'))
15  and n=4;
```

The only difference from Query 3 is the call to *urr()* on line 14, enforcing all RPs to execute on different compute nodes in the back-end cluster.

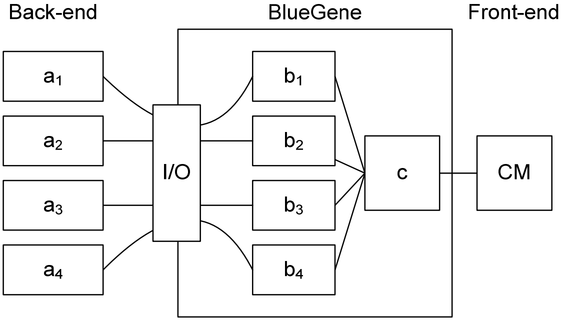


Figure 12. Execution distribution of Query 4.

The execution distribution of Query 5, shown in Figure 13, utilizes different I/O nodes for the communication of streams from the back-end cluster.

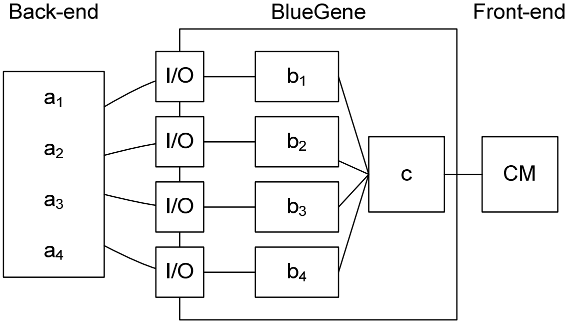


Figure 13. Execution distribution of Query 5.

The following SCSQL query defines Query 5.

```
select extract(c) from
bag of sp a, bag of sp b, sp c,
integer n
where c=sp(streamof(sum(merge(b))),
           'bg')
and   b=spv(
 (select streamof(count(extract(p)))
  from sp p
  where p in a),
            'bg', psetrr())
and a=spv(
 (select gen_array(3000000,100)
  from integer i where i in iota(1,n)),
          'be', 1) and n=4;
```

This query differs from Query 3 in the processor selection on line 10. The function *psetrr()* returns a stream of BlueGene compute node numbers, where each succeeding node number belongs to a new *pset* in a round-robin fashion. This will parallelize the inbound communication over different I/O nodes, since compute nodes belonging to different *psets* will use different I/O nodes.

Finally, the execution distribution of Query 6 is shown in Figure 14. This query differs from Query 5 in that back-end stream processes run on different nodes in the back-end cluster.

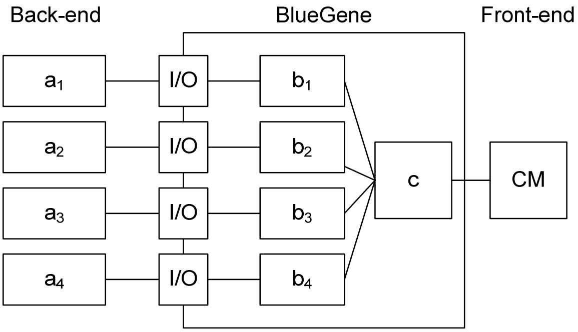


Figure 14. Execution distribution of Query 6.

The following SCSQL query defines Query 6.

```
 1  select extract(c) from
 2  bag of sp a, bag of sp b, sp c,
 3  integer n
 4  where c=sp(streamof(sum(merge(b))),
 5            'bg')
 6  and   b=spv(
 7   (select streamof(count(extract(p)))
 8    from sp p
 9    where p in a),
10             'bg', psetrr())
11  and a=spv(
12   (select gen_array(3000000,100)
13    from integer i where i in iota(1,n)),
14           'be', urr('be'))
15  and n=4;
```

The difference from Query 5 is the call to *urr()* on line 14, assigning all SPs to different compute nodes in the back-end cluster.

Figure 15 compares the BG inbound streaming bandwidth for Queries 1 through 6. *n* is the number of RPs in the back-end cluster that inject streams into the BlueGene. The y-axis measures the total inbound streaming bandwidth from the back-end cluster into the BlueGene compute nodes.

We observe the following:

1. Queries 1 through 4 all communicate using a single I/O node on the BlueGene. They all have significantly lower bandwidth than that of Queries 5 and 6. Thus, as expected, it is favorable to use many I/O nodes.
2. The streaming bandwidth of Queries 3 and 4 are slightly better than that of Queries 1 and 2. Changing from one to two receiving BlueGene compute nodes off-loads the communication burden, while further increasing the number of receiving compute nodes is not worthwhile since the total streaming bandwidth does not increase. Hence, it pays off to increase the number of receiving compute nodes from one to two even if there is only one I/O node available. This communication topology should be used in the node selection algorithm when compute nodes are available but the number of I/O nodes is limited.

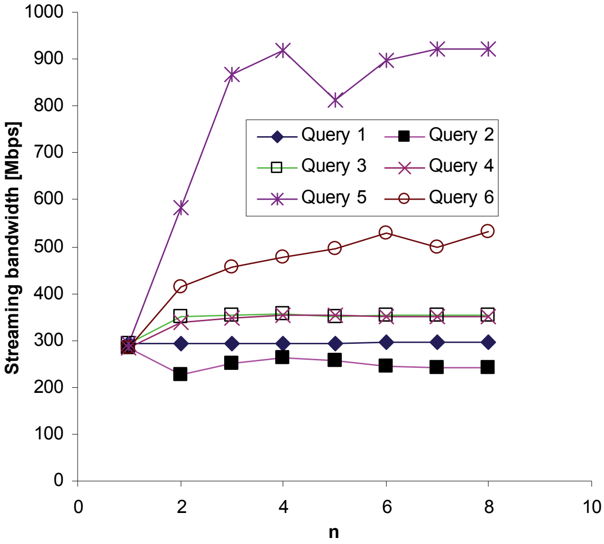


Figure 15. Results for queries 1 through 6.

3. As can be seen, the best streaming bandwidth is achieved for Query 5, which peaks at ~920 Mbps. It is surprising that a single 1 Gbps connection from the back-end cluster is faster than four separate 1 Gbps connections as in Query 6. It is thus faster to inject streams over different I/O nodes from the same back-end cluster compute node than from different back-end compute nodes. This indicates coordination problems in the I/O node when communicating with many outside nodes. The conclusion is that the node selection algorithm should attempt to co-locate back-end RPs to the same compute node until saturation.

4. Similarly, the streaming bandwidth of Query 1 is better than that of Query 2, indicating that it is better to run as many RPs on the same back-end node as possible rather than running them on different back-end nodes.
5. In Query 5, there is a significant performance dip for *n*=5. This is probably because there were only four I/O nodes available on the BlueGene partition where the experiments were performed. For *n*>4, compute nodes have to share I/O nodes and therefore the bandwidth decreases. In this case, the node selection algorithm could resort to increase the number of receiving compute nodes as in observation (2) above.

We are currently investigating how to extend the node selection algorithm with the above knowledge.

4. Related work

There are many data stream management system (DSMS) implementations, some of which execute on a single node [3] [5] [7] [9] [14] [17] [19], and some are distributed [1] [4] [10] [11] [16] [20] [21]. Some of these implementations provide high-level SQL-like query languages such as STREAM [3] and TelegraphCQ [5]. In [3], streams are treated as continuously updated relations, while in [5] they are implemented as external functions emitting tuples as an unbounded bag. Unlike all other DSMS projects, the SCSQ data model treats both streams and processes as first class objects. Stream processes allow users to specify massively parallel and distributed computations in CQs by dynamically starting stream processes at run time. Furthermore, the SCSQ user can optionally even influence the location for the node assignments, which has been used in this paper to measure communication performance.

Tribeca [19] provides *pipes* as first-class objects in its query language. These pipes are similar to our stream data type but Tribeca provides no parallelization of their execution, and no dynamic process creation. Similarly, WaveScope [9] provides a stream processing language where arbitrary computations can be specified as functions over streams in a non-distributed stream processing environment.

SPC [11] was evaluated using the Linear Road Benchmark [2] on a highly parallel PC cluster. However, the distribution is manual and SPC has no query language as SCSQL.

Dynamic load balancing for distributed DSMSs has been studied in [4] [20] [21]. In Borealis [1], a central coordinator migrated stream processing operators between nodes using load statistics [21]. Medusa nodes migrate operators between each other using computational economy methods [4]. In D-CAPE [20], different initial distribution and redistribution strategies were experimentally evaluated. The explicit optional node placement primitives of

SCSQ can be used for static load balancing, as was done in this paper to measure performance. In addition, SQCQL provides user primitives to specify how to logically parallelize algorithms dynamically through stream processes. This is orthogonal to load balancing.

Distributed execution of expensive user-defined stream query functions has been studied in GSDM [10]. GSDM distributes its stream computations by selecting and composing distribution templates from a library. By contrast, all distribution topologies are expressed as SCSQL queries. In [10], only one parallelization topology (partition, compute, and combine) for user-defined functions is provided. Mapreduce [8] also provides another special distribution topology, namely map and reduce. SCSQL allows the specification of any communication topology. Sawzall [17] features a high-level language that enables compact specifications of massively parallel mapreduce tasks. However, Sawzall is restricted to the mapreduce distribution topology. Furthermore, Sawzall lacks many advanced operators for aggregation and computation, whereas SCSQ features all common stream operators including window aggregation.

5. Conclusions and future work

We presented the SCSQ system, which is a DSMS that runs in a massively parallel hardware environment featuring a BlueGene. Several different kinds of clusters are included in the execution of a continuous query.

The query language SCSQL provides both *streams* and *stream processes* as first class objects. The users of SCSQ are thereby given control over the parallelization of stream queries and functions. Users specify parallel computations by assigning sub-queries to stream processes executed in parallel. We have shown how to parallelize mapreduce and radix FFT using SCSQL. Furthermore, SCSQL also allows the user to specify allocation sequences that restrict and prioritize which compute nodes to be chosen for execution. Using such allocation sequences, we specified different physical communication topologies for a mapreduce-like query. These experiments measured different topologies for inbound streaming into BlueGene. The measurement showed that in order to achieve reasonable performance, a considerable amount of I/O nodes must be designated to handle input streams. In our experiments, we also discovered that it is favorable to use as few as possible input compute nodes in the back-end cluster. This indicates that the BlueGene I/O is a bottleneck. These experiments provide basis for extending the node selection algorithm.

Moreover, our experiments show that the flexibility of the query language provides a powerful tool for investigating the streaming performance of any computer environment. These experiments can easily be repeated on other

kinds of clusters to understand their streaming performance and to provide basis for specific node selection algorithms.
SCSQL allocation sequences were also used to measure the bandwidth of communication between compute nodes inside BlueGene using native MPI. The impact of buffer sizes and double buffering used in the MPI communication was measured for different topologies. The optimal stream buffer size for MPI communication inside BlueGene was highly dependent on whether point-to-point or merging stream communication was performed. In general, the buffer should be much larger in the case of stream merging. Double buffering proved to be less important in our experiments.

Furthermore, the location of the BlueGene compute nodes highly affects the inter-node communication since data may be routed through intermediate nodes in the 3D-torus of BlueGene. We showed that stream merging performs up to 60% better if no busy intermediate nodes are involved in the communication.

We are currently experimenting with refinements of the node selection algorithm for the BlueGene based on the results of this paper. It should be investigated whether it is possible to parameterize the node selection algorithm so that it can be used in any parallel hardware environment. In the current hardware configuration, we have only four I/O nodes and four nodes in the back-end cluster. It remains to be investigated what happens for large amounts of back-end and I/O nodes. It is also important to analyze the performance of continuous queries involving expensive functions. Further measurements could be made using benchmarks such as The Linear Road Benchmark [2]. The goal is to understand how to distribute streams and computations optimally in a heterogeneous hardware environment.

Acknowledgements

This work is supported by ASTRON.

References

1. D. J. Abadi et al, “The Design of the Borealis Stream Processing Engine”, Proc. CIDR 2005 Conf., Asilomar, CA.
2. A. Arasu et al, “Linear Road: A Stream Data Management Benchmark”, Proc. VLDB 2004 Conf., Toronto, Canada, pp 480–491.
3. A. Arasu et al, “STREAM: The Stanford Data Stream Management System”, http://infolab.stanford.edu/stream/.
4. M. Balazinska, H. Balakrishnan, and M. Stonebraker, “Contract-Based Load Management in Federated Distributed Systems”, Proc. 1st Symp. on Networked Systems Design and Implementation, USENIX Association 2004, pp 197 – 210.

5. S. Chandrasekaran et al, “TelegraphCQ: Continuous Dataflow Processing for an Uncertain World”, Proc. CIDR 2003, Asilomar, CA.
6. M. Cherniack et al, “Scalable distributed stream processing”, Proc. CIDR 2003, Asilomar, CA.
7. C. Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk, “Gigascope: A Stream Database for Network Applications”, Proc. SIGMOD 2003 Conf., San Diego, CA, pp 647–651.
8. J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Proc. 6th Symp. on OS Design and Implementation, USENIX Association 2004, pp 137 – 150.
9. L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, S. Madden, “The Case for a Signal-Oriented Data Stream Management System”, Proc. CIDR 2007, Asilomar, CA.
10. M. Ivanova and T. Risch, ”Customizable Parallel Execution of Scientific Stream Queries”, Proc. VLDB 2005, Trondheim, Norway, pp 157–168.
11. N. Jain et al, “Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core”, Proc. SIGMOD 2006, Chicago, IL, USA.
12. V. Kumar, A. Grama, A. Gupta, and G. Karypis, ”Introduction to Parallel Computing”, The Benjamin Cummings Publishing Company, Inc., 1994.
13. LOFAR, http://www.lofar.nl/.
14. S. Madden, M. A. Shah, J. M. Hellerstein, and Vijayshankar Raman, “Continuously adaptive continuous queries over streams”, Proc. SIGMOD 2002 Conf., Madison, WI, USA, pp 49–60.
15. J. E. Moreira et al, “Blue Gene/L programming and operating environment”, IBM J. of Research and Development, 49(2/3), 2005, pp 367–376.
16. K. W. Ng and R. R. Muntz, “Parallelizing user-defined functions in distributed object-relational DBMS”, Proc. IDEAS Symp. 1999, Montreal, Canada, pp 442–450.
17. R. Pike, S. Dorward, R. Griesemer, S. Quinlan, “Interpreting the Data: Parallel Analysis with Sawzall”, Scientific Programming Journal 13:4, pp. 227–298.
18. E. A. Rundensteiner et al, “CAPE: A Constraint-Aware Adaptive Stream Processing Engine”, in N.Chaudhry, K. Shaw, and M. Abdelguerfi (eds.): “Stream Data Management”, Advances in Database Systems Series, Springer 2005, pp 83–111.
19. M. Sullivan, A. Heybey, “Tribeca: A System for Managing Large Databases of Network Traffic”, Proc. USENIX Conf., New Orleans, 1998.
20. T. Sutherland, B. Liu, M. Jbantova, E. A. Rundensteiner, ”D-CAPE: Distributed and SelfTuned Continuous Query Processing”, Proc. CIKM 2005 Conf., Bremen, Germany.

21. Y. Xing, S. Zdonik, J.-H. Hwang, “Dynamic Load Distribution in the Borealis Stream Processor”, Proc. ICDE 2005 Conf., Tokyo, Japan.
22. E. Zeitler, T. Risch, “Processing high-volume stream queries on a super-computer”, Proc. ICDE 2006 Workshops, Atlanta, GA, USA.

Paper III

Gyozo Gidofalvi, Torben Bach Pedersen, Tore Risch, and Erik Zeitler. 2008. Highly scalable trip grouping for large-scale collective transportation systems. In Proceedings of the 11th international conference on Extending database technology: Advances in database technology(EDBT '08). ACM, New York, NY, USA, 678-689.
DOI=10.1145/1353343.1353425
http://doi.acm.org/10.1145/1353343.1353425

Highly Scalable Trip Grouping for Large–Scale Collective Transportation Systems

Gyozo Gidofalvi
Geomatic ApS
gyg@geomatic.dk

Torben Bach Pedersen
Aalborg University
tbp@cs.aau.dk

Tore Risch
Uppsala University
Tore.risch@it.uu.se

Erik Zeitler
Uppsala University
erik.zeitler@it.uu.se

Abstract– Transportation-related problems, like road congestion, parking, and pollution are increasing in most cities. In order to reduce traffic, recent work has proposed methods for vehicle sharing, for example for sharing cabs by grouping “closeby” cab requests and thus minimizing transportation cost and utilizing cab space. However, the methods proposed so far do not scale to large data volumes, which is necessary to facilitate large-scale collective transportation systems, e.g., ride-sharing systems for large cities.

This paper presents *highly scalable trip grouping algorithms*, which generalize previous techniques and support input rates that can be orders of magnitude larger. The following three contributions make the grouping algorithms scalable. First, the basic grouping algorithm is expressed as a continuous stream query in a data stream management system to allow for a very large flow of requests. Second, following the divide-and-conquer paradigm, four space-partitioning policies for dividing the input data stream into sub-streams are developed and implemented using continuous stream queries. Third, using the partitioning policies, parallel implementations of the grouping algorithm in a parallel computing environment are described. Extensive experimental results show that the parallel implementation using simple adaptive partitioning methods can achieve speed-ups of several orders of magnitude without significantly effecting the quality of the grouping.

1. Introduction

Transportation-related problems, like congestion, parking, and pollution are increasing in most cities. Waiting in traffic jams not only degrades the quality of social life, but according to estimates, the economic loss caused by traffic jams in most countries is measured in billions of US dollars yearly. Parking is also a serious problem. In some large cities, it is estimated that as many as 25% of the drivers on the road are only looking for empty parking places. This again causes unnecessary congestion. Finally, the increasing number of vehicles idling on the roads results in an unprecedented carbon emission, which has unquestionably negative effects on the environment.

By reducing the number of vehicles on the roads, Collective Transportation (CT) clearly provides a solution to these problems. Public transportation, the most common form of CT, tries to meet the general transportation demands of the population at large. By generalizing the transportation needs, the individual is often inconvenienced by long wait times at off-peak hours or between connections, and a limited number of access points (bus, metro, train stops) from which the individual is forced to use other methods of transportation (walking, bicycling, using a private car). Ride-sharing, or car pooling, which is another form of CT is becoming widespread in metropolitan areas. Ride-sharing is often encouraged by local transportation authorities by facilitating car pool lanes that are only accessible to multiple-occupancy vehicles and by eliminating tolls on bridges and highways for these vehicles. Despite all the encouragement, there is a tremendous amount of unused transportation capacity in the form of unoccupied seats in private vehicles. This fact can mainly be attributed to the lack of effective systems that facilitate large-scale ride-sharing operations. The systems that do exist [3, 15, 22] are either 1) offered from a limited number access points due to the system infrastructure constraints, 2) have inadequate methods for the positioning of trip requests and/or vehicles, or 3) have either inefficient or ineffective methods for matching or grouping trip requests and trip offers.

Yet another form of CT, namely cab-sharing, was recently proposed [12]. The key idea of cab-sharing is to use unoccupied cab space to reduce the cost of transportation, ultimately resulting in direct savings to the individual. The described Cab-Sharing System (CSS) overcomes most of the above limitations of existing ride-sharing systems. In particular, at the heart of the system is a trip grouping algorithm that is able to find subsets of closeby trip requests, which can be grouped into collective cab fares to minimize the transportation cost, or equivalently maximize the savings to the user. Using a simple implementation in standard SQL, assuming a reasonable number (high spatio-temporal density) of trip requests, the trip grouping algorithm was demonstrated to be able to group trip requests effectively. The trip grouping algorithm can be generalized to facilitate other CT systems, e.g., a ride-sharing system. However, as it is demonstrated in the present paper, due to its algorithmic complexity, the grouping algorithm scales poorly as the volume of trip requests increases. This limits its applicability to facilitate large-scale CT systems, such as a metropolitan or nation-wide ride-sharing system.

To make the trip grouping algorithm scale to input rates several orders of magnitude larger than in a typical cabsharing application, this paper makes the following three contributions. First, using a Data Stream Management System (DSMS), SCSQ [24], the trip grouping algorithm is expressed as a continuous stream query to allow for continuous processing of large trip request streams. Second, following the divide-and-conquer paradigm, *static* and *adaptive* versions of two space-partitioning policies (*point quad* and *KD partitioning*) for dividing the input data stream into sub-streams are developed and implemented using continuous stream queries. Finally, using the partitioning policies, the grouping algorithm is implemented using a data stream management system in a parallel computing environment. The parallelization of the implementation is facilitated by using an extension of the query language, in which processes are query language objects. Extensive experimental results show that the parallel implementation using simple partitioning methods can achieve speed-ups of several orders of magnitude without significantly affecting the quality of the grouping. In particular, an adaptive partitioning method called *adaptive KD partitioning* achieves the best overall performance and grouping quality.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 defines the vehicle-sharing problem, reviews the operational aspects of a recently proposed Cab-Sharing System (CSS), describes and analyzes a trip grouping algorithm that solves the vehicle-sharing problem and is employed to facilitate the CSS. Furthermore, a new Ride-Sharing System (RSS) is proposed, and the trip grouping algorithm is adapted to meet the application requirements of the proposed RSS. Section 4 describes the main contributions of the paper in making the trip grouping algorithm highly scalable, hence applicable in large-scale CT system, such as an RSS. Section 6 describes and analyzes the results of the experiments that were conducted to measure the performance of the proposed highly scalable trip grouping algorithm. Finally, Section 7 concludes and points to future research directions.

2. Related work

The optimization of CT has been studied in the scientific community for years [5, 20]. However, with the exception of the work presented in [12], on which the present paper is based, it is believed that no previous research has considered the online grouping of trip requests. The problem of grouping n objects into a number of groups is in general referred to as the clustering problem, which is an extensively researched problem in computer science. However, the unique requirements of the problem of vehicle-sharing mean that general clustering techniques have limited applicability.

Vehicle-sharing as a form of CT has been considered in industrial and commercial settings. For example, most taxi companies in larger cities have been offering the possibility of shared transportation between a limited number of frequent origins and destinations. Scientifically very little is known about the computational aspects of these vehicle-share operations. However, the computer systems supporting such operations are likely to be semiautomatic, to perform batch-grouping of requests, and to suffer from scalability problems. In comparison, the trip grouping algorithm proposed in this paper is automatic, performs online-grouping or requests, and is highly scalable.

More automatic systems that perform online optimization of vehicle-sharing also exist [3, 15, 22]. These systems however perform a computationally easier task. They either match pairs of trip requests only [15] or are offered from/between a limited set of locations [3, 15, 22]. Additionally, the high volume scalability of these systems has not been demonstrated. Nonetheless, the analysis in [21] and the existence of these systems are evidence that the problem considered by the paper is real and has industrial applications.

Parallel processing of high-volume data streams has been considered by several papers [4, 7, 17, 18, 23, 24, 25]. Some of these study the parallelization of continuous stream queries [7, 23, 24]. GSDM [7] decomposes the computation of a single continuous stream query into a partition, a compute, and a combine phase. In GSDM, the distributed execution strategies are expressed as *data flow distribution templates*, and queries implementing the three phases are specified in separate scripts. In contrast, SCSQ [24] exposes the parallelization phases to the query language so that the distribution patterns becomes part of a single parallel, continuous stream query. This paper utilizes the stream processing engine and query language in SCSQ to express and evaluate different (parallel) stream processing strategies for an RSS.

In GSDM, two different stream partitioning strategies are considered: *window distribute* (WD) and *window split* (WS). In WD, entire logical windows are distributed among compute nodes. In WS, an operator dependent stream split function splits logical windows into smaller ones and assigns them to particular compute nodes for processing. WS has several advantages over WD. First, in applications where the execution time of the stream query scales superlinearly with the size of the logical window, WS provides superior parallel execution performance over WD. Second, in realtime response systems, where the query scales superlinearly, WD is not applicable as it can introduce severe delays in the result stream. Third, in systems where the quality of the results that are computed in parallel are highly dependent on the tuples inside the logical windows of the compute nodes, WD provides inferior results in quality over WS, because individual tuples are not considered in the partition phase. As all of the above three conditions hold in the case of vehicle-sharing, WD is clearly not of interest. WS is similar to the spatial stream partitioning methods presented in this paper in the sense that both presented partitioning methods consider individual tuples in the partitioning process. However, in the static cases no windows are formed over the stream, but rather tuples are assigned to compute nodes based on a general partitioning table. In contrast, in the adaptive cases windows are formed over the stream, the partitioning table is periodically updated based on the tuples in a window, and then tuples are assigned to compute nodes the same way as in the static case.

Database indices support the efficient management and retrieval of data in large databases. In particular, spatial indices support efficient retrieval of spatial objects, i.e., objects that have physical properties such as location and extent. Spatial indices can be divided into two types: *data partitioning* and *space partitioning* spatial indices [19]. The partitioning mechanisms used in spatial indices have a close relation to the partitioning performed in the present paper.

Data partitioning indices usually decompose the space based on Minimum Bounding Rectangles (MBRs). A primary example is the R-tree that splits space with hierarchically nested, and possibly overlapping Minimum Bounding Rectangles (MBRs) [13]. However, for the application at hand, data partitioning schemes are not well suited for several reasons. They often use a non-disjoint decomposition of space. Consequently, a naïve partitioning based on MBRs could either assign requests to several partitions, and hence later to several shares, or could assign requests from a region where several MBRs overlap to several partitions, thereby potentially eliminating the chance for good matches. While a disjoint partitioning of space could be derived based on the MBRs, computation to derive such a partitioning would be complex and potentially expensive, and the derived partitions will most likely not be balanced.

On the other hand, space partitioning indices decompose the entire space into disjoint cells. These disjoint cells can be based on a regular grid, or on an adaptive grid. Regular grids can result in empty partitions because of skewed data distributions. Hence, a regular grid is not well-suited for the application at hand as it does not support load-balancing.

Quad-trees partition the space into four quadrants in a recursive fashion [6]. Quad-trees divide each region into four equally sized regions, while *point quad trees* [19] allow the size of the regions to be dynamic. Quad-trees have been extended to higher dimensions also. One of the space partitioning methods used in this paper is quite similar to a 1-level deep, four dimensional point quad tree with the exception that in the herein considered space partitioning method a split point is not necessarily a data point. The *k*-*d*-tree is a space partitioning spatial index that hierarchically divides each dimension into two along each of the *k* dimensions [1, 2]. The other partitioning method used in this paper corresponds to a 1-level deep, four dimensional *k*-*d*-tree.

3. Vehicle-sharing

Large-scale, personalized, on-demand CT systems need efficient and effective computer support. Systems providing this support have two aspects. The first aspect is operational aspect as to how information is communicated between the user and the service provided by the system, and how trip requests are processed. There second aspect is computational or algorithmic and deals with how the optimization of CT is performed. The following subsections study an existing CT system and propose a new one. Section 3.1 formalizes the vehicle-sharing problem, adopted from [12]. Section 3.2 describes the operational aspects of a Cab-Sharing System (CSS) – an instance of a CT system in which the shared ve- hicles are cabs. Section 3.3 describes the computational or algorithmic aspects of the trip grouping algorithm employed in the CSS. Section 3.4 describes the problems that arise when the trip grouping algorithm is applied in larger scale CT systems. Section 3.5 proposes a Ride-Sharing Service (RSS) and describes its operational requirements. Finally, Section 3.6 describes how the trip grouping algorithm in Section 3.3 can be modified to meet these requirements.

3.1 The Vehicle-Sharing Problem

Let $\boldsymbol{R}^2$ denote the 2-dimensional Euclidean space, and let $T \equiv \boldsymbol{N}^+$ denote the totally ordered time domain. Let $R = \{r_1, . . . , r_n\}$ be a set of *trip requests* $r_i = <t_r, l_o, l_d, t_e>$, where $t_r \in T$ is the *request time*, $l_o \in \boldsymbol{R}^2$ and $l_d \in \boldsymbol{R}^2$ are the *origin* and *destination locations*, and $t_e \geq t_r \in T$ is the *expiration time*, i.e., the latest time by which the trip request must be accommodated. A trip request $r_i = <t_r, l_o, l_d, t_e>$ is valid at time t if $t_r \leq t \leq t_e$. $\Delta t = t_e - t_r$ is called the *wait time* of the trip request. A vehicle-share $s \subseteq R$ is a subset of the trip requests. A vehicle-share is valid at time t if all trip requests in s are valid at time t. Let $|s|$ denote the number of trip request in the vehicle-share. Let $d(l_1, l_2)$ be a distance measure between two locations l_1 and l_2. Let $m(s, \mathrm{d}(., .))$ be a method that constructs a valid and optimal pick-up and drop-off sequence of requests for a vehicle-share s and assigns a unique distance to this sequence based on $d(., .)$. Let the *savings* p for a trip request $r_i \in s$ be $p(r_i, s) = 1 - m(s, d(., .)) / [|s| \cdot m(\{r_i\}, d(., .))]$. Then the vehicle-sharing problem is defined as follows.

DEFINITION 1. For a given maximum vehicle-share size K, and minimum savings `min_savings` $\in [0, 1]$, the vehicle-sharing problem is to find a disjoint partitioning $S = \{s_1 \uplus s_2 \uplus \ldots\}$ of R, such that $\forall s_j \in S$, s_j is valid, $|s_j| \leq K$, and the expression

$$\sum_{s_j \in S} \sum_{r_i \in s_j} p(r_i, s_j)$$

is maximized under the condition that $\forall r_i \in s_j : p(r_i, s_j) \geq \texttt{min\_savings}$ or $\{r_i\} = s_j$.

3.2 Overview of the Cab-Sharing System

The Cab-Sharing System (CSS) proposed in [12] is a Location-Based Service (LBS) in the transportation domain. In its most simple form, it is accessible to the user via a mobile phone through an SMS interface. The components and operation of the CSS is depicted in Figure 1 and can be described as follows. The user inputs two addresses with an optional maximum time that s/he is willing to wait. The service in turn then:

1. geocodes the addresses,
2. calculates an upper bound on the cost of the fare,
3. validates the user's account for sufficient funds,
4. submits the geocoded request to a pool of pending requests,
5. within the maximum wait time period finds a nearly optimal set of "closeby" requests using a number of heuristics (described in Section 3.3),
6. delivers the information about the set (request end points, and suggested pickup order) to the back-end cab dispatch system,
7. delivers information about the fare (estimated time or arrival, cost, savings, etc…) to the involved users.

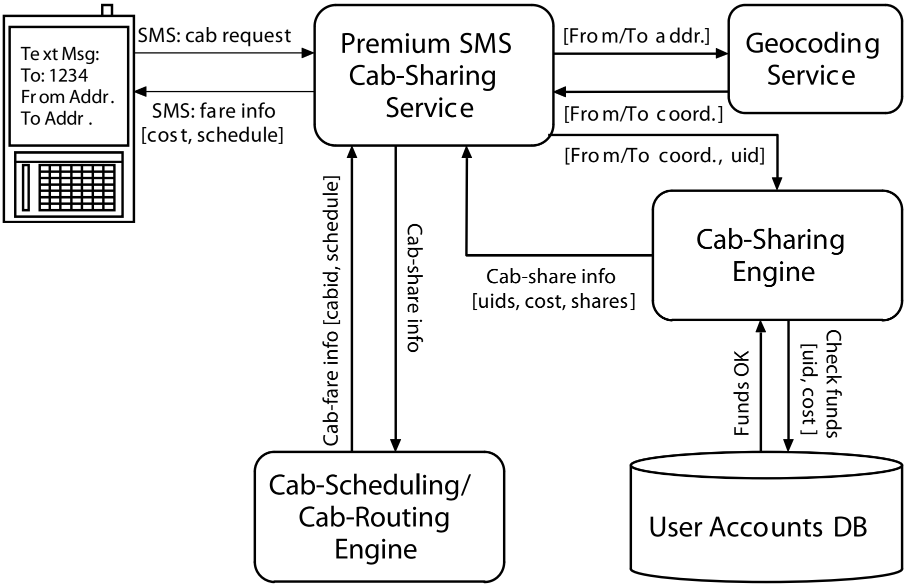


Figure 1. Cab-sharing service components and process.

3.3 A Trip Grouping Algorithm

Finding the optimal solution to the vehicle-sharing problem is computationally difficult. Given n requests, the number of possible disjoint partitionings, where the size of the vehicle-shares is exactly K is:

$$\binom{n}{K} \times \binom{n-K}{K} \times \ldots \times \binom{2K}{k} \times \binom{k}{k} = \frac{n!}{\lceil n/K \rceil \times K!} .$$

In the case of n = 100 and K = 4, this expression evaluates to a number that has 155 digits. The number of possible disjoint partitions, where the size of the vehicle-shares is at most K = 4 is even larger. Clearly, evaluating all possible options and selecting the most optimal one is not a feasible approach. Instead, the Trip Grouping (TG) algorithm at the heart of the CSS tries to derive a nearly optimal solution by employing a number of heuristics and approximations. The steps of the TG algorithm along with the applied heuristics and approximations are described next.

1. Distinguish between the set of expiring trip requests (R_x) and all valid requests (R_q). Wait with mandatory grouping of trip requests until expiration time. A request can also be grouped into a vehicle-share before its expiration time with another expiring request. This *lazy* heuristic does not make the algorithm miss out on an early cost-effective grouping for the request, but rather gives the requests more opportunities to be part of a grouping.
2. Based on the distance measure $d(.,.)$, define a pairwise *fractional extra cost* (FEC) between two requests and calculate it for every pair of expiring and valid requests. In the TG algorithm the fractional extra cost between two requests r_i and r_j (w.r.t. r_i) is defined as $FEC(r_i, r_j) = [d(r_i.l_o, r_j.l_o) + d(r_i.l_d, r_j.l_d)] / d(r_i.l_o, r_i.l_d)$. In the case when the distance measure $d(.,.)$ is the Euclidean distance, the calculations of fractional extra costs between three requests r_1, r_2, and r_3 (w.r.t. r_1) are shown in Figure 2. Note that the defined fractional extra cost is an upper bound on the true fractional extra cost, as there may be a shorter route than to serve the requests in the order assumed by the fractional extra cost calculation, i.e., $r_i.l_o \rightarrow r_i.l_o \rightarrow r_i.l_d \rightarrow r_i.l_d$.
3. Consider the best, i.e., lowest cost / highest savings, K-sized vehicle-share for an expiring request $r_i \in R_x$ to be composed of the first K requests with lowest FEC for r_i. This heuristic assumes that pair-wise fractional extra costs are additive.
4. Estimate the Amortized Cost (AC) of a vehicle-share s (w.r.t. r_i) as the normalized cumulative sum of FECs as

$$AC(r_i,s)=\frac{1+\sum_{r_j\in s}FEC(r_i,r_j)}{|s|}.$$

This heuristic assumes that there exists an optimal pick-up and drop-off sequence for requests in *s*, such that the cost of this sequence $m(s, d(.,.)) \leq AC(r_i, s) \cdot d(r_i.l_o, r_i.l_d)$.

5. Greedily group the best maximum *K*-sized vehicle-share that has the minimum amortized cost over all expiring trip requests. This heuristic is greedy because it possibly assigns a not-yet-expiring request r_j to a vehicle share of an expiring request, without considering what the current or even future best vehicle-share would be for r_j.
6. Remove requests that are part of the best vehicle-share from further consideration.
7. Repeat steps 2 through 7 as long as the best vehicle-share meets the minimum savings requirement.
8. Assign remaining trip requests to their own (single person) "vehicle-shares".

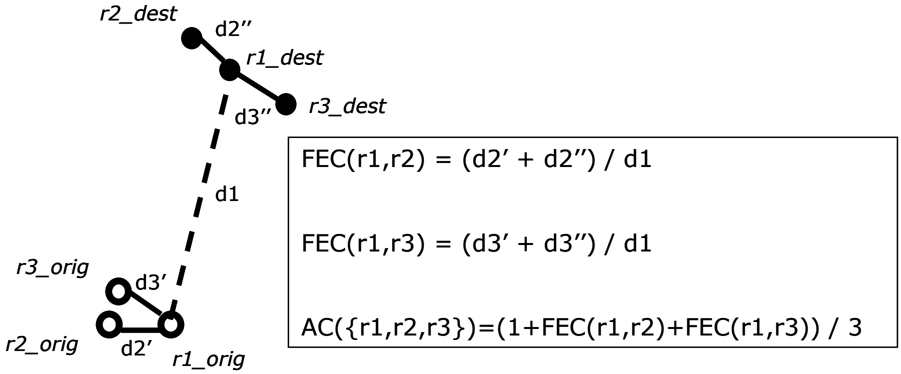

r2_dest
d2″
r1_dest
d3″
r3_dest
d1
r3_orig
d3′
r2_orig
d2′
r1_orig
FEC(r1,r2) = (d2′ + d2″) / d1
FEC(r1,r3) = (d3′ + d3″) / d1
AC({r1,r2,r3})=(1+FEC(r1,r2)+FEC(r1,r3)) / 3

Figure 2. Illustration of fractional extra cost (FEC) and amortized cost (AC) calculations w.r.t. request *r1*.

Even though the TG algorithm is based on heuristics, estimations and assumptions, in [12], it has been found to effectively optimize the vehicle-sharing problem. Furthermore, while some assumptions about extra costs for vehicle-shares do not hold in all cases, the combination of the approximations and assumptions result in an estimated cost for the vehicle-shares that is higher than the true minimum cost if the optimal pick-up and drop-off sequence is considered.

3.4 Problems with Large-Scale CT Systems

Unfortunately, the TG algorithm cannot be naïvely applied to facilitate a large-scale CT system, such as a ride-sharing system. The TG algorithm needs to calculate the pairwise fractional extra costs between expiring requests and all requests in the queue, entailing on the order of $O(n^2)$ cost calculations. In [12] a simple but effective implementation of the TG algorithm was able to handle loads of up to 50,000 requests per day, during which at peak traffic hours the number of requests within 10 minutes was at most 2,500. However, as input sizes increase, the execution times of any serial implementation of the TG algorithm will reach a point where continuous grouping is not possible. Then, the algorithm is not able to find nearly optimal groups for all the expiring request before they actually expire. This is demonstrated in Figure 3, where a load of 250,000 requests with common wait times of 10 minutes are grouped minute-by-minute using a highly efficient implementation of the TG algorithm. This implementation of the TG algorithm is able to keep up with the request flow most of the time, but when the number of pending requests exceeds about 5,200 (during rush hour), it is not able to find groups for the expiring requests within the allowed execution time of 60 seconds. In the example the grouping cycle time of the TG algorithm is 60 seconds, i.e. the algorithm is responsible for grouping the request that will expire within the next 60 seconds. Altering this grouping cycle time does not eliminate the problems of the algorithm in the case of large input sizes. Figure 3 also reveals that the computational complexity of the implementation of the TG algorithm is $O(n^3)$. This is due to the fact that, as described by the third heuristic in Section 3.3, the best K-sized vehicle-share is composed of the first K requests with lowest FEC for an expiring request. This necessitates a linear-time top-K selection for each expiring request, making the algorithmic complexity of the TG algorithm $O(n^3)$. Consequently, the above described scalability problems severely limit the applicability of the TG algorithm in a large-scale CT system.

3.5 Ride-Sharing Application Requirements

Ride-sharing is a type of vehicle-sharing where private vehicles are used as transportation. This fact represents additional requirements on solution to the general trip-sharing problem. In the context of ride-sharing there are *ride-requests* and *ride-offers*. Ride-requests are synonymous to trip requests both in form and semantics, with the exception that ride-requests do not necessarily have to be served. Ride-offers have at least three important attributes in addition to the attributes of a trip request. The first attribute specifies whether the offering person is willing to leave his/her vehicle behind. A person offering a ride with willingness of leaving his/her vehicle behind is either willing to take alternate modes of transportation or relies on the efficient operation of the ride-sharing system for future trips until he/she returns to his/her vehicle. A person not willing to leave his her vehicle behind values or needs his/her independence throughout the day. The second attribute specifies a *maximum relative extra cost* the offering person is prepared to incur. Finally, the third attribute specifies the *maximum number of additional passengers* the offering person's vehicle can accommodate.

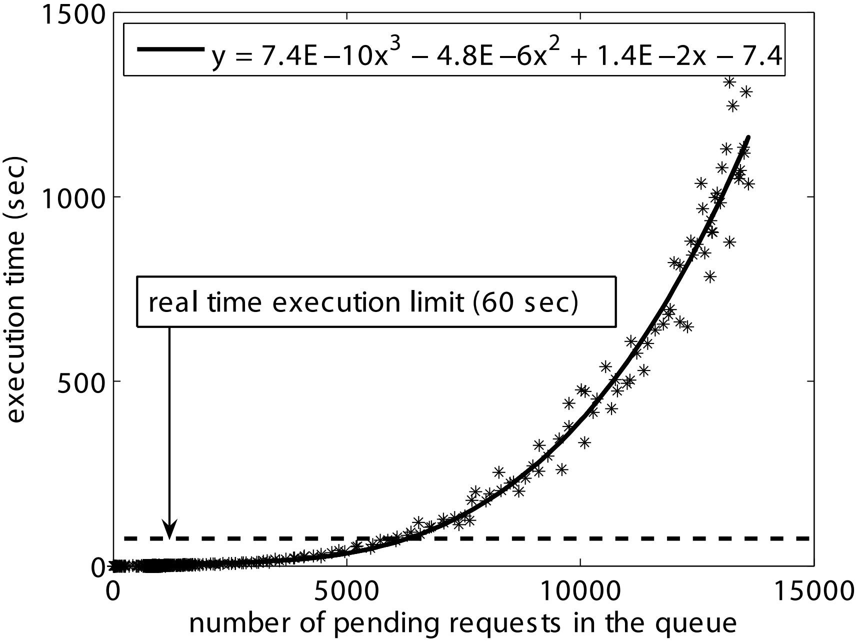


Figure 3. Scalability problems of the general trip grouping algorithm.

3.6 Application of the TG Algorithm in a RSS

It is clear that the TG algorithm cannot be applied in its current form for a ride-sharing application. However, a few simple modifications can make it applicable. First, in the context of ride-sharing, the ride offering person would like to leave as soon as the best vehicle-share that can be constructed meets the maximum relative extra cost requirements of the ride-offer. Hence, it makes sense to prioritize the order of greedy grouping based on the time the ride-offers have been present in the system. Second, because maximum relative extra cost requirements are defined by ride-offers individually, in every grouping cycle (execution of the TG algorithm) the best vehicle-share for all ride-offers needs to be considered. Third, every vehicle-share needs to fulfill the following two conditions: 1) it can contain only one ride-offer where the offering person is not willing to leave his/her vehicle behind, and 2) it has to contain at least one ride-offer of any type. To fulfill the above conditions it is enough to distinguish between two different sets: 1) the set of ride-offers of either type $\{R_o^{\bar{o}} \cup R_o^{o}\}$, and 2) the joint set of ride-request and ride-offers where the offering person is willing to leave his / her vehicle behind $\{R_r \cup R_o^{o}\}$. Associating these sets to sets used by the TG algorithm as $R_x = \{R_o^{\bar{o}} \cup R_o^{o}\}$ and $R_q = \{R_r \cup R_o^{o}\}$, the vehicle shares constructed by the TG algorithm fulfill the above two conditions.

Obviously, the modifications to the TG algorithm that are necessary to facilitate the proposed RSS are straight-forward. However, to preserve clarity in representation, the remainder of the paper considers only the implementation of a highly scalable TG algorithm.

4. Highly scalable trip grouping

Although the TG algorithm can be modified to meet the unique requirements of the proposed RSS, as it was demonstrated in Section 3.4, the algorithm in its present form does not scale with the input size and hence cannot be applied in large scale CT systems, such as the proposed RSS. This section describes a parallel implementation of the TG algorithm in the SCSQ Data Stream Management System.

Queries and procedures in SCSQ [23] (pronounced *sis-queue*) are specified in the query language SCSQL [24] (pronounced *sis-kel*). SCSQL is similar to SQL, but is extended with streams as first-class objects. SCSQ also features a main memory database. This database is used to keep the trip requests that are waiting, along with statistics about the data distributions. The waiting requests are processed by the TG algorithm and the statistics are used by the partitioners.

Details of the implementations are organized as follows. Section 4.1 describes how the trip grouping algorithm is implemented as a stored procedure in SCSQL. Section 4.2 outlines how SCSQ allows parallelization of the continuous stream query implementation of the TG algorithm. Section 4.3 describes four spatial partitioning methods that are used to partition the stream of trip requests into sub-streams for parallelization purposes.

4.1 Processing of a Request Stream

The TG algorithm is expressed as a procedure in SCSQL, which is listed below.

```
create function tg(vector input_window,
                   integer K, real min_savings,
                   integer wait_time)->vector
as begin
  declare vector ex, vector bcss, timeval ct;
  insert_q(in(input_window));
  set ct = get_end(input_window);
  set ex = select_ex_q(curr_time, wait_time);
```

```
 (9)     set bcss = {};
(10)     for each vector r where r = in(ex)
(11)     begin
(12)       remove_q(r);
(13)       set s = select subvector(ac,0,i)
(14)               from vector fec, vector ac,
(15)                    integer i, integer k
(16)               where fec = topk(calc_FEC(r),2,K)
(17)               and ac = calc_AC(fec,2)
(18)               and i = min(ac,2);
(19)       if savings(s) >= min_savings
(20)       begin
(21)         set bcss = concat(bcss,members(s));
(22)         remove_q(members(s));
(23)       end;
(24)       else
(25)         set bcss = concat(bcss,r);
(26)     end;
(27)     result bcss;
(28)   end;
```

The tg procedure takes an input window of the most recently arrived trip requests, and the three algorithm parameters K, min_savings, and wait_time. The output of tg is a vector of best vehicle-shares, bcss. tg executes as follows. First, on line 6, all requests in input_window are added to the main memory table of waiting requests q. Then, on line 7, based on the wait time parameter and the current time ct (indicated by the end of the input window), expiring requests, ex, are selected from q. The for each loop on line 10 iterates over each request r in ex as follows. On line 12, the request r is removed from the q. Then, in a compound query on lines 13–18, the best, maximum K-sized vehicle-share for r is found. The first part of the compound query, on line 16, calculates the fractional extra costs calc_FEC(r)=<r,ri,fec> between r and all other requests in q, and selects the tuples for the K requests with the lowest fractional extra costs. The remaining parts of the compound query, on lines 17–18, calculates the amortized costs calc AC(fec)=<r,ri,ac> based on the top-K fractional extra costs, and selects the lowest of these costs. The best vehicle-share that corresponds to this lowest amortized cost is assigned to s on line 13. Finally, if the savings of s is greater than equal to min savings, then the members of s are added to the best vehicle-shares, bcss (line 21), and are removed from q (line 22). Otherwise, r could not share its trip, and will be the only one in its vehicle-share (line 25). The implementations of the derived functions insert_q, get_end, select_ex_q, remove_q, subvector, calc_FEC, savings, and members are omitted to preserve brevity. For efficiency reasons, core functions that need to iterate over a set, such as `topk` and `calc_AC` are implemented as foreign functions in Lisp. Foreign functions allow subroutines defined in C/C++, Lisp, or Java to be called from SCSQL queries. The implementation of these functions is also omitted.

4.2 Parallel Stream Processing in SCSQ

Apart from streams, SCSQL includes Stream Processes (SPs) as first-class objects in queries. SPs allows dynamic parallelization of continuous queries, which is used in this paper to divide the incoming trip requests. The user associates subqueries with SPs. Massively parallel computations are defined in terms of sets of parallel subqueries, executing on sets of SPs.

The output of an SP is sent to one or more other SPs, which are called *subscribers* of that SP. The user can control which tuples are sent to which subscriber using a *postfilter*. The postfilter is expressed in SCSQL, and can be any function that operates on the output stream of its SP. For each output tuple from the SP, the postfilter is called once per subscriber. Hence, the postfilter can transform and filter the output of an SP to determine whether a tuple should be sent to a subscriber. Postfilters are used in the experiments to partition the input stream between the SPs that are carrying out TG.

The divide-and-conquer experiments are expressed as queries in SCSQL. All these queries have the same communication pattern between SPs, as shown in Figure 4. A Partition SP reads a stream of incoming trip requests (S1). That stream is partitioned into partial streams, which are sent to the Compute SPs. Each Compute SP executes the `tg` procedure on its partial stream. Also, each Compute SP evaluates the savings achieved, by comparing the total cost of all trips with the total cost of the shared trips. The results of all Compute SPs are merged together by a Combine SP. The resulting stream of cab requests (S2) is sent to the user.

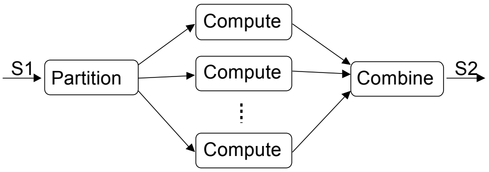


Figure 4. Communication pattern of TGs working in parallel.

4.3 Spatial Partitioning Methods

Section 3.4 showed that the TG algorithm does not scale well enough for large-scale CT systems. The key idea to overcome the scaling issue is a divide-and-conquer approach. Each request r_i = <t_r, l_o, l_d, t_e> are characterized by its origin and destination locations, $l_o \in \boldsymbol{R}^2$ and $l_d \in \boldsymbol{R}^2$. Hence, a request can be geographically characterized by a point in $l_o \times l_d$. In other words, a request is characterized by a point in $\boldsymbol{R}^4$. The divide-and-conquer approach is to partition this space and assign each partition to one TG. Intuitively, this approach will gain in execution time since each TG algorithm has less workload, but will lose some of the vehicle-sharing opportunities since none of the partitions are able to probe all combinations that a serial implementation can do. The goal is to find a partitioner that executes efficiently and achieves maximum savings. The following partitioning strategies are implemented in SCSQL and investigated experimentally.

4.3.1 Baseline Queries

Two baseline queries are executed; the unpartitioned query and the round-robin query. These queries form a performance baseline of the best and worst possible savings and execution speeds. All other methods should be compared to the measurements of these two queries.

The *unpartitioned* query applies a single TG algorithm on the entire request stream without any partitioning. Since all requests are going to a single TG, all possible sharing opportunities will be investigated. The unpartitioned query will give the best savings, but it will also take the longest time to execute because all burden will be placed on a single node. The unpartitioned query is expressed in SCSQL as follows:

```
select tg(v, 4, 0.8, 600, 60)
from vector v, charstring file
where v = twinagg(streamfile(file), 60.0, 60.0)
and file in
{"L16.dat","L8.dat","L4.dat","L2.dat","L1.dat"};
```

The `streamfile(file)` function reads tuples that are stored in `file`, and streams them out. The `twinagg(inputstream, size, stride)` function is taking a stream as the first argument and emits a time window over the last `size` seconds, every `stride` seconds. Hence, if `size=stride`, twinagg emits tumbling (consecutive and non-overlapping) windows of the input stream. This `twinagg()` makes sure that `tg()` always will get one minute worth of requests each time. Hence, `tg()` will get called once per minute. If no requests have arrived during a certain minute, `twinagg()` will emit an empty window for that minute. `tg(input_window, K, min_savings, wait_time)` performs the trip grouping algorithm. The query is executing once per file in the collection of filenames given on the last line of the query.

The *round-robin* partitioner will send the first request to one working SP. The next request will be sent to another working SP, and so on. This way, each SP will be given exactly 1/*n* of the total load, so the load balance is perfect. Since the round-robin partitioning scheme is perfectly load balanced, it will achieve the maximum possible execution speed. On the other hand, a TG algorithm executing in an SP that is operating on a round-robin data partition can be expected to give inferior savings since nearby requests not necessarily go to the same TG. Thus, the round-robin partitioner is expected to achieve the least savings. It is expressed in SCSQL as:

```
select merge(b)
from bag of sp b, sp c, integer n, charstring file
where b = spv(select streamof(tg(twinagg(stract(c),
              60.0, 60.0), 4, 0.8, 600)))
              from integer i where i=iota(1,n))
and c = sp(winagg(streamfile(file), n, n), n, 'rr')
and n in {16, 8, 4, 2}
and file in
{"L16.dat", "L8.dat", "L4.dat", "L2.dat", "L1.dat"};
```

In this query, the output of `streamfile` is passed into `winagg(input_stream, size, stride)`, which is forming tumbling windows of size `n`, `n` being the number of subscribers to the Partition SP `c`. Each window is an ordered set of tuples, so it is represented as a vector. The round-robin function `rr`, is applied once per subscriber. For subscriber *i*, `rr` picks up the *i*-th element in the vector emitted from `winagg`. The `SP(stream, nsubscribers, postfilter)` is assigning `stream` and `postfilter` to a new SP, which should expect `n` subscribers. Thus, a combination of a `winagg` on a stream and a vector dereference in the postfilter function results in a round-robin partitioner.

`iota(m,n)` generates all integers from `m` to `n`. Hence, the query in the call to `spv(bag of stream)` creates `n` duplicates of the query `streamof(tg(twinagg(stract(c),60.0, 60.0), 4, 0.8, 600))`, where `stract(c)` is extracting the stream from stream process `c`. Each one of these queries will be assigned to a stream process. Finally, the output of all the stream processes in `b` will be merged. Refer to Figure 4 for a graphical representation of the communication pattern: The partition is done at SP `c`, compute is performed by the SPs in `b`, and the combination is done in the `merge` at top level.

4.3.2 Static Point Quad Partitioning

Static point quad partitioning (SPQ) calculates from historical data the medians of each dimension of the trip requests. Each dimension of the four-dimensional trip request data space is split once along the median of each dimension. Figure 5(a) shows the SPQ partitions for some data points in two dimensions. By splitting each dimension once, SPQ par titions the four-dimensional trip request data space into 16 regions. One or more regions can be assigned to one SP, executing a TG algorithm for that region. This SCSQL query executes SPQ:

```
select merge(b)
from bag of sp b, sp c, integer n, charstring file
where b = spv(select streamof(tg(twinagg(stract(c),
              60.0, 60.0), 4, 0.8, 600)))
              from integer i where i=iota(1,n))
and c = sp(streamfile(file),n,'pq')
and n in {16,8,4,2}
and file in
{"L16.dat","L8.dat","L4.dat","L2.dat","L1.dat"};
```

The difference between this query and the round-robin query above is only in the call to the partitioning SP `c`. Instead of applying postfilter function `rr` on a window, SP `c` is applying the `pq` postfilter on the tuples from `streamfile`. For each tuple, `pq` decides which subscriber it should go to.

4.3.3 Static KD Partitioning

Static KD partitioning (SKD) splits trip request data in a hierarchical fashion by processing dimensions one after the other as follows. For a given dimension, SKD first calculates the *local* median for that dimension, and then splits the local trip request data for the dimension based on the median into approximately equal sized subsets. Figure 5(b) shows the SKD partitions for some data points in two dimensions. The data is first split around the median of the horizontal dimension, then the data in each of the so obtained partitions is further split around the local (horizontal) median of each of the partitions. By splitting once per dimension, the KD also partitions the four-dimensional trip request data space into 16 regions. The SCSQL query that executes SKD differs from that of SPQ in that it applies another postfilter function at the partitioning SP, namely `kd` instead of `pq`. Since the difference is so small, the SCSQL query is not shown here.

4.3.4 Adaptive Point Quad Partitioning

The trip request data distribution changes over time. During the morning rush hours people want to move from their homes (residential district) to their work (business and industrial districts). During the evening rush hours the opposite is true. The trip requests that correspond to the morning rush hour movements are likely to fall in different partitions than the trip requests that correspond to the evening rush hour movements. Consequently, the "morning rush hour" partitions will be densely populated in the morning-hours, and the “evening rush hour” partitions will be densely populated in the evening hours. Clearly, a static partitioning method does not consider these temporal changes in data distribution and is therefore likely to result in temporarily unbalanced partitions.

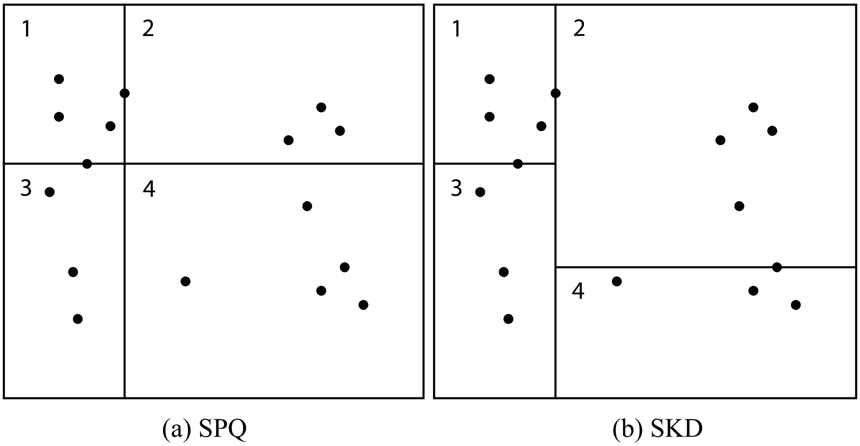


Figure 5. Illustrations of the static partitioning methods.

The adaptive point quad partitioning (APQ) adjusts the boundaries of the partitions periodically, based on statistics obtained from a recent history buffer of the trip request stream, and distributes the newly arriving trip requests according the newly adjusted partitions. Figure 6(a) shows two consecutive partitionings that are constructed by the APQ partitioning for some data points in two dimensions. Hollow dots represent data points that were present when the previous partitioning was constructed, but are not present or are not relevant for the construction of the current partitioning. In contrast, solid rectangular markers represent data points that were not present when the previous partitioning was constructed, but are relevant for the construction of the current partitioning. Solid and dashed lines represent current and previous partition boundaries. The following SCSQL query executes TG algorithm with APQ:

```
select merge(b)
from bag of sp b, sp c, integer n, charstring file
where b = spv(select streamof(tg(twinagg(stract(c),
              60.0, 60.0), 4, 0.8, 600)))
              from integer i where i=iota(1,n))
and c = sp(pqstat(streamfile(file),
           600.0, 60.0, 10),n,'pq')
and n in {16,8,4,2}
and file in
```

```
        {"L16.dat","L8.dat","L4.dat","L2.dat","L1.dat"};
```

This query differs from the SPQ query in the call to the partitioning SP `c`. The `streamfile` function is wrapped by `pqstat(inputstream, size, stride, samplefreq)`. This function emits the same stream as its input stream, and maintains statistics in a main memory table of SCSQ. Every `stride×samplefreq` seconds, `pqstat` computes medians in each dimension of $l_o \times l_d$ across the tuples seen in the last `size` seconds. These median values are then used in the `pq` postfilter. This way, the partitioning decisions are always done on recent data.

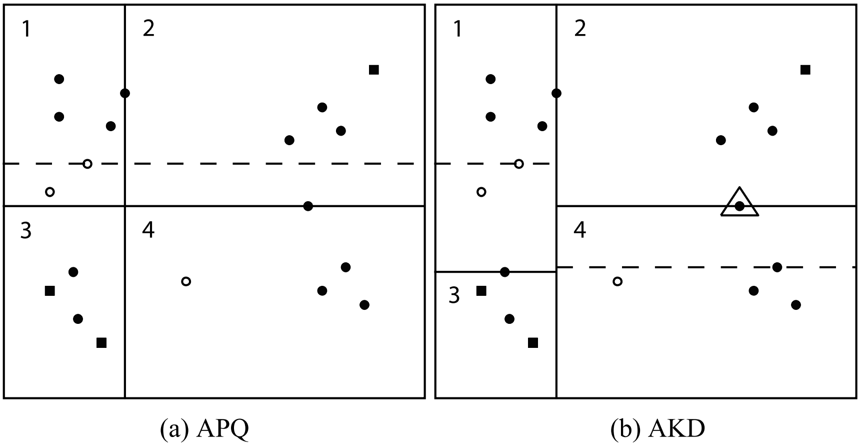
4.3.5 Adaptive KD Partitioning

The adaptive KD partitioning (AKD) adjusts the boundaries of the partitions periodically, based on statistics obtained from a recent history buffer of the trip request stream, and distributes the newly arriving trip requests according the newly adjusted partitions. Figure 6(b) shows two consecutive partitionings that are constructed by the AKD partitioning for some data points in two dimensions. The semantics of the symbols used in the figure are the same as in the case of the APQ partitioning. However, Figure 6(b) depicts a situation that can happen in either one of the adaptive spatial partitioning methods. Consider the data point inside the triangle. Since it was present when the previous partition was constructed it has been assigned to compute node 2 for processing. According to the newly constructed partitions however, it should be assigned to compute node 4. To avoid communication between compute nodes, the following design choice is made: once a data point is assigned to a partition (compute node), it is never reassigned to another partition, even if the newly adjusted partitions would suggest this.

The SCSQL query that executes SKD differs from SPQ in that it applies another statistics wrapper function and another postfilter function at the partitioning SP, namely `kdstat` instead of `pqstat` and `kd` instead of `pq`. `kdstat` works analogously to `pqstat` with the difference that it maintains dynamical versions of local dimension splits of the kind that SKD has. Since the difference between this query and the APQ query is so small, the SCSQL AKD query is not shown here.

5. Density-based spatial stream partitioning

In all spatial partitioning methods proposed in this paper, the space of requests is split by planes. The locations of the splitting planes are determined by the medians of request data. These splitting planes potentially eliminate the discovery of good shares, when members of the good shares are on different sides of a splitting plane. This naturally leads to some degradation in the overall grouping. The degradation is larger when the planes are cutting through denser regions of the request space with many sharing opportunities, than when the planes are cutting through sparser regions of the request space. Since neither of the proposed partitioning methods consider the distribution density of the requests, the degradation of grouping quality due to boundary effects can be expected to be approximately the same for all four partitioning methods. However, as Section 6 demonstrates, this degradation is rather small.



(a) APQ (b) AKD

Figure 6. Illustrations of the dynamic partitioning methods.

No matter how small the degradation is, simple spatial partitioning methods that take into account the density of the data could reduce the degradation. The objective of such a density-based partitioning is to determine the positions of the splitting planes so that they pass through regions where data is sparse. To achieve this, a simple but effective clustering method [9] can be used to find local minima in the multimodal data distributions along each dimension, and place splitting planes at those locations. Figure 7 shows the distributions for each dimension of the request data during morning peak hours and off-peak hours. During the morning peak hours, there does not seem to be any regions where the request data is very sparse. However, during off-peak hours, when people who are not working are most likely to be in one of the larger shopping malls, the distributions of the destination dimensions (*tx*, *ty*) are clearly multimodal. In this later situation, ensuring that splitting planes are chosen correctly at local minima would minimize the boundary effects. However, since most of the requests are during peak hours, the overall average grouping achieved by the parallel TG algorithm would not be substantially improved.

Since the local minima are likely not to be at the median values of the dimensions, there exists a trade-off between equal-sized partitions and partitions with minimal boundary effects. A dual-objective partitioning that takes this trade-off into consideration could weigh the expected degradation against the imbalance between the created partitions. Although the implementation of the density-based and the dual-objective spatial stream partitioning methods is straight-forward, it is left for future research.

The proposed spatial stream partitioning methods are devised to scale the TG algorithm to very large flows of requests. However, they can be considered as a general approach to make computationally intensive spatial analysis tasks scalable through parallelization. For example, the density-based and the dual-objective spatial stream partitioning methods can be applied to speed up spatial clustering of streams, spatio-temporal rule mining [10], or the processing of high-resolution image streams.

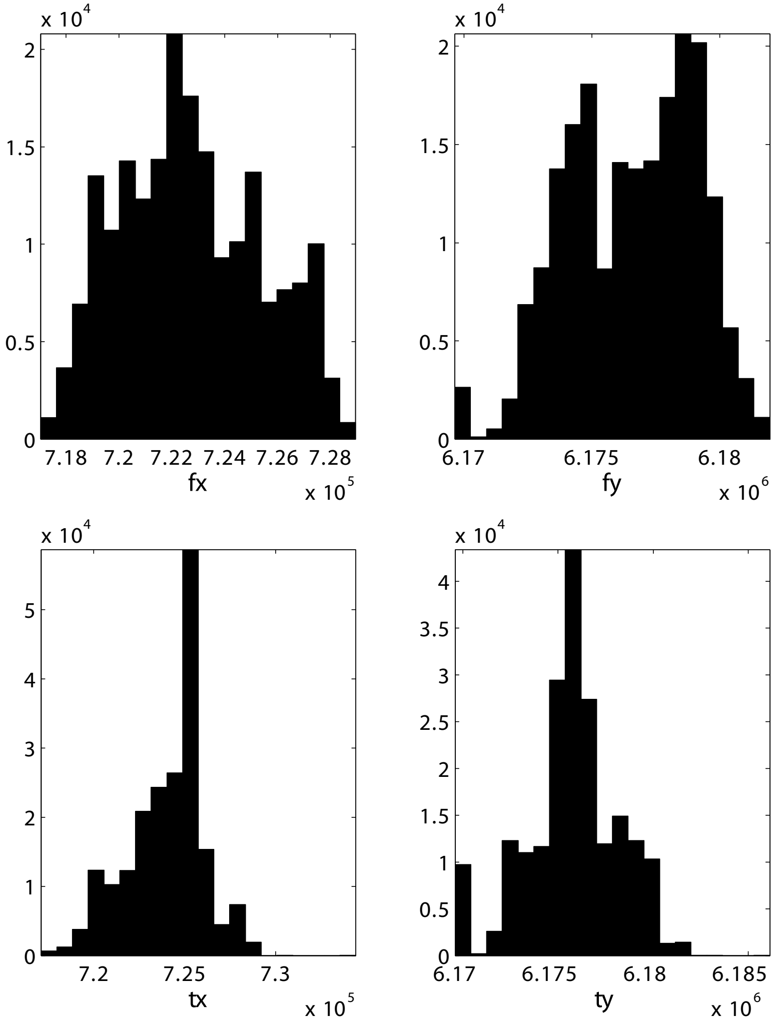
6. Experiments

The parallel implementations of the TG algorithm were tested on a cluster of Intel® Pentium® 4 CPU 2.80GHz PCs. Each SP in the query language started a running process (RP) on a separate node in the cluster. TCP/IP over Fast Ethernet was used to carry streams between the nodes.

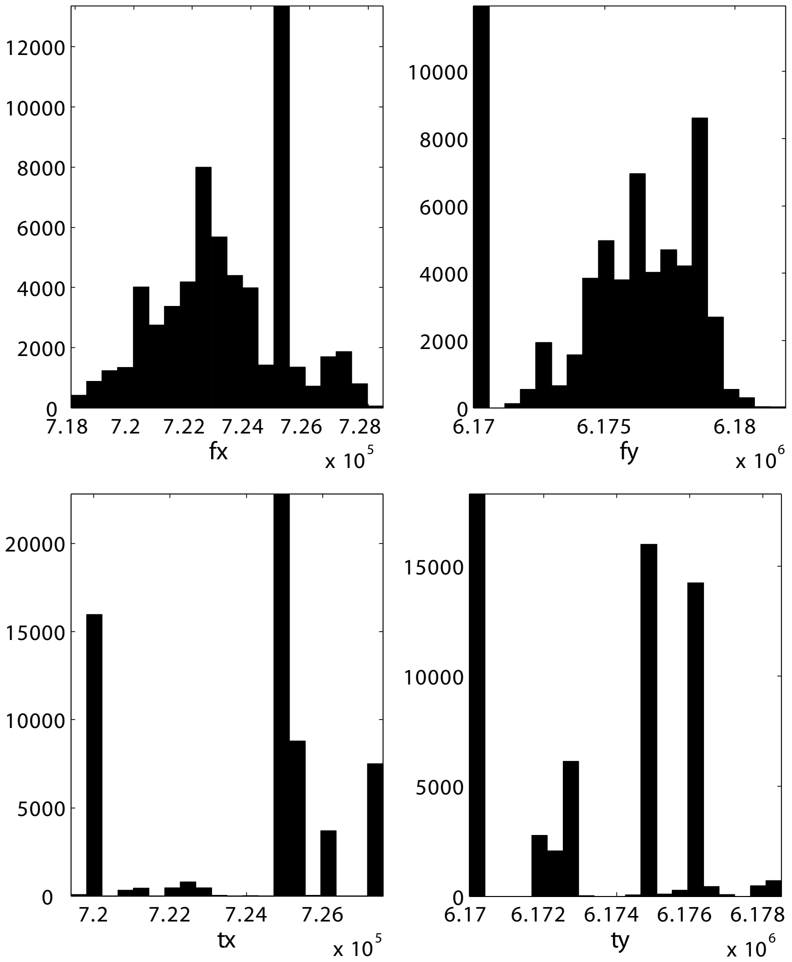
Trip request data was simulated using ST-ACTS, a spatio-temporal activity simulator [11]. Based on a number of real world data sources, ST-ACTS simulates realistic trips of approximately 600,000 individuals in the city of Copenhagen, Denmark. For the course of a workday, out of the approximately 1.55 million generated trips, approximately 251,000 trips of at least 3-kilometer length were selected and considered as trip requests. To test the scalability of each of the parallel implementations using the four spatial stream partitioning methods, decreasing sized subsets of the total load of 251,000 trip requests were constructed by only considering every second, fourth, eighth and sixteenth trip request in the input stream. These subsets are referred to as 1/2, 1/4, 1/8, 1/16 load, respectively.

To evaluate the effectiveness of the four spatial stream partitioning methods, for the purposes of parallelization of the TG algorithm, two measures were used: (overall) execution time and average savings achieved by the grouping (also referred to as the quality of the grouping or quality for short).

The reported savings for each vehicle-share are based on amortized costs, which has been shown to overestimate the true cost of a vehicle-share that considers the optimal pick-up and drop-off sequence of requests. Hence, the reported savings underestimate the true savings. Nonetheless, the reported savings can be used as an unbiased measure for the quality of the grouping.



(a) Morning peak hours



(b) Off-peak hours

Figure 7. Request data distributions along each dimension. "f" and "t" stand for "from" and "to", respectively. Hence, fx and fy are request origin dimensions, while tx and ty are request destination dimensions.

For each of the partitioning methods an extensive set of experiments were performed for fixed algorithm parameters ($K = 4$, `min_saving` = 0.2, and $\Delta t = 10$ minutes) under varying loads using degrees of parallelization. The adaptive partitioning methods updated the partitions every 10 minutes based on the trip request that arrived in the last 10 minutes.

6.1 Baseline Performance

To establish a point of reference for the performance measures the baseline queries specified in Section 4.3.1 were executed. Table 1 shows the results for the unpartitioned query. Savings obtained by the unpartitioned query (serial execution) are *considered* to be optimal, while running times are *considered* to be worst case performance. Note that these measures are "optimal" and "worst case" with respect to the TG algorithm. Moreover, as it is demonstrated in Section 3.3, due to the computational complexity of the *vehicle-sharing problem*, the calculation of a truly optimal grouping, even in the case of a few requests, is infeasible. Due to the large difference in scale between serial and parallel execution times, serial execution times are not shown in subsequent figures. Savings achieved by the unpartitioned query (serial execution) are also not shown in subsequent figures, but are used to report relative performance of the parallel executions in terms of savings and quality.

Table 1. *Performance of the serial TG algorithm.*

load	execution time (sec)	savings
0.06125	28.8	0.325
0.125	120.1	0.388
0.25	702.9	0.445
0.5	16343.5	0.491
1	69771.6	0.530

In comparison, the round-robin query was executed to obtain optimal execution times due to perfect load balancing and worst case savings due to the distribution independent partitioning of requests between SPs. The results of these experiments are shown in Figures 8 and 9 as RR, however it is emphasized that RR is not one of the proposed spatial stream partitioning methods, but is *only* used as a reference.

6.2 Absolute Performance of the Parallel TG Algorithms

Figures 8 and 9 show the absolute performance of the parallel TG algorithm for varying load and degrees of parallelization using different spatial stream partitioning methods. From Figure 8(a) it can be seen that the execution times of all of the methods decrease as the parallelism is increased. Figure 8(a) also reveals that the adaptive versions of the spatial partitioning methods adjust well to the changing spatial distribution of the requests, resulting in more balanced partitions and ultimately faster execution times when compared to their static version. The improvement in execution time due to adaptive partitioning is most evident for the SPQ partitioning. Figure 8(b) shows that while the execution time of the TG algorithms can be scaled, the underlying algorithmic complexity of the TG algorithm executed on the compute nodes does not change. The effect of the underlying algorithmic complexity is more observable for spatial partitioning methods that construct less balanced partitionings, in particular SPQ.

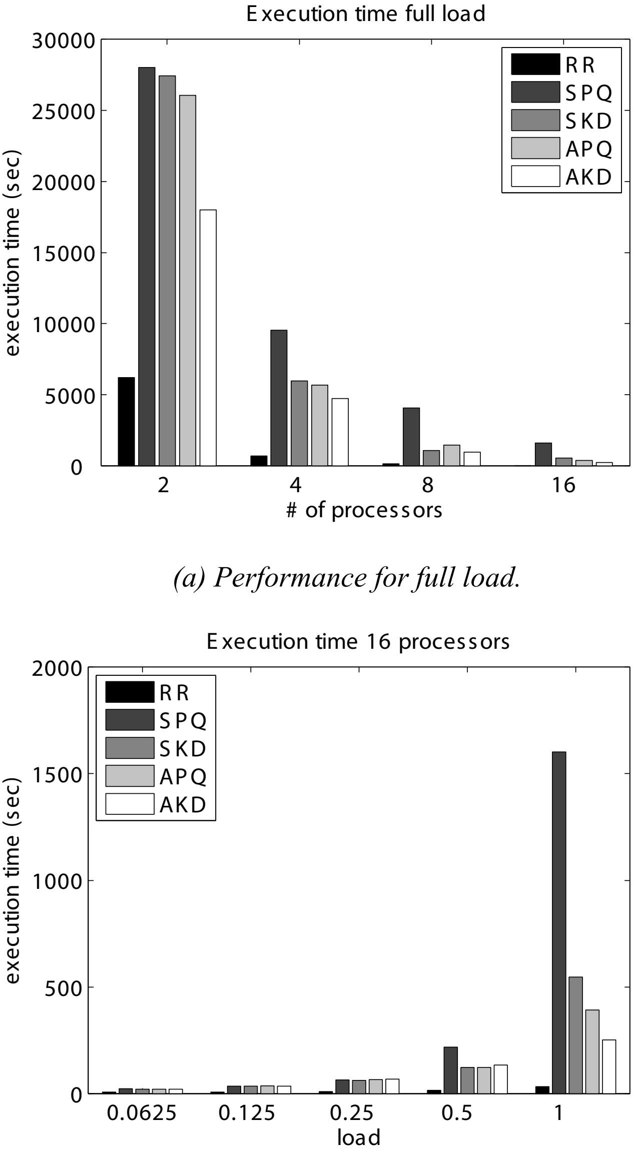
Figure 9(a) shows that in general the quality of the grouping decreases as the degree of parallelization is increased. However in the case of non-spatial partitioning (RR) this degradation is significant, while in the case of either one of the four spatial partitioning methods it is negligible. Figure 9(b) shows that the grouping quality increases as the load is increased. This is due to the simple fact that the spatio-temporal density of the trip requests increases. As a consequence, the likelihood that a request becomes part of a "good" vehicle share increases. The almost negligible differences between the qualities achieved by the four partitioning methods, as explained in Section 5, is due to the fact that since neither of the partitioning methods consider the data densities, but only the medians of the dimensions, the total degradation due to boundary effects is approximately the same for the four partitioning methods.

6.3 Relative Performance of the Parallel TG Algorithms

Figure 10 shows relative execution times of the parallel TG algorithms when compared to the optimal execution time that is achieved by RR partitioning due to perfect load balancing. With the exception of the SPQ partitioning, all other partitioning methods result in parallel execution times that are within the same order of magnitude as the optimal. There are potentially two sources for this slowdown: the cost of partitioning and the extended execution times due to improper load balancing. Since adaptive partitioning methods have to maintain a limited history of the stream and periodically recompute partition boundaries based on this history, they do additional work compared to their static counterparts. Figure 10 shows that execution times resulting from adaptive partitioning are significantly shorter than the execution times achieved by static partitioning. Hence, it is clear that the additional time needed to perform the spatially partitioned parallel queries can mainly be attributed to unbalanced partitions.

Finally, comparing the savings in Figure 9(b) to the savings in Table 1 reveals that the grouping quality achieved by either one of the partitioning methods is within the 95% of the optimal quality for the full load. Even if the load is decreased to 1/16 of the total load, all the spatial partitioning methods still achieve approximately 90% of the maximum possible savings.

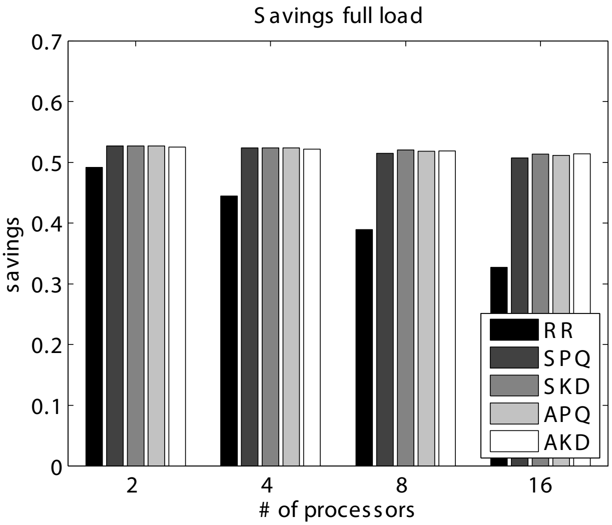
The experiments can be summarized as follows. First, RR partitioning has perfect load balance and is a very simple partitioning method, hence it has the fastest execution time. However, RR partitions the space badly and achieves a bad grouping quality. Second, using a spatial partitioning method improves grouping quality. All spatial partitioning methods achieve at least 95% of the maximum possible savings in the case of the full load. Third, the adaptive partitioning methods always execute faster than their static equivalents. That is because the adaptive methods constantly adapt the partitioning according to the last tuples observed, which will lead to better load balance.



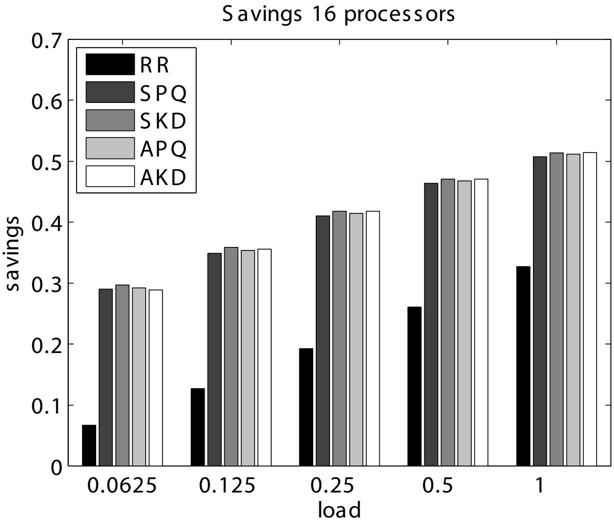
(a) Performance for full load.

(b) Performance for 16 processors.

Figure 8. Execution times for the parallel TG algorithm for different partitioning methods for varying parallelization and load.



(a) Quality for full load.



(b) Quality for 16 processors.

Figure 9. Savings for the parallel TG algorithm for different partitioning methods for varying parallelization and load.

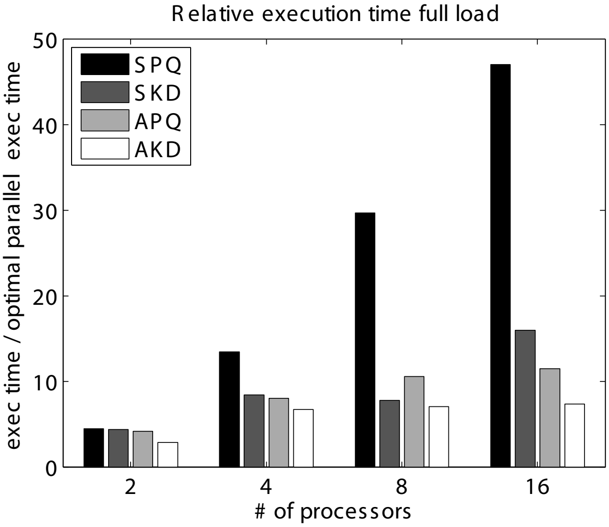


Figure 10. Relative performance for the parallel TG algorithm (compared to RR partitioning) for different partitioning methods for varying parallelization.

At the same time, the savings are approximately the same for both the static and adaptive partitionings. Adaptive partitioning is also preferred from an operational point of view, since it does not need any prior knowledge about the data distribution. Finally, since all partitioning methods (except RR) achieve about the same savings, the preferred method is the one with the fastest execution time of SPQ, SKD, APQ, and AKD. Thus, AKD is the best partitioning method.

7. Conclusions and future work

This paper proposed highly scalable algorithms for trip grouping to facilitate large-scale collective transportation systems. The algorithms are implemented using a parallel data stream management system, SCSQ. First, the basic trip grouping algorithm is expressed as a continuous stream query to allow for a very large flow of requests. Second, following the divide-and-conquer paradigm, four spatial stream partitioning methods are developed and implemented to divide the input request stream into sub-streams. Third, using the infrastructure of SCSQ and the partitioning methods, parallel implementations of the grouping algorithm are executed in a parallel computing environment. Extensive experimental results show that the parallel implementation using simple adaptive partitioning methods can achieve substantial speed-ups, without significantly affecting the quality of the grouping. As discussed in Section 5, spatial partitioning is not only appropriate for the given application, but it is applicable to parallelize computationally expensive spatial analysis tasks. As it was demonstrated, SCSQ can easily accommodate the parallel implementation of such tasks.

Future work will be along four paths. First, for the adaptive partitioning methods, the effects of keeping a longer history versus sampling more frequently will be investigated. Second, the density-based and dual-objective spatial stream partitioning methods will be implemented and their effectiveness evaluated. Third, the proposed partitioning methods, independent of the rate of flow, always construct a fixed number of partitions. While not substantially, but as the number of partitions increases the grouping quality decreases. Hence, an adaptive partitioning approach in which the number of partitions is increased / decreased depending on the rate of flow will be devised and tested. Finally, to preserve clarity the paper presented the generic TG algorithm in its simplest form. In particular, in the presented version all vehicles are assumed to have the same passenger capacity and all requests have a common minimum savings parameter. Furthermore, in-route grouping, i.e., assigning requests to already active but not fully-occupied vehicle-shares, is not handled by the simple version of the TG algorithm. Future work will consider the implementation of a more complex version of the TG algorithm that addresses the above issues.

Acknowledgements

This work was supported in part by ASTRON and the Danish Ministry of Science, Technology, and Innovation under grant number 61480.

8. References

9. J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM, (18)9:509–517, 1975.
10. J. L. Bentley and M. I. Shamos. Divide–and–Conquer in Multidimensional Space. In Proc. of ACM–STOC, pp. 220–230, 1976.
11. CARLOS Ride–Sharing System. http://www.carlos.ch/
12. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, et al. Scalable Distributed Stream Processing. In Proc. of CIDR, 2003.
13. T. G. Crainic, F. Malucelli, and M. Nonato. Flexible many–to–few + few–to–many = an almost personalized transit system. In Proc. of TRISTAN, pp. 435–440, 2001.
14. R. Finkel and J.L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. In Acta Informatica 4 (1), pp. 1–9, 1974.
15. M. Ivanova and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. In Proc. of VLDB, pp. 157–168, 2005.

16. G. A. Frank and D. F. Stanat. Parallel Architecture for k–d Trees. Technical report, North Carolina University at Chapel Hill Dept. of Computer Science, May 1988.
17. M. Gebski and R. K. Wong. A New Approach for Cluster Detection for Large Datasets with High Dimensionality. In Proc. of DaWaK, pp. 498–508, 2005.
18. G. Gidófalvi and T. B. Pedersen. Spatio–Temporal Rule Mining: Issues and Techniques. In Proc. of DaWaK, pp. 275–284, 2005.
19. G. Gidófalvi and T. B. Pedersen. ST–ACTS: A Spatio–Temporal Activity Simulator. In Proc. of ACM–GIS, pp. 155–162, 2006.
20. G. Gidófalvi and T. B. Pedersen. Cab–Sharing: An Effective, Door–to–Door, On–Demand Transportation Service. In Proc. of ITS, 2007.
21. A. Guttman. R–trees: a dynamic index structure for spatial searching. In Proc. of SIGMOD, pp. 47–57, 1984.
22. T. Hägerstrand. Space, time and human conditions. In Dynamic allocation of urban space, ed. A. Karlqvist et. al. Lexington: Saxon House Lexington Book, 1975.
23. Hitchstrers. http://www.hitchsters.com
24. C. S. Jensen, D. Pfoser, and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In Proc. of VLDB, pp. 395–406, 2000.
25. R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. StreamGlobe: Processing and Sharing Data Streams in Grid–based P2P Infrastructures. In Proc. of VLDB, pp. 1259–1262, 2005.
26. B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In Proc. of VLDB, pp. 1338–1341, 2005.
27. H. Samet. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, 2006.
28. Transportation Problems. http://www.di.unipi.it/optimize/transpo.html
29. Taxibus – Intelligent Group Transportation. http://www.taxibus.org.uk/index.html
30. texxi – Transit Exchange XXIst Century. http://www.texxi.com
31. E. Zeitler and T. Risch. Processing high–volume stream queries on a supercomputer. In Proc. of ICDEW, pp. 144, 2006.
32. E. Zeitler and T. Risch. Using stream queries to measure communication performance of a parallel computing environment. In Proc. of ICDCSW, pp. 65–74, 2007.
33. Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In Proc. of ICDE, pp. 791–802, 2005.

Paper IV

Scalable Splitting of Massive Data Streams

Erik Zeitler and Tore Risch
Department of Information Technology, Uppsala University, Sweden
{erik.zeitler, tore.risch}@it.uu.se

Abstract– Scalable execution of continuous queries over massive data streams often requires splitting input streams into parallel sub-streams over which query operators are executed in parallel. Automatic stream splitting is in general very difficult, as the optimal parallelization may depend on application semantics. To enable application specific stream splitting, we introduce splitstream functions where the user specifies non-procedural stream partitioning and replication. For high-volume streams, the stream splitting itself becomes a performance bottleneck. A cost model is introduced that estimates the performance of splitstream functions with respect to throughput and CPU usage. We implement parallel splitstream functions, and relate experimental results to cost model estimates. Based on the results, a splitstream function called autosplit is proposed, which scales well for high degrees of parallelism, and is robust for varying proportions of stream partitioning and replication. We show how user defined parallelization using autosplit provides substantially improved scalability (L = 64) over previously published results for the Linear Road Benchmark.

Keywords: Distributed stream systems, parallelization, query optimization.

1. Introduction

Data Stream Management Systems (DSMS) are becoming commonplace for a wide range of scientific and industrial applications, with high-volume data streams and queries that involve complex computations. Scalable execution in such applications requires parallelization. The parallelization of a query is called the *parallelization strategy*. In general, it is very difficult to automate the parallelization strategy, since the optimal parallelization may depend on application semantics. Our approach is to extend the query language with second-order functions to enable the user to specify non-procedural parallelization strategies. These functions split an input stream into large collections of parallel streams over which queries produce collections of result streams. Depending on the application, this collection of result streams can be merged, aggregated or further partitioned.

Splitstream functions partition and/or replicate input streams into a collection of streams. For each tuple in the input stream, splitstream decides whether the tuple should be sent to one specific DSMS node (partitioning) or many DSMS nodes (replication). Partitioning a stream is necessary when executing expensive queries. Replication is required, e.g., when aggregates are computed over data distributed over many local DSMS nodes. A splitstream function is compiled and optimized into a *splitstream plan*. We show how to automatically generate an optimized splitstream plan with high throughput and low CPU cost given a non-procedural splitstream specification. A generic splitstream function *autosplit* is defined that generates an optimized parallel splitstream plan based on a simple decision rule. To investigate the scalability of splitstream functions, we have parallelized an implementation of the Linear Road Benchmark (LRB), which is called *scsq-plr*. We focus on the performance bottleneck in the parallelization strategy of *scsq-plr*, which is splitting the stream of position reports and account balance queries. In summary, we present the following results:

- Splitstream functions are introduced, which enable non-procedural user defined specification of parallelization strategies.
- A cost model is introduced that estimates CPU utilization and throughput of splitstream plans.
- A theoretically optimal tree shaped splitstream plan is devised that has maximum throughput according to the cost model. This plan is compared with other splitstream plans.
- A generic splitstream function *autosplit* automatically generates tree shaped splitstream plans. *Autosplit* is shown to improve the scalability of LRB substantially.

2. Splitstream Functions

A *stream function*, *Q*(*S*, …) → *So* is a parameterized query that transforms one or more input stream arguments *S* into one or more output streams *So*. A *parallelization function* operates on collections of streams, and is used for specifying parallel executions of stream functions. Figure 1 illustrates three basic classes of parallelization functions; *splitstream*, *mapstream*, and *mergestream*. *splitstream* splits an input stream into two or more output streams. The number of output streams of a splitsteam is called its *width*. *mapstream* applies a stream function on each stream in a collection of streams, while *mergestream* merges or joins a collection of streams into a single output stream. Examples of mergestream functions are stream union and windowed stream join. Although all parallelization functions are used in the final evaluation experiment, the focus of this paper is to optimize splitstream functions since they are shown to be a performance bottleneck.

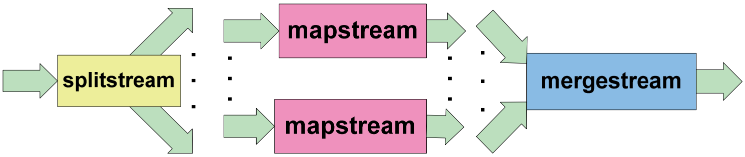


Figure 1. Splitstream, mapstreams, and mergestream.

A splitstream function has the basic signature *splitstream*(*stream s*, *integer w*, *function rfn*, *function bfn*) → *vector of stream sv*. The input stream *s* is split into w output streams in the vector *sv*. The first functional argument *rfn* is the routing function, having signature *rfn*(*object tpl*, *integer w*) → *integer*, which returns the output stream number (between 0 and *w* – 1) for each tuple that should be routed to a single output stream. The functional argument *bfn*(*object tpl*) → *boolean* is the broadcast function, which returns true for tuples to be broadcasted to all output streams. *bfn* and *rfn* return nil for tuples that should neither be broadcasted nor routed. *rfn* and *bfn* are defined declaratively in the query language by the user.

2.1 Parallelizing LRB

LRB [1] simulates a traffic system of expressways with variable tolling that depends on the utilization of the roads and the presence of accidents. Vehicles undertake journeys in the expressway system consisting of *L* expressways while emitting stream of position reports. An implementation must respond correctly to the continuous and historical queries of the benchmark within the allowed maximum response time (MRT). The number of expressways that an implementation is able to handle is called the L-rating of the implementation. An LRB implementation can be seen as a stream function *LR*(*S*) → *So*. The LRB input stream *S* consists of four kinds of tuples; *P*, *A*, *D*, and *E* (event type 0, 2, 3, and 4, respectively), of which 99% are position reports *P*. The rest of the tuples are account balance queries *A* (0.5%), daily expenditure queries *D* (0.1%), and estimated travel time queries *E* (0.4%). Currently, *E* tuples are ignored [1]. The *D* tuples are computed over historical data, their frequency is very low, and the allowed MRT is 10 sec, so any DBMS can respond to *D* tuples within the required time. Allowed MRT for *P* and *A* tuples are five seconds. Since these tuples are very frequent, they have to be processed efficiently. The input stream rate increases continuously during the 180 minutes of the simulation. The result stream *So* contains toll and accident alerts (event type 0 and 1), and query responses (event type 2 and 3). Some position reports do not result in toll alerts, so the rate of *So* is less than that of *S*.

Our single node LRB implementation *scsq-lr* [17] spent most of its CPU time computing segment statistics. This processing is local to each expressway, i.e., events on one expressway are independent of events on other expressways. Thus, the key to efficient parallelization is to partition the input stream into *L* parallel streams, and execute one instance of *LR()* for each expressway, as is employed in *scsq-plr*. The *A* tuples require account balance information. In *scsq-plr*, a local account table is maintained on each *LR()*, so that vehicles accumulate account balance locally on each expressway. Then, account balance queries must aggregate account data from all expressways. Therefore, all *A* tuples are broadcasted to all DSMS nodes running *LR()*.

Figure 2 illustrates this parallelization strategy for $L = 4$. The input stream is first split by *splitstream*$_D$, whose routing function *rfnD*(*e*, *w*) is defined as:

```
create function rfnD(Event e, Integer w) → Integer as
select i from integer i where
(eventtype(e)<3 and i=0) or (eventtype(e)=3 and i=1);
```

In *splitstream*$_X$, the body of *rfnX*(*e*, *w*) is `select expressway(e) where eventtype(e)=0`, while *bfnX*(*e*) is defined as `select eventtype(e)=2`. Each stream from *splitstream*$_X$ is processed by an *lr* node (executing *LR()*), whose result stream is split by *splitstream*$_O$ using *rfnO*(*e*, *w*) defined as `select eventtype(e)`. *splitstream*$_O$ and *splitstream*$_D$ do not broadcast, so they have no *bfn*. All toll and accident alert result streams are merged with union-all. Account balance answers from each *splitstream*$_O$ are joined on query id and added together.

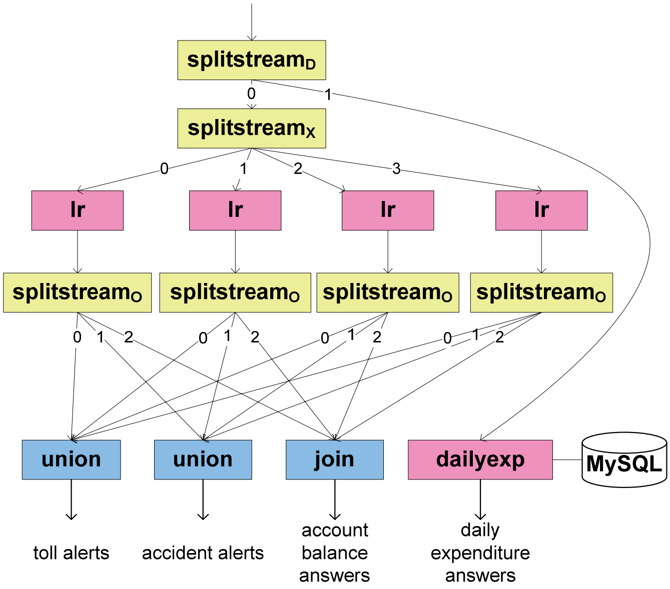


Figure 2. The parallelization strategy in scsq-plr. L = 4.

2.2 Single Process Splitstream

A splitstream function is naïvely implemented by a single process splitstream operator *fsplit*, its modules being shown in Figure 3. The input stream S is read and de-marshalled by the *consume* module. In the *process* module, *rfn* and *bfn* are called for each tuple. Each *emit* module marshals and emits tuples to its output stream So_i, $i = 0\dots w{-}1$.

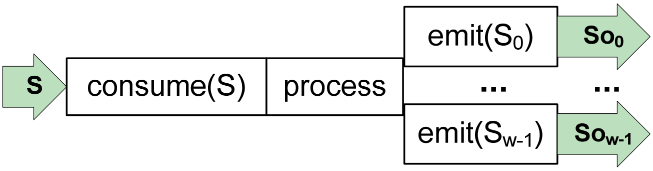


Figure 3. Modules of *fsplit*.

The rate Φ of a stream is defined as the number of tuples per second. The CPU cost C for executing *fsplit* in Figure 3 is computed as

$$C = \Phi\big(cc + cp(o + r + w \cdot b) + ce(r + w \cdot b)\big). \tag{1}$$

In Equation 1, the consume cost *cc* measures reading and de-marshalling one input tuple, the process cost *cp* measures the execution of *rfn* and *bfn* per input tuple, and the emit cost *ce* measures emitting a tuple. b is the *broadcast percentage*, which is the proportion of tuples in the input stream to be emitted to all w output streams according to *bfn*. Notice that b is multiplied by w. r is the *routing percentage*, i.e. the proportion of tuples to be routed according to *rfn*, while o is the *omit percentage*, which is the proportion of tuples that are not emitted at all. As a tuple is either broadcasted, routed, or omitted, $r + b + o = 1$. Thus, the cost C decreases if o increases because of smaller emit cost. Assuming *rfn* routes each tuple with equal probability for all output streams $So_0 \dots So_{w-1}$, the rate of each output stream is $\Phi o_i = \Phi \cdot (b + r / w)$ for all i.

For *scsq-plr*, Table 1 shows percentages o, b, and r, widths w, and output stream rates Φ of *splitstream*$_D$, *splitstream*$_X$, and *splitstream*$_O$, respectively. E (0.4%) tuples are dropped by *splitstream*$_D$. P (99%) and A (0.5%) tuples are routed to *splitstream*$_X$, and D (0.1%) tuples are routed to *dailyexp()*. *splitstream*$_X$ broadcasts A tuples to all *lr()* nodes and routes P tuples of expressway $j = 0\dots L{-}1$ to the corresponding *lr()*. Thus, *splitstream*$_X$ has $b = 0.5\% / 99.5\% \approx 0.5\%$, and $r = 99\% / 99.5\% \approx 99.5\%$. Each *splitstream*$_O$ routes the low rate result stream Φr_i from one *lr*() node.

According to Equation 1, the cost of *fsplit* increases when w is increasing if $b > 0$. Therefore, the cost of *splitstream*$_X$ increases when scaling L, turning *splitstream*$_X$ into the bottleneck when executing *scsq-plr* with high L. Stream replication is a scalability problem for large w, even if b is very close to zero, as in LRB.

Table 1. *Tuple percentages, widths, and output stream rates of splitstream functions in LRB.*

	o	*b*	*r*	*w*	Φ
D	0.4%	0%	99.6%	2	$\Phi_X = 99.5\% \cdot \Phi \quad \Phi_D = 0.1\% \cdot \Phi$
X	0%	0.5%	99.5%	*L*	$\Phi_i = \Phi_X \cdot (0.5\% + 99.5\% / L)$
O	0%	0%	100%	3	$\Phi r_i < \Phi_i$

3. Splitstream Trees

To alleviate the bottleneck in *splitstream*$_X$ when scaling *w*, we propose a hierarchical splitstream plan, called a *splitstream tree*. Each level ℓ in a splitstream tree is numbered, starting from 1 at the root to the depth *d*. Each node in the tree executes *fsplit*, and the width of the nodes on level ℓ is called the fanout f_ℓ of level ℓ. A hierarchical hash function defined in this section enables any user defined *rfn* to be executed in a splitstream tree. Furthermore, we introduce a cost model for splitstream trees. Using this cost model, a splitstream tree with maximum throughput can be generated if *r* and *b* are known. We compare its performance to a practical splitstream tree, which does not require knowledge of *r* and *b*.

3.1 Multi-Level Hash Function

Since each level ℓ in a splitstream tree has fanout f_ℓ, the result of *rfn* on level ℓ must be an integer in range $[0, f_\ell - 1]$. In addition, a splitstream tree must result in the same set of output streams as that of *fsplit*. To fulfill these requirements, the hierarchical hash function defined in Equation (2) is applied on the result of the routing function *rfn(t)* at each level ℓ and tuple *t*.

$$h_\ell(rfn(t)) = \mathrm{mod}\left(\left\lfloor \frac{rfn(t)}{\lambda_{\ell-1}} \right\rfloor, f_\ell\right). \tag{2}$$

The denominator $\lambda_{\ell-1}$ of Equation 2 is the cumulative fanout of level $\ell - 1$, i.e., the fanout that the tuple has undergone in the tree levels above level ℓ. The cumulative fanout at the root is $\lambda_0 = f_0 = 1$, and the cumulative fanout λ_ℓ is

$$\lambda_\ell = \prod_{k \leq \ell} f_k\,. \tag{3}$$

The output streams of a node at level ℓ are denoted $So^{(\ell)}_i$, $i = 0 \ldots f_\ell - 1$. For example, if *splitstream*$_X$ in Figure 2 is executed as a splitstream tree with $f_1 = 2$ and $f_2 = L/2$, then $h_1(rfn)$ routes position reports of even-numbered expressways to output stream $So^{(1)}_0$ and position reports of odd-numbered expressways to $So^{(1)}_1$, according to Equation 2. On level 2, $h_2(rfn)$ routes tuples of expressway number x to $So^{(2)}i$, $i = \lfloor x/2 \rfloor$. *bfn* is the same in all nodes, so that one copy of each broadcast tuple arrives at each leaf.

3.2 A Cost Model for Splitstream Trees

In the following discussion, we assume that then omit percentage $o = 0$, as in our *splitstream*$_X$ example. If $b > 0$ (and thus $r < 1$), the routing percentage decreases at each level. This is because the number of tuples to broadcast stay the same in all output streams, whereas the number of tuples to be routed decreases per level. Equation 4 defines the routing and broadcasting percentages r_ℓ and b_ℓ at level ℓ.

$$r_\ell = \frac{r}{r + b \cdot \lambda_{\ell-1}}; \quad b_\ell = \frac{b \cdot \lambda_{\ell-1}}{r + b \cdot \lambda_{\ell-1}} \tag{4}$$

The rate of one of the output streams at level ℓ is $\Phi o^{(\ell)}$.

$$\Phi o^{(\ell)} = \Phi \cdot \left(b + \frac{r}{\lambda_\ell} \right) \tag{5}$$

The cost C_ℓ of executing a node at level ℓ in a splitstream tree depends on the output stream rate from level $\ell - 1$ according to Equation 5.

$$C_\ell = \Phi o^{(\ell-1)} \cdot \left(cc + (cp + ce) \cdot (r_\ell + f_\ell \cdot b_\ell)\right) \tag{6}$$

The emit capacity E of a node executing the *fsplit* operator is defined as its maximum stream rate. The throughput $\Phi_{\max}$ of a splitstream tree is limited by E and by the level in the splitstream tree with the highest cost.

$$\Phi_{\max} = \frac{E}{\max\limits_{\ell}(C_\ell)} \tag{/}$$

Finally, the total cost of a splitstream tree of depth d can be estimated by adding the splitstream costs for all nodes at each level. The number of nodes at level ℓ is $\lambda_{\ell-1}$.

$$C = \sum_{\ell=1}^{d} \lambda_{\ell-1} \cdot C_\ell \tag{8}$$

3.3 Maxtree and Exptree splitstream trees

Assuming that the percentages r and b are known and constant, it is possible to construct an optimal splitstream tree that maximizes the throughput according to the cost model. We call this splitstream tree *maxtree*, which maximizes the throughput while minimizing the total cost. The cost at level $\ell = 1$ is minimized by choosing $f_1 = 2$, so that $C_1 = \Phi \cdot (cc + (r + 2b) \cdot (cp + ce))$.

Levels are added until $\lambda_d \geq w$. By choosing a fanout f_ℓ of each level $\ell > 1$ such that $C_\ell \leq C_1$, we ensure that no downstream level in the splitstream tree will be a bottleneck. The total cost in Equation 8 is minimized by minimizing the number of splitstream tree levels. The number of levels are minimized by maximizing f_ℓ on all levels $\ell > 1$, while keeping $C_\ell \leq C_1$. Solving f_ℓ for $C_\ell = C_1$ using Equation 6 obtains the following formula for the optimal fanout f_ℓ at level ℓ (see [17] for details).

$$f_\ell = 2 + \left\lfloor \frac{r}{b}\left(1 + \frac{cc}{cp + ce}\right)\left(1 - \frac{1}{\lambda_{\ell-1}}\right) \right\rfloor . \tag{9}$$

The ratio between the costs $a = cc/(cp+ce)$ depends on the costs of *rfn* and *bfn* and on the properties of the computing and network environments. In general, these parameters are unknown, so the formula in Equation 9 cannot be determined. Therefore, *maxtree* can only be used for comparison in controlled experiments where a is known. We determined $a = 1.08$ for *splitstream*$_X$ in a preliminary experiment. To simplify the theoretical discussion of *maxtree*, a was rounded to 1.

Equation 9 shows that optimal fanout f_ℓ increases quickly for small $\ell > 1$ if $r > 0$. Based on this observation, we introduce a splitstream tree called *exptree*, which increases its fanout for each level with a constant factor. *exptree* was set to generate trees with $f_1 = 2$, and $f_\ell = 2 \cdot f_{\ell-1}$ for all $\ell > 1$. We show that the performance of *exptree* will be almost as good as that of *maxtree*, without the need to know a, r, and b.

3.4 Theoretical Evaluation

Throughput and total CPU cost were estimated for the splitstream plans using Equations 7 and 8, assuming $cc = 1$ and $a = 1$. In a scale-up evaluation, w was scaled from 2 to 256 while keeping $b = 0.5\%$, as in *splitstream*$_X$. In a robustness evaluation, b was scaled from 0 to 1 while keeping $w = 64$. Figure 4 shows the estimated performance.

In the scale-up evaluation, the estimated throughput was plotted in Figure 4 (a) as the percentage of emit capacity E. The estimated total CPU cost was plotted in Figure 4 (b). As expected, the single-process *fsplit* degrades when w increases. On the other hand, *fsplit* also consumes the least total CPU. The CPU cost of *exptree* increases when a new tree level is added, e.g. when increasing w from 8 to 16. For such small values of $b = 0.5\%$ as in LRB, *maxtree* generates a shallower tree and thus consumes less CPU resources than *exptree*.

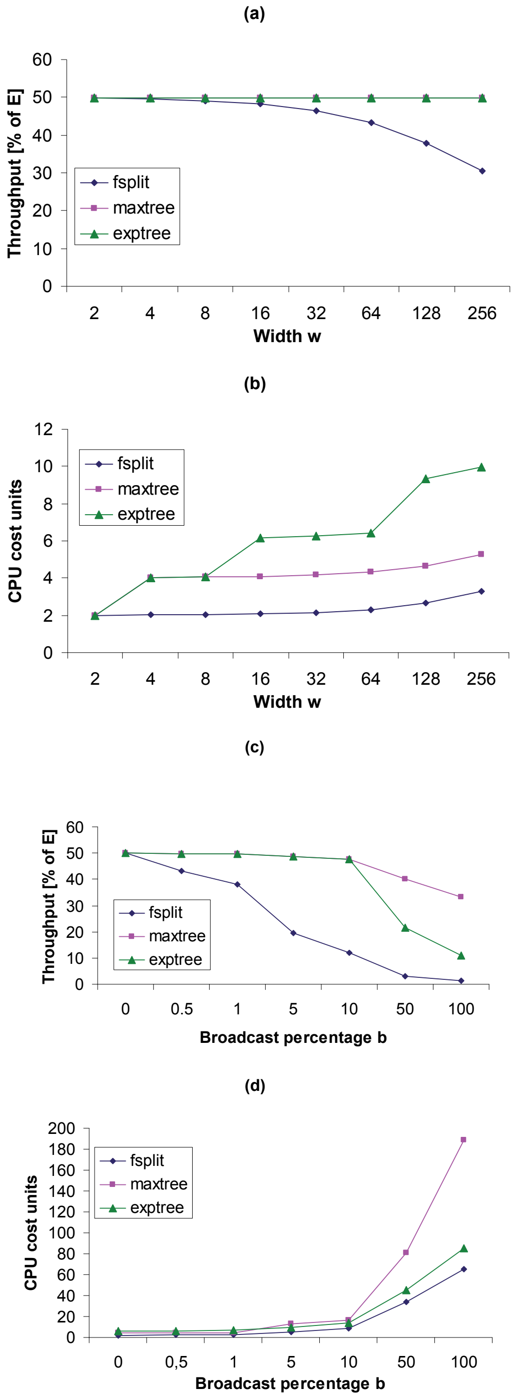


Figure 4. (a) Estimated throughput and (b) total CPU cost, $b = 0.5\%$. (c) Estimated throughput and (d) total CPU cost, $w = 64$.

When scaling *b* in the robustness evaluation, Figure 4 (d) shows that the CPU cost of *maxtree* increases sharply when *b* increases. If b ≈ 1 in Equation 9, $f_\ell = 2$ on all *maxtree* levels, resulting in a binary tree. A splitstream tree with so many nodes consumes a lot of CPU. Figure 4 (c) shows that all splitstream functions have the same throughput for $b = 0$, but the throughput of *fsplit* drops quickly when *b* increases. For moderate values of *b* (up to 10%), the estimated throughput of *exptree* is the same as that of *maxtree*. For higher values of *b*, the estimated throughput of *exptree* is lower, however much better than *fsplit*.

4. Experimental Setup

The splitstream functions were implemented using our prototype DSMS SCSQ [21]. Queries and views are expressed in terms of typed functions in SCSQ's functional query language SCSQL, resulting in one of three collection types *stream*, *bag*, and *vector*. A stream is an object that represents ordered (possibly unbounded) sequences of objects, a bag represents relations, and a vector represents bounded sequences of objects. For example, vectors are used to represent stream windows, and vectors of streams are used to represent ordered collections of streams.

Queries are specified using SCSQL in a client manager. The distributed execution plan of a query forms a directed acyclic graph of stream processes (SPs), each emitting tuples on one or more streams. Continuous query definitions are shipped to a coordinator. Unless otherwise hinted, the coordinator dynamically starts new SPs in a round robin fashion over all its compute nodes, so that the load is balanced across the cluster. The coordinator returns a handle of each newly started SP.

In the SPs, a cost-based query optimizer transforms each query to a local stream query execution plan (SQEP), by utilizing the query optimizer of Amos II [9]. A SQEP reads data from its input streams and delivers data on one or more of its output streams. Stream drivers for several communication protocols are implemented using non-blocking I/O and carefully tuned buffers. A timer flushes the output stream buffers at regular time intervals to ensure that no tuples will remain for too long. The SCSQ kernel is implemented in C, where SQEPs are interpreted. SQEPS may call the Java VM to access DBMSs over JDBC. Thus, an SP may be stateful in that it stores, indexes, and retrieves data using internal main memory tables or external databases. In *scsq-plr*, local main memory tables are used to store account balance data, and MySQL is used to store daily expenditure data.

In our experiments, each SP is a UNIX process on a cluster of compute nodes featuring two quad-core Intel® Xeon® E5430 CPUs @ 2.66GHz and 6144 KB L2 cache. Six such compute nodes (48 cores in total) were available for the experiments. For large splitstream trees, there were fewer CPUs than SPs. Then, some SPs were co-located on the same CPU. For inter-node communication, TCP/IP was used over gigabit Ethernet. Intra node communication used TCP/IP over the loopback interface. Throughput is computed by measuring the execution time of SCSQ over a finite stream. The CPU usage of each SP is determined using a profiler in SCSQ that measures the time spent in each function by interrupt driven sampling.

5. Preliminary Experiments

Two preliminary experiments were performed. The purpose of the first one is twofold: We show that the emit capacity for moderately sized tuples is bound by the CPU and not by the network. We also show that the emit capacity E for an SP, and thus the cost, is the same for moderately sized tuples no matter if streaming inter or intra node. Since the cost is the same, the scheduling of SPs is greatly simplified.

One SP was streaming tuples of specified size to another SP, which counted them. Intra node streaming was performed with the SPs on the same compute node, while they were on different nodes for inter node streaming. The emit capacity is shown in Figure 5 (a), with less than 3.5% relative standard deviation. For tuples of moderate size, the emit capacity is the same for inter and intra node streaming. LRB input stream tuples have 15 attributes, occupying 83 bytes including header. The network bandwidth consumption is 143 Mbit/s for these tuples, which is significantly less than the capacity of a gigabit Ethernet interface. Streaming moderately sized tuples as in LRB is CPU bound, because of the overhead of marshalling and (de)allocating many small objects. For tuples of size greater than 512 bytes, the intra node throughput is better. Usually however, tuples are smaller.

The purpose of the second preliminary experiment is to measure consume, process, and emit costs (*cc*, *cp*, and *ce*) *splitstream*$_X$ in our environment, as required by *maxtree*. We do that by executing *splitstream*$_X$ as an *fsplit* with $w = 1$. One SP generated a stream of 3 million tuples. A second SP applied *fsplit* with $w = 1$ on the stream from the first SP, using the *rfn* and *bfn* of *splitstream*$_X$. A third SP counted the number of tuples in the single output stream from *fsplit*.

The CPU times obtained from the *fsplit* SP are shown in Figure 5 (b). Using these CPU times, $a = cc / (cp + ce) \approx 1.08$, which is used in all *maxtree* experiments. Furthermore, the throughput of this simple splitstream was $\Phi_{max} = 109) \cdot 10^3$ tuples per second (relative standard deviation 0.6%). In LRB, the maximum input stream rate is 1670 tuples per second and expressway, so this throughput corresponds to 65 (109000/1670) expressways in LRB. No splitstream tree can be expected to have higher throughput than *fsplit* with $w = 1$. Thus, no splitstream tree will be able to split the input stream of LRB for $L > 65$.

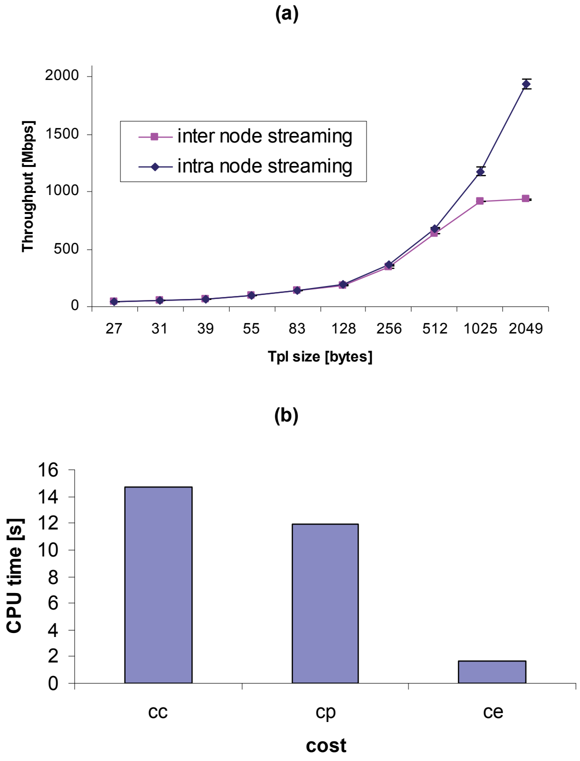


Figure 5. (a) Inter and intra node emit capacity, (b) CPU time breakdown for *fsplit* with $w = 1$.

6. Experimental Evaluation

The goal with the experimental evaluation is to investigate the properties of the splitstream plans in a practical setting. Throughput and total CPU consumption were studied in a scale-up experiment and a robustness experiment, set up in the same way as in the analytical evaluation. In order to establish statistical significance, each experiment was performed five times and the average is plotted in the graphs.

Figure 6 shows the throughput and CPU usage of the splitstream trees. Error bars (barely visible) show one standard deviation. All experimental results agree perfectly with the theoretical estimates in Figure 4 with one exception: The measured throughput of *maxtree* shown in Figure 6 (c) was significantly lower for large values of *b* than estimated. This is because the total CPU usage exceeds the CPU resources available for our experiments. If resources were abundant, *maxtree* should have been feasible for splitting streams with a high broadcast percentage. If resources are limited, *exptree* is shown to achieve the same throughput as *maxtree* at a smaller CPU cost. The experiments confirm that our cost model is realistic.

6.1 Autosplit

We observe that *exptree* achieves the same scale-up as *maxtree*. Furthermore, the robustness of *exptree* is the same as that of *maxtree* when resources are not abundant. Based on these results, we implement *autosplit* using the following decision rule: If *bfn* is present, generate an *exptree*. If only *rfn* is present and thus $b = 0$, generate a single *fsplit*, since a single *fsplit* has the same throughput as the splitstream trees for $b = 0$, but consumes less CPU.

6.2 LRB Performance

To verify the high scalability of *autosplit*, it was used as the splitstream function in *scsq-plr* as shown in Figure 2. *autosplit* generated an *exptree* for *splitstream*$_X$ and *fsplit* for *splitstream*$_D$ and *splitstream*$_O$ since they had no *bfn*. The performance of LRB using *autosplit* is compared to LRB using *fsplit* in all splitstream functions. To simplify the experiments, the *dailyexp()* node was disabled since the daily expenditure processing has no bearing on scalability of LRB stream processing in *scsq-plr*.

When using the round robin scheduler of the coordinator described in Section 4, *scsq-plr* with *autosplit* achieved $L = 52$. The limiting factor was that the first node of the plan was not granted enough CPU resources, because too many SPs were assigned to the same multi-core compute node. By adding a hint to the coordinator to limit the number of SPs on the first compute node, the L-rating for *autosplit* improved to $L = 64$, as illustrated by Figure 7. The y-axis is the MRT, and the x-axis is the number of minutes into the simulation. *fsplit* keeps up until minute 125, when response time accumulates and exceeds the allowed MRT at 129 minutes.

When $b = 0.5\%$ as in *splitstream*$_X$, the maximum throughput of *fsplit* with $w = 64$ is 100000 tpl/sec according to the results in Figure 6(c). At 125 minutes, the stream rate for $L = 64$ is getting close to 100000 tpl/sec. Thus, *fsplit* is unable to keep up with the increasing input stream rate. Since the maximum throughput of *exptree* is higher, *autosplit* achieves the higher L-rating of $L = 64$. The bumps in the curves are because of cron jobs executing on the compute nodes beyond our control.

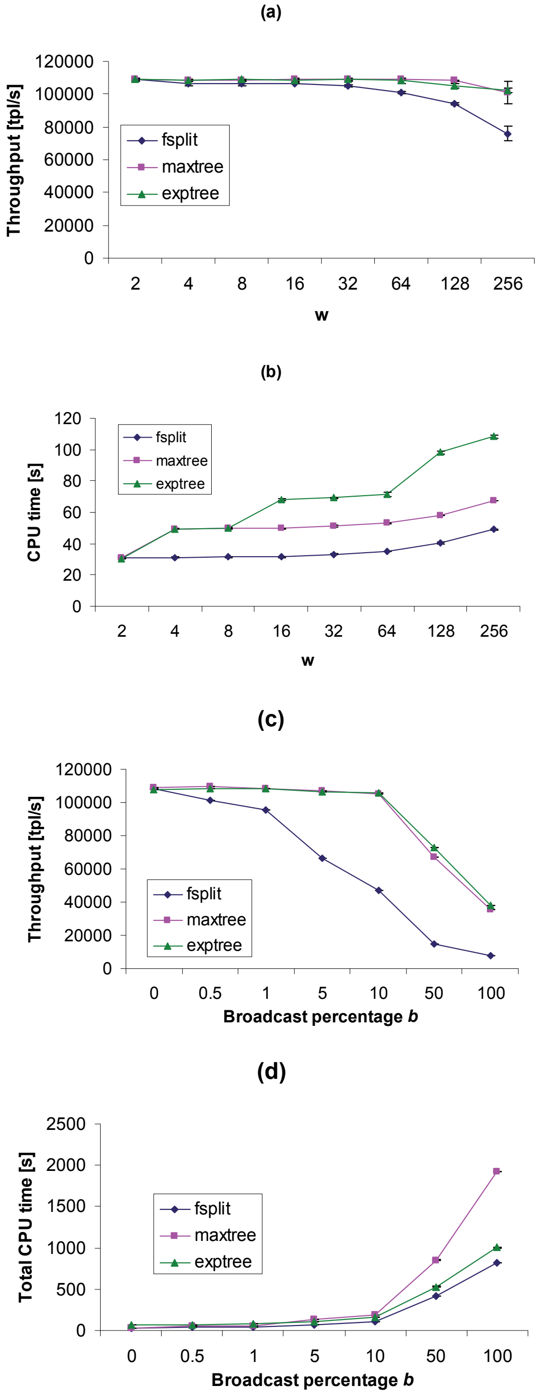


Figure 6. (a) Measured throughput and (b) measured total CPU cost for *b* = 0.5%. (c) Measured throughput and (d) measured total CPU cost for *w* = 64.

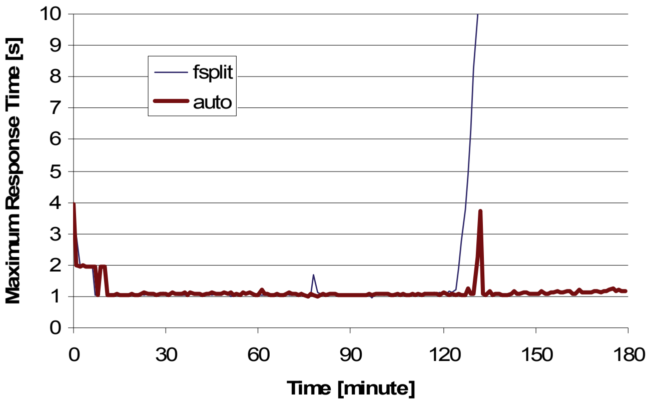


Figure 7. Maximum response time for $L = 64$.

In conclusion, we have shown that *fsplit* cannot achieve $L = 64$ in LRB, and that smart scheduling is necessary to take full advantage of *autosplit*. *fsplit* with smart scheduling was measured to achieve $L = 52$. Notice that in standard LRB, the improvement with *autosplit* could not be expected to be very large. However, as indicated theoretically by Figure 4 (a) and experimentally in Figure 6 (c), the gain will be bigger if the broadcast percentage b is greater.

7. Related Work

This paper complements other work on parallel DSMS implementations [4, 8, 12, 15, 19], by allowing the user to specify non procedural stream splitting, and by parallelizing the execution of stream splitting. This allows parallel execution of expensive queries over massive data streams.

In previous work [22], we introduced stream processes, allowing the user to manually specify parallel stream processing. The stream splitting proved to be very efficient for online spatio-temporal optimization of trip grouping [7], based on static or dynamic routing decisions. Similarly, GSDM [12] distributed its stream computations by selecting and composing distribution templates from a library, in which some basic templates were defined including both splitting and joining. By contrast, the stream splitting in this paper is specified through declarative second order splitstream functions, allowing optimizable stream splitting insensitive to the percentages of tuples to broadcast or route.

Gigascope [4] was extended with automatic query dependent data partitioning in [14] for queries that monitored network streams. The query execution was automatically parallelized by inferring partitioning sets based on aggregation and join attributes in the queries. The stream splitting was performed in special hardware, which provided high throughput. By contrast, we have developed a method to split streams involving both routing and broadcasting by generating efficient hierarchical splitstream plans executing on standard PCs. Furthermore, splitstream functions allow the user to declaratively specify splitstream strategies, which allows parallelization of queries that cannot be parallelized automatically.

Efficient locking techniques were developed in [5] to parallelize aggregation operators using threads. Since SCSQ uses processes instead of threads for parallelization, locking is not an issue.

Partitioning a query plan by statically distributing the execution of its operators proved to be a bottleneck in [13]. In [2], query plans were partitioned by dynamically migrating operators between processors. However, expensive operators are still bottlenecks. In our work, the bottleneck was overcome by splitting the input stream into several parallel streams, and further reduced by parallelizing the stream splitting itself. Furthermore, allowing both routing and broadcasting provide a powerful method to parallelize queries, as shown by *scsq-plr*.

The Flux operator [18] dynamically repartitions stateful operators in running streams by adaptively splitting the input stream based on changes in load. By contrast, we have studied user defined stream splitting. Dynamic scheduling of distributed operators in continuous queries has been studied in [19] and [23]. A dynamic distributed scheduling is introduced in [19] based on knowledge about anticorrelations in load between different independent operators in a plan. In [23], stream operators are dynamically migrated between compute nodes based on the current load of the nodes. By contrast, this paper concentrates stream splitting for parallel processing downstream. However, scheduling proved to be important, and future work should investigate the effectiveness of these approaches when used with parallelization functions.

Dryad [10] generalizes Map-Reduce [6] by implementing an explicit process graph building language where edges represent communication channels between vertices representing processes. By contrast, SCSQ users specify parallelization strategies over streams on a higher level using declarative second order parallelization functions. These parallelization functions are automatically translated into parallel execution plans (process graphs) depending on the arguments to the parallelization functions.

SCOPE [3] and Map-Reduce-Merge [20] are more specialized than Dryad, providing an SQL-like query language over large distributed files. The queries are optimized into parallel execution plans. Dryad, Map-Reduce-Merge, and SCOPE operate on sets rather than streams. None of these provide parallelization functions.

Out of the existing implementations of LRB, IBM's Stream Processing Core (SPC) is the only attempt to parallelize the execution [13]. The SPC implementation of LRB was partitioned into 15 building blocks, each of which performed a part of the implementation. One processing element computed all segment statistics on a single CPU, which proved to be a bottleneck. With the SCSQ implementation and *autosplit*, we achieved over 25 times the L-rating of the SPC implementation by user defined parallelization. The performance difference between SCSQ and SPC illustrates (i) the importance of how the execution is parallelized; and (ii) the usefulness of splitstream functions where the user provides application knowledge for the parallelization declaratively by specifying *rfn* and *bfn*.

For streams, *rfn* and *bfn* are analogous to fragmentation and replication schemes for distributed databases [16]. However, for distributed databases the emphasis is mainly on distributing data without skew. In our case, there are orders of magnitude higher response time demands on stream splitting and replication than on disk data fragmentation and replication. Therefore, the performance of stream splitting is critical.

8. Conclusions and Future Work

We investigated the performance of *splitstream functions*, which are parallelization functions that provide both partitioning and replication of an input stream into a collection of streams. A splitstream function is compiled into a *splitstream plan*. We first defined a theoretical cost model to estimate the resource utilization of different splitstream plans, and then investigated the performance of these splitstream plans experimentally using the SCSQ DSMS. Based on both theoretical and experimental evaluations, we devised the splitstream function *autosplit*, which splits an input stream, given the degree of parallelism, and two functions specifying how to distribute and partition the input stream. The *routing function* returns the output stream number for each input tuple that should be routed to a single output stream. The *broadcast function* selects the tuples that should be broadcasted to all output streams. *autosplit* was shown to generate a robust and scalable execution plan with performance close to what is theoretically optimal for a tree shaped execution plan. *autosplit* was used to parallelize the Linear Road DSMS Benchmark (LRB), and shown to achieve an order of magnitude higher L-rating than other published implementations.

A simple scheduler was used in the experiments, which balanced the load evenly between the compute nodes for all splitstream plans. This scheduler achieved $L = 52$ in the LRB experiment. By hinting the scheduler not to overload the first node of the execution plan, the L-rating improved to 64, which is close to the theoretically maximum throughput for *scsq-plr* in our cluster environment.

As splitstream has shown to be sensitive to the cost of *rfn* and *bfn*, future work includes optimizing splitstream for a wider class of *rfn* and *bfn*. By devising a cost model like in [24], the scheduling of SPs can be further improved. The robustness of dynamic rescheduling and SP migration should be investigated. It should be investigated whether other, non-tree shaped splitstream plans can improve performance further. Furthermore, other application scenarios are being studied within the iStreams project [11].

Acknowledgements

This project is supported by VINNOVA in the iStreams project. The experiments were performed using UPPMAX resources. We greatly appreciate the insightful comments by the anonymous reviewers.

References

1. Arasu, A., et al.: Linear Road: A Stream Data Management Benchmark. In: VLDB (2004)
2. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-Based Load Management in Federated Distributed Systems. In: NSDI (2004)
3. Chaiken, R., et al.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In: VLDB (2008)
4. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: A Stream Database for Network Applications. In: SIGMOD (2003)
5. Das, S., Antony S., Agrawal, D., El Abbadi, A.: Thread Cooperation in Multicore Architectures for Frequency Counting over Multiple Data Streams. In: VLDB (2009)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI (2004)
7. Gidofalvi, G., Pedersen, T. B., Risch, T., Zeitler, E.: Highly scalable trip grouping for large-scale collective transportation systems. In: EDBT (2008)
8. Girod, L., Mei, Y., Newton, R., Rost, S., Thiagarajan, A., Balakrishnan, H., Madden, S.: XStream: A Signal-Oriented Data Stream Management System. In: ICDE (2008)
9. Risch, T., Josifovski, V., Katchaounov, T.: Functional Data Integration in a Distributed Mediator System. In: Gray, P.M.D., Kerschberg, L., King, P.J.H., Poulovassilis, A. (eds.): The Functional Approach to Data Management (2004)
10. Isard, M., et al.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. ACM SIGOPS Operating Systems Review, Volume 41, 59–72, (2007)

11. iStreams homepage, http://www.it.uu.se/resnearch/group/udbl/html/iStreams.html.
12. Ivanova, M., Risch, T.: Customizable Parallel Execution of Scientific Stream Queries, In: VLDB (2005)
13. Jain, N., et al.: Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In: SIGMOD (2006)
14. Johnson, S., Muthukrishnan, Shkapenyuk, V., Spatscheck, O.: Query-Aware Partitioning for Monitoring Massive Network Data Streams. In: SIGMOD (2008)
15. Liu, B., Zhu , Y., Rundensteiner, E. A.: Run-Time Operator State Spilling for Memory Intensive Long-Running Queries. In: SIGMOD (2006)
16. Özsu, M. T., Valduriez, P. Principles of Distributed Database Systems, Second Edition. Prentice-Hall (1999)
17. SCSQ-LR homepage, http://user.it.uu.se/~udbl/lr.html.
18. Shah, M. A., Hellerstein, J. M., Chandrasekaran, S., Franklin, M. J.: Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In: ICDE (2002)
19. Xing, Y., Zdonik, S., Hwang, J.-H.: Dynamic Load Distribution in the Borealis Stream Processor. In: ICDE (2005)
20. Yang, H., Dasdan, A., Hsiao, R.-L. Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD (2007)
21. Zeitler, E., Risch, T.: Processing high-volume stream queries on a super-computer. In: ICDE Workshops (2006)
22. Zeitler, E., Risch, T., Using stream queries to measure communication performance of a parallel computing environment. In: ICDCS Workshops (2007)
23. Zhou, Y., Ooi, B. C., Tan, K.-L., Wu. J.: Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In: On the Move to Meaningful Internet Systems (2006)
24. Zhou, Y., Aberer, K., and Tan, K.-L.: Toward massive query optimization in large-scale distributed stream systems. In: Middleware (2009)

Paper V

Erik Zeitler, Tore Risch: Massive scale-out of expensive continuous queries.
In Proceedings of the 37th International Conference on
Very Large Data Bases (VLDB), 2011.

Proceedings of the VLDB Endowment, Vol. 4, No. 11.
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

Massive scale-out of expensive continuous queries

Erik Zeitler and Tore Risch
Department of Information Technology, Uppsala University, Sweden

{erik.zeitler, tore.risch}@it.uu.se

Abstract– Scalable execution of expensive continuous queries over massive data streams requires input streams to be split into parallel sub-streams. The query operators are continuously executed in parallel over these sub-streams. Stream splitting involves both partitioning and replication of incoming tuples, depending on how the continuous query is parallelized. We provide a stream splitting operator that enables such customized stream splitting. However, it is critical that the stream splitting itself keeps up with input streams of high volume. This is a problem when the stream splitting predicates have some costs. Therefore, to enable customized splitting of high-volume streams, we introduce a parallelized stream splitting operator, called parasplit. We investigate the performance of parasplit using a cost model and experimentally. Based on these results, a heuristic is devised to automatically parallelize the execution of parasplit. We show that the maximum stream rate of parasplit is network bound, and that the parallelization is energy efficient. Finally, the scalability of our approach is experimentally demonstrated on the Linear Road Benchmark, showing an order of magnitude higher stream processing rate over previously published results, allowing at least 512 expressways.

1. Introduction

Decision-making in real time over streaming data requires processing of continuous queries involving expensive computations. Applications include scientific and engineering settings where complex analyses are performed over streams of high volume from instruments and equipment. To enable scalable execution of such continuous queries with expensive computations, input streams must be split into parallel sub-streams over which the expensive query operators are continuously executed in parallel. Naïvely implemented, stream splitting becomes a bottleneck for input streams of high volume, non-trivial parallelization conditions, or when massive parallelization of query operators is required.

We eliminate this bottleneck by introducing a novel parallel stream splitting operator, called *parasplit,* which splits input streams of high volume according to non-trivial customized parallelization conditions into massively parallel sub-streams. Expensive query operators are applied on these sub-streams in parallel. By parallelizing not only the expensive query operators but also the stream splitting, we show that the maximum stream rate of parasplit is network-bound and not bound by the cost of the split conditions. We estimate energy efficiency by measuring CPU cost, and show that the additional CPU cost of parallelizing the stream splitting in parasplit is moderate compared to the cost of only executing the stream splitting. Thus, we enable processing of expensive continuous queries close to the maximum capacity of the network.

To facilitate data-parallel stream processing, an input stream S must be split into q parallel streams over which an expensive continuous query operator Q is applied in parallel on separate CPUs PQ_j, $j = 1 \ldots q$. A typical parallelization of an expensive function Q on a high-volume stream S is shown in Figure 1. A *splitstream* operator splits S into q parallel streams by partitioning and/or replicating input streams into a collection of streams. For each tuple in the input stream, splitstream decides whether the tuple should be sent to one specific DSMS node (partitioning) or to many DSMS nodes (replication), according to a specification provided by the user. Q is applied on each parallel stream. The result streams from each application of Q can be merged, e.g. based on time stamps [4]. It is easy to see that when scaling the cost of Q and the rate of the input stream S, it is necessary to scale the parallelism q in order to keep up with the input stream rate.

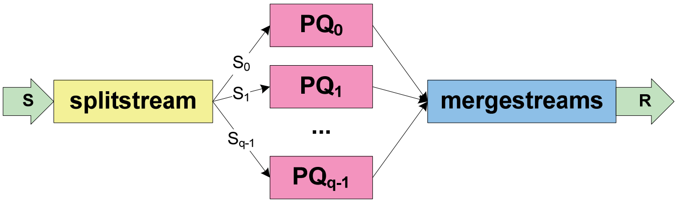


Figure 1. Streamed data parallelism.

For each tuple in S, it must be decided to which parallel sub-stream S_j the tuple should be sent. However, non-trivial routing decisions will prohibit scalability when the input streams rate increases. Furthermore, a large value of q may affect the cost of the split. Parasplit is a splitstream function that eliminates these bottlenecks by parallelizing its split predicates.

We proceed by introducing splitstream functions in general and parasplit in particular (Section 2). In Section 3, we introduce *stream processes* (SPs) as a DSMS node executing a sub-plan in a distributed environment. A cost model for the SPs used by parasplit is defined (Section 3.1), resulting in general heuristics for automatic parallelization of parasplit (Section 3.2). Parasplit has been implemented in the parallel DSMS SCSQ [32]. It is shown both theoretically and experimentally how to achieve network bound stream rates, independent of the cost of the stream splitting function and *q* (Section 4.1). The cost heuristic is validated experimentally and compared to the theoretical model (Section 4.2). Finally, we apply parasplit on the Linear Road Benchmark (LRB) [3], and show that it enables an order of magnitude higher stream processing rate over previously published results, allowing 512 expressways (Section 4.3). We conclude by contrasting this contribution to other work in parallel stream processing and outline future work.

2. Splitstream Functions

A splitstream function has the basic signature

splitstream(stream S, integer q, function rfn, function bfn) → vector of stream sv.

Variants of splitstream may have additional parameters. The input stream *S* is split into *q* output streams in the vector *sv*. The first functional argument *rfn* is the routing function, having signature *rfn*(*object tpl*, *integer q*) → *integer*, which returns the output stream number (between 0 and *q*–1) for each tuple that should be routed to a single output stream. The function *bfn*(*object tpl*) → *boolean* is the broadcast function, which returns true for tuples to be broadcasted to all output streams. *bfn* and *rfn* return *nil* for tuples that should be neither broadcasted nor routed, i.e. omitted. For example, splitstream in Figure 1 splits the input stream *S* into *q* parallel streams according to its routing and broadcast functions, resulting in a vector of *q* parallel streams. Since *rfn* and *bfn* have non-zero cost, splitstream may become a bottleneck for high input stream rates. The splitstream function

parasplit(stream S, integer q, function rfn, function bfn) → vector of stream sv

eliminates this bottleneck by scaling out the execution of *rfn* and *bfn* in addition to *Q*.

A call to parasplit dynamically creates a distributed execution plan that consists of many intercommunicating distributed operating system processes, each running a sub-plan. Such processes are called *stream processes*, (SPs). Each SP computes tuples of its output streams by processing its input streams according to its local sub-plan.

Figure 2 shows the SPs involved in parasplit, with $q = 8$ and $p = 3$. First, the window router *PR* reads entire physical windows of size *W* containing binary represented tuples from the input stream *S*. Each physical window is uniformly and randomly routed to one of the *p* parallel sub-streams S_i, $i =$

$0\ldots p-1$. Uniform routing balances T_{ij} for all i, while random routing eliminates any time periodicities present in the attributes used for splitting, which balances T_{ij} for all j over time. Since the window router is processing entire physical windows, its cost is not a bottleneck for sufficiently large windows and suitable scheduling, as will be validated.

Second, each window splitter PS_i unpacks the tuples of the physical windows of its sub-stream S_i received from PR. According to the stream splitting functions *rfn* and *bfn*, each tuple is distributed to zero, one or more continuous query processors PQ_j, $j = 0\ldots q-1$. The output stream rate of a window splitter is potentially greater than its input stream rate if any tuples are broadcasted. Since a compute node in a cluster usually has only a single network interface, its output stream rate may not exceed its input stream rate. Therefore, parasplit schedules all window splitters on other compute nodes than the window router.

Third, each query processor PQ_j merges all received streams T_{ij}, $i = 0\ldots p-1$, into a local stream U_j, over which the expensive query operator Q is executed. Since the tuples arriving at PQ_j have travelled through p different window splitters in parallel, the order of arrival of the tuples at each query processor may be different than their order of arrival at the window router. Each tuple of the input stream S is time stamped before arrival. To maintain time order in U_j, each query processor PQ_j always moves the tuple from T_{ij}, $i = 0\ldots p-1$, with the least timestamp to U_j. Since the window router uniformly distributes tuples over all S_i, all streams T_{ij} have the same rate for all i. Therefore, the merge operator of PQ_j does not have to idle-wait for tuples due to empty inputs.

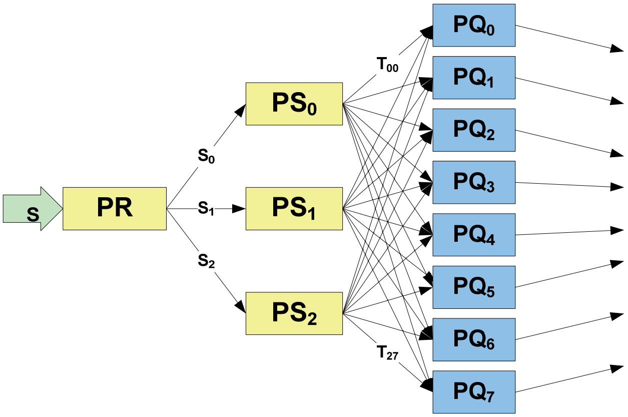


Figure 2. Parasplit.

3. A Cost Model for Stream Processes

A distributed continuous query execution plan consists of intercommunicating SPs. Each SP runs an execution plan containing some or all of the sub-plans (modules) and operators shown in Figure 3. Each input stream S_j, $j = 0 \dots u-1$, is first read and de-marshalled by a *consume* operator. The *merge* module merges several streams into one according to its installed sub-plan. The *compute* module executes a continuous sub-plan over the merged input streams. In the *split* module, tuples that are emitted from the compute module are processed according to stream splitting partitioning and replication conditions specified by *rfn* and *bfn*. Each *emit* operator marshals and emits tuples to its result stream R_i, $i = 0 \dots q-1$. In parasplit, the window router *PR* and window splitters PS_i have only one consume operator but several emit operators, while each query processor PQ_j has several consume operators. For example, in the parasplit example shown in Figure 2, each SP executing PQ_j merges three input streams and emits one output stream.

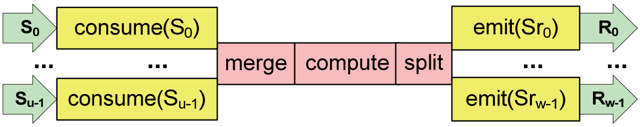


Figure 3. Modules and operators in a stream process.

An SP in SCSQ is implemented as a UNIX process in a cluster. However, the cost model can be applied on SPs in any distributed environment. Consume and emit operators in SCSQ are implemented for TCP, UDP, MPI, and UNIX pipes.

Next, we introduce a cost model for processing a tuple in an SP. We investigate how the cost of executing each SP in parasplit is affected by the scale-out of the window splitters PS_i and the query processors PQ_j. Based on this, we define a heuristic that enables us to easily achieve maximum input stream rate.

3.1 Cost of streaming a tuple

The CPU cost C for an SP to process an incoming tuple is computed using Equation (1).

$$\begin{aligned} C = cr + (cp + cm) \cdot u + cq + \\ \sigma\big(cs(o + r + q \cdot b) + ce(r + q \cdot b)\big) \end{aligned} \qquad (1)$$

The read cost *cr* is the cost of reading and de-marshalling an input tuple in a consume operator. As a merge module polls the input streams for pending data and merges the read tuples, it has both a poll cost *cp* and a merge cost *cm*, which are multiplied by the number of input streams *u*. The query cost *cq* is the cost per tuple of executing the compute module on the merged stream. The selectivity of the sub-plan is σ, so the result stream rate is multiplied by σ, which affects the costs of split and emit. The split cost *cs* is the cost to execute the split module per tuple received from the compute module. The emit cost *ce* is the cost of marshalling and emitting a tuple on one output stream. In the split module, *b* is the proportion of tuples to be replicated to all *q* output streams according to a replication condition. *b* is called the broadcast percentage. Hence, *b* is multiplied by *q* in Equation (1). The routing percentage *r* is the proportion of output tuples to be routed according to a partitioning condition. *o* is the omit percentage, which is the proportion of output tuples that are neither broadcasted nor routed. As each output tuple is either broadcasted, routed, or omitted, $r + b + o = 1$.

The coefficients of the general cost Equation (1) depend on the operators executed by the SP. We now specialize Equation (1) for each kind of SP in parasplit.

3.1.1 Window router

The cost C_{PR} of executing the window router *PR* in parasplit is given by Equation (2). *PR* has only one consume operator and therefore does not poll or merge any tuples. Furthermore, *PR* does not execute any compute module. Therefore, *PR* has $cp = cm = cq = 0$ and $\sigma = 1$. As *PR* routes every incoming window to the window splitters PS_i, $r = 1$, and $b = o = 0$. Finally, the split and emit costs cs_W and ce_W of *PR* are the costs of distributing entire windows of the input stream *S* to the window splitters.

$$C_{PR} = cr_W + cs_W + ce_W \tag{2}$$

3.1.2 Window splitter

Equation (3) models the cost of processing a tuple in a window splitter PS_i. Like *PR*, each PS_i has $cp = cm = cq = 0$ and $\sigma = 1$, as it does not execute any merge or compute module. The cost of reading a tuple from *PR* is estimated by cr_W, as each incoming stream S_i contains the same kind of physical windows as the incoming stream to *PR*. *cs* estimates the cost of executing *rfn* and *bfn* per tuple in PS_i. *ce* models the cost of emitting each tuple from PS_i.

$$C_{PS} = cr_W + cs(o + r + q \cdot b) + ce(r + q \cdot b) \tag{3}$$

3.1.3 Query processor

Equation (4) models the cost per tuple in each query processor PQ_j of merging the streams T_{ij} from the window splitters PS_i, $i = 0\ldots p{-}1$, and executing the continuous query operator *Q*. *cr* is the read cost of reading a single tuple.

As each PQ_j merges p streams, the poll and merge costs are multiplied by p. Finally, O is the cost of executing the expensive continuous query operator Q in the compute module and emitting the result downstream.

$$C_{PQ} = cr + p \cdot (cp + cm) + O \tag{4}$$

3.2 A heuristic stream rate model for parasplit

We define the maximum stream rate of each kind of SP in parasplit as Φ_{PR} for the window router PR, Φ_{PS} for the window splitters PS_i, and Φ_{PQ} for the query processors PQ_j. Each of these maximum stream rates are potential bottlenecks, since they all affect the maximum possible stream rate in parasplit. In other words, the maximum stream rate of parasplit $\Phi_{\mathrm{PARASPLIT}} = \min(\Phi_{\mathrm{PR}}, \Phi_{\mathrm{PS}}, \Phi_{\mathrm{PQ}})$. In particular, the window router is the critical bottleneck, since the window splitter and query processors are parallelized. The hypothesis is that for a large enough window size W, Φ_{PR} should be network bound.

3.2.1 Physical window size

The input stream to PR is delivered as physical windows, each window containing W bytes. The cost of PR is influenced by the physical window size W. With a large enough window, the cost of executing PR for each window is insignificant compared to the communication cost. Then, Φ_{PR} is expected to approach maximum network speed, independent of communication protocol, which is validated experimentally for 1 Gbps. The first step is to find a large enough W such that Φ_{PR} is maximized. To determine the window size W, we profile the window router once and for all in the cluster used. PR is executed with $p = 4$ routing windows to PS_i containing only the consume operators. The window size is doubled until there is less than 0,15% improvement of Φ_{PR}. On our cluster, we achieved $\Phi_{\mathrm{PR}} = 987$ Mbps for $W = 16$ kB, which is close to our wire speed, so PR is network bound. The profiling is fast, as each measurement is run for 1 second. In our experiments, it converged after 9 rounds.

3.2.2 Window splitter parallelism

For a given call to parasplit, p must be determined. Let Φ_{D} be the desired input stream rate. We choose p such that $p \cdot \Phi_{\mathrm{PS}} \geq \Phi_{\mathrm{D}}$ so that the window splitters are not bottlenecks. Equation (3) estimates the cost per tuple of splitting a tuple in a window splitter according to *rfn* and *bfn*. In our heuristic we assume that $cr_W = 0$, as the cost of reading a tuple from a physical window is assumed to be low in comparison to the more expensive *rfn* and *bfn*. We assume $o = 0$, which will over-estimate C_{PS}.

Assuming that parasplit is mainly used for scaling out computations by partitioning the input data stream, we estimate the broadcast frequency to be rather low. To accommodate parallelization strategies involving high amounts of broadcast, *b* is configurable, but we set a default value of *b* = 0.01. Based on this reasoning, we approximate C_{PS} of Equation (3) with $\hat{C}_{PS}$ of Equation (5):

$$\hat{C}_{PS} = (cs + ce)\cdot(0.99 + 0.01\cdot q) \tag{5}$$

Next, *cs*+*ce* is estimated by measuring the maximum stream rate $\Phi_{PS}^{(1)}$ of a single window splitter on a small section of the input stream with $q = 1$. $cs+ce = 1/\Phi_{PS}^{(1)}$. Furthermore, *p* should be as small as possible to minimize the merge cost in Equation (4), while still satisfying $p \geq \Phi_D/\Phi_{PS}$. Therefore, *p* is estimated by Equation (6). The maximum stream rate of parasplit with this heuristic value of *p* is evaluated in the experiments.

$$\hat{p} = \left\lceil \frac{\Phi_D}{\Phi_{PS}^{(1)}}\cdot(0.99 + 0.01\cdot q) \right\rceil \tag{6}$$

By estimating $o = 0$, our heuristic in Equation (6) may choose a *p* that is slightly greater than what is needed to keep up with the desired stream rate Φ_D. A too low *p* may not keep up with Φ_D. However, we note that if a lower bound of *o* is known, a smaller *p* could be chosen to save cost in Equation (4), which is future work.

3.2.3 Efficiency of parasplit

To estimate the energy efficiency of parasplit, we define efficiency η as the CPU time ratio between all PS_i, $i = 0\ldots p-1$, and all SPs involved in parasplit. Formally, the efficiency is given by Equation (7), where $C_{PQ}^{O=0}$ is the cost of performing the read, poll, and merge in *PQ*, i.e. the cost of executing *PQ* with no continuous query installed. With no query execution cost, $O = 0$.

$$\eta = \frac{p\cdot C_{PS}}{C_{PR} + p\cdot C_{PS} + q\cdot C_{PQ}^{(O=0)}} \tag{7}$$

The efficiency is a measurement of the additional work incurred by executing parasplit in comparison to executing a window splitter in a single process. Note however that a window splitter of a single process would not be able to achieve the stream rate of parasplit.

4. Evaluating parasplit

The purposes of the experiments are the following:

- Validate the heuristic for parasplit.
- Validate that parasplit is network bound in practice.
- Evaluate the efficiency of parasplit by measuring the CPU cost overhead of parallelizing *rfn* and *bfn* in parasplit.
- Show that parasplit allows one order of magnitude more expressways over previous work in an LRB implementation.

In all experiments, each SP is a UNIX process on a cluster of compute nodes, each node featuring two quad-core Intel® Xeon® E5520 CPUs @ 2.27GHz and 8 MB L2 cache. For the scale-up experiments, a maximum of 70 such compute nodes were available. TCP was used for stream communication between SPs. All SPs of parasplit were distributed over different compute nodes in this cluster. Thus, the capacity of the 1 Gbps network interfaces were the upper bound for all inter-process stream rates in the experiments.

As a test stream, we use the input stream of event tuples *e* of LRB. There are four kinds of events; *P*, *A*, *D*, and *E*, of which 99% are position reports *P* that are emitted from vehicles travelling on the expressways numbered from 0 to L–1 . The rest of the tuples are account balance queries *A* (0.5%), daily expenditure queries *D* (0.1%), and estimated travel time queries *E* (0.4%). Our LRB implementation scsq-plr [26], parallelizes the execution by distributing the input stream events per expressway, i.e. $q = L$. In our experiments, input events of type *D* and *E* are omitted, so o = 0.5%. Type *P* events are routed to the query processors PQ_j executing the corresponding expressway j=0…L–1, so r = 99%. Type *A* events are broadcasted, so b = 0.5%. The input stream is split according to *rfnLR*(*e*, *q*) defined as `select expressway(e) where eventtype(e)=P`, and *bfnLR*(*e*) is defined as `select eventtype(e)=A`. In order to measure maximum input stream rate, the LRB input events were streamed at maximum possible rate.

4.1 Window router stream rate

The goal of this experiment is to confirm that *PR* is not CPU bound but network bound for sufficiently large window size. In the experiment, one SP was executing *PR*, which received and routed an input stream of physical windows of size *W* to *p* window splitters with only consume operators installed. *p* was varied from 4 to 512. Figure 4 shows Φ_{PR} for different *p* when varying *W* from 72 bytes to 16kB.

The maximum stream rate is 980 Mbps, achieved for p ≤ 64. A slightly lower maximum stream rate of 975 Mbps was measured for p = 128. For higher values of *p*, the performance degrades for unknown reasons.

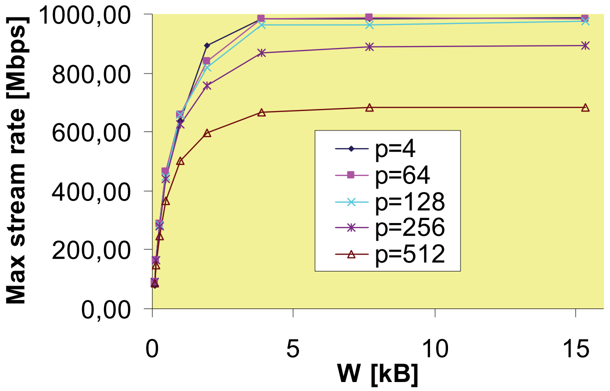


Figure 4. Φ_{PR} for different W, p.

To alleviate the degradation in stream rate when scaling p, we made an experiment with a two-level tree of window routers with equal fanout $\sqrt{p}$ on each level, and W = 16 kB. Figure 5 shows that the degradation then becomes negligible even for very large values of p. Thus, a *PR tree* has higher Φ_{PR} for high values of p than a single process *PR*.

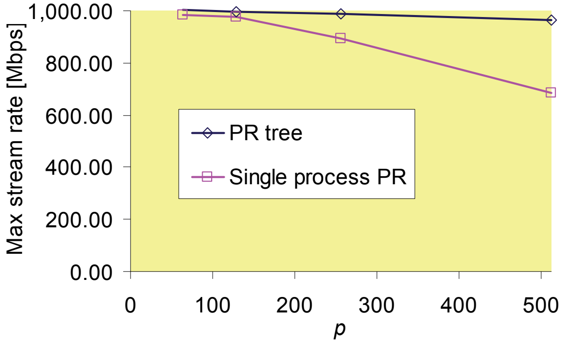


Figure 5. Φ_{PR} for different p.

4.2 Parasplit scale-up

The scale-up is defined as $\Phi_{PARASPLIT}$ when q is increased. In the scale-up experiments, we measure $\Phi_{PARASPLIT}$ when varying q and setting p according to the heuristic given in Section 3.2.2. $\Phi_{PS}^{(1)}$ was measured to 123.7 Mbps, and Φ_D was set to 1 Gbps. The scale-up of the heuristic parasplit using the approximate Equation (6) was compared to the scale-up according to the cost model given by Equations (2) and (3). The values of cr_W, cs, and ce were obtained by detailed profiling of one *PR* node and of one PS_i node executing *rfnLR*() and *bfnLR*(). For reference, we also measure the scale-up of parasplit with p = 1 and with $p = q$. For p = 1, all stream splitting is performed in a single process, i.e. the naïve *fsplit* [32], which is the baseline for the experiments. To compare with the so far best published stream splitting strategy, we also measure the scale-up of *maxtree* [32]. Based on knowledge of communication costs, and b and r, *maxtree* forms an optimized tree of split-stream processes, where each process splits the input stream according to *rfn* and *bfn*. The maximum stream rate of maxtree is sensitive to the cost of *rfn* and *bfn*, a limitation not present in parasplit. To make maxtree fully comparable, its implementation is slightly improved over [32] by reading physical windows of the input stream rather than individual tuples.

Figure 6 shows that parasplit achieves an order of magnitude higher maximum stream rate than *maxtree* and naïve *fsplit* (p = 1) for high values of q. The single *PR* measurements have a single process window router, whereas the *PR* tree measurement employs a tree of window routers, as devised in Section 4.1. It is clear that p must be chosen carefully, since parasplit with neither p = 1 nor $p = q$ does scale. As predicted by Equation (3), p = 1 does not scale with q.

In the *single PR* experiments, 849 Mbps was measured for q = 512 and the heuristic setting of p = 55 in Equation (6), whereas 840 Mbps was achieved for q = 512 and the cost model setting of p = 44 in Equations (2) and (3). The scale-up of heuristic parasplit (*parasplit, single PR*) is the same as that of parasplit with p chosen using the cost model (*cost model, single PR*). This shows that our heuristics are sound. The best maximum stream rate was achieved using a tree shaped window router (*parasplit PR tree*), confirming the results in Figure 5. In particular, for q = 512, *PR tree* achieves a maximum stream rate of 913 Mbps (p = 55 as set by the heuristic in Equation (6)).

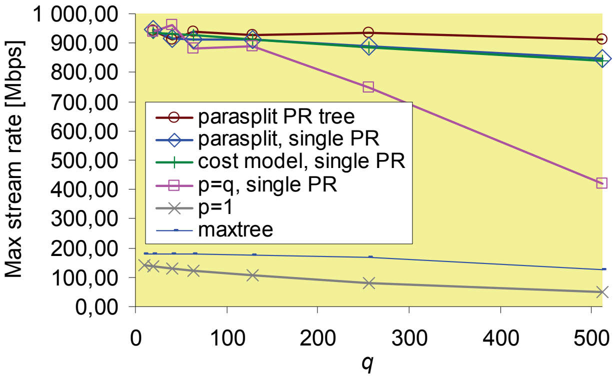


Figure 6. Scale-up.

4.3 Parasplit efficiency

The purpose of this experiment is to measure the CPU overhead that parasplit incurs when parallelizing *rfn* and *bfn*. The total CPU time of each SP was measured using system performance counters in the `/proc` file system. Parasplit was invoked with a dummy query *Q* that only counted the incoming tuples. The cost *O* of this simple query was subtracted from C_{PQ} before η was computed using Equation (7). The same experiments were performed as in Section 4.2 except for *maxtree*.

Figure 7 shows the efficiency when increasing *q*. As expected, the exact cost model based setting of *p* (*cost model, single PR*) has the highest efficiency. However, we notice that the efficiency of the heuristic parasplit variants (*parasplit, single PR* and *parasplit, PR tree*) is very close to that of the cost model. Finally, we notice that the efficiency goes down with bad choices of p (*p*=1, *p*=*q*).

Substantially over-estimated $p = q$ is particularly bad, since the poll and merge costs in the query nodes are then multiplied by *p* in Equation (4). We conclude that *p* should be set to the recommended heuristic value, and that a *PR tree* should be used in parasplit for all values of *p* and *q*, as *parasplit PR tree* achieves superior scale-up and does not degrade efficiency substantially compared to any of the *single PR*.

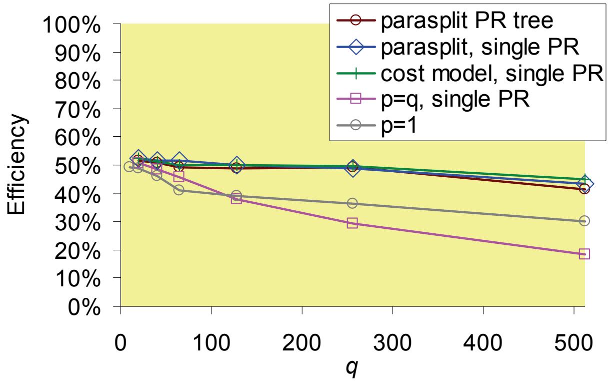


Figure 7. Parasplit efficiency for increasing *q*.

4.4 LRB experiment

As a final experiment, we compare the achievable stream rate of *scsq-plr* using parasplit to other implementations of LRB [3]. The number of expressways that an implementation is able to handle is called the *L-rating* of the implementation. An LRB implementation produces five result streams; toll and accident alerts (event type *T* and *AA*), and query responses (event type *A*, *D*, and *E*). Currently, *E* tuples are ignored in all LRB implementations [3]. The *D* tuples are computed over data that does not change during the LRB simulation. In an experiment performed after the publication [32], we verified that a conventional database on a single compute node was sufficient to handle queries over historical data (event type *D*) for an L-rating up to 64. However, the conventional DBMS cannot handle the very high query rates presented here. A solution would involve scaling out the historical database over many compute nodes, which is future work. In the present experiment, we choose to ignore the *D* tuples. As a consequence, the implementation used here results in three output streams (event type *T*, *AA*, and *A*).

Parasplit was used to split the input stream in *scsq-plr* according to Figure 8, using only a single process PR. The input stream rate for each expressway in LRB is maximum 1700 tuples/s. The size of each tuple is 72 bytes, so the input stream rate will be $\Phi_D = 1700 \cdot 512 \cdot 72 \cdot 8$ Mbps ≈ 500 Mbps. Given $q = 512$ and $\Phi_D = 500$ Mbps, parasplit determines $p = 25$ according to Equation (6), as $\Phi_{PS}{}^{(1)} = 123.7$ Mbps.

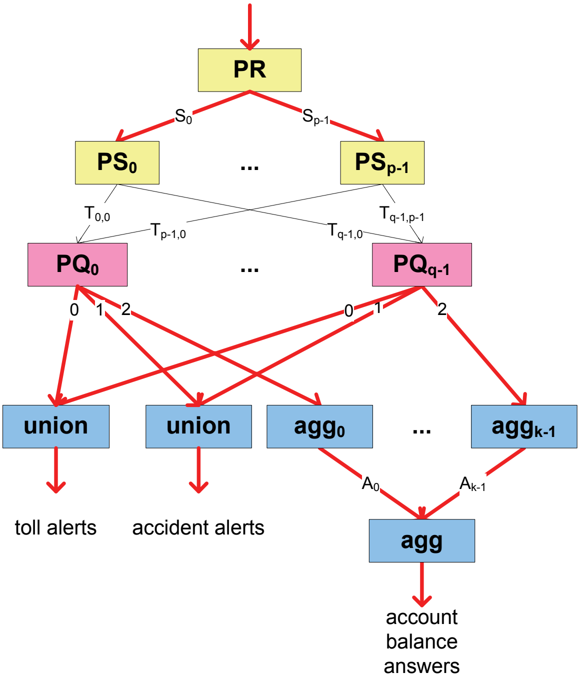


Figure 8. scsq-plr using parasplit.

The expensive continuous query *Q* is here the computation of the LRB query result streams. Each PQ_j node was processing all tuples of expressway *j*. The output stream of each lr_j is split on event type according to the routing function *rfnO*(*e*) defined as `select eventtype(e);` where event type *T* is 0, *AA* is 1 and *A* is 2 in the output stream.

The toll and accident alert result streams are merged using stream union-all (i.e. ignoring timestamp order). Account balance answers from all lr_j are grouped on query id, and the sums of the account balances from all expressways lr_j are aggregated for each query id.

Aggregating *q* streams of account balance responses with a total stream rate of $q \cdot \Phi_a$ results in an output stream rate of Φ_a after the aggregation. In *scsq-plr* with q = 512, the total stream rate of account balance answers is much greater than the capacity of the 1 Gbps network interfaces used in the experiments. Similar to aggregation trees of [31], account balance answers are hierarchically aggregated in two level tree as shown in Figure 8, with k = 22 ≈ √512 for the two-level distributed aggregation tree of 512 input streams. Finally, all union and aggregation processes were scheduled on different compute nodes, so that disk and network throughput for these processes was no bottleneck.

LRB requires a maximum response time (MRT) of 5 seconds for events of type *T*, *AA*, and *A*. In our LRB experiment we measured the maximum response time for all events *e* in the output stream to be MRT(*e*) < 5 s. Thus, we conclude that *scsq-plr* using parasplit achieves an L-rating of 512, with daily expenditure queries disabled. Table 1 lists the currently published LRB implementations.

Table 1. *LRB implementations.*

Name	Year	L	#cores	Comment
Aurora [3]	2004	2.5	1	
SPC [19]	2006	2.5	170	3 GHz Xeon
XQuery [6]	2007	1.5	1	
scsq-lr [26]	2007	1.5	1	laptop
DataCell [23]	2009	1	4	1.4s average response time
stream schema [13]	2010	5	4	
scsq-plr [32]	2010	64	48	*maxtree*
CaaaS [9]	2011	1	2	Streaming MapReduce
scsq-plr	2011	512	560	Parasplit. *D* disabled

5. Related work

This paper complements other work on parallel DSMS implementations [1] [11] [14] [16] [18] [24] [29] [32], by employing massive scale-out based on customizable stream splitting functions.

The fragmentation and replication conditions provided as meta-data in a distributed database [25] corresponds to the routing and broadcast functions in parasplit. While the emphasis of distributed databases is scaling out data, the extreme stream rates for DSMSs require scaling out also the routing and broadcast functions, which is the topic of this paper.

In previous work [15], we have shown that stream splitting, utilizing non-trivial routing decisions, proved to be very efficient when parallelizing online spatio-temporal optimization of transportation. Splitstream functions were introduced in [32], where tree-shaped distributed execution plans were shown to improve the rate of stream splitting. These techniques enabled an L-rating of 64 in LRB. However, the splitstream trees developed in [32] were sensitive to the cost of *rfn* and *bfn*. By contrast, we have shown that parasplit achieves an order of magnitude higher stream rates independent of the cost of *rfn* and *bfn* by parallelizing the stream splitting in a lattice.

GSDM [18] distributed its stream computations by generating parallel execution plans with tree shaped stream splitting, through parameterized code generators. Parallelizing the queries in GSDM was reported to achieve a maximum stream rate of 16 Mbps. By contrast, parasplit is network bound by utilizing physical windows and a lattice based stream splitting strategy and allows not only routing but also broadcasting of tuples.

SPADE [14] has a stream splitting operator that includes capabilities of replicating tuples [2], similar to *splitstream*. StreamInsight [21] has both stream splitting operator and a broadcast operator that replicates entire streams to multiple processing operators, similar to a publish/subscribe-system. By contrast, parasplit allows fine grained customized specification of what individual tuples in the stream to broadcast or route. The throughput of distribution and replication in System S was reported to degrade with the number of output nodes in [2]. Custom stream partitioning was also shown to be a bottleneck in [7]. By contrast, we have shown that parasplit provides network bound stream processing by massive scale-out of customized splitting and broadcasting in a lattice shaped distributed execution plan.

Gigascope [11] was extended with automatic query dependent data partitioning in [20] for computing aggregates in high-volume network monitoring queries, distributed over the output from special hardware splitting a very high volume input stream. The evaluation focused on aggregation of a number of input streams, each with a stream rate of 200 Mbps. The input stream splitting was outside the scope of their work, as it was assumed to be performed by special hardware. By contrast, our work focuses on stream splitting in software rather than hardware, scaling up to network stream rate by parallelizing the stream splitting on standard PCs.

Partitioning a query plan by statically distributing the execution of its operators proved to be a bottleneck in SPC [19]. In Medusa [5], query plans were partitioned by dynamically migrating operators between processors. However, expensive operators are still bottlenecks. In our work, such bottlenecks are eliminated by both splitting the input stream into several parallel streams, and by parallelizing the stream splitting itself. Furthermore, allowing combined routing and broadcasting in parasplit provides a powerful method for data parallelization.

Of the existing implementations of LRB shown in Table 1, there are three attempts to parallelize the execution: SPC [19], Stream Schema [13], and Continuous analytics as a Service (CaaaS) [9]. Unlike these systems, parasplit provides massive scale-up by automatic parallelization of *rfn* and *bfn*, enabling network bound input stream rates independent of the cost of parallelization.

The SPC implementation of LRB was partitioned into 15 processing elements, each of which executed a separate stream operator. The operator that computed all segment statistics became a hot spot. By contrast, parasplit uses data parallelism rather than operator parallelism, and is shown to achieve over 100 times the number of expressways of the SPC implementation. This performance difference illustrates the usefulness of customizable data parallelization provided by parasplit.

Automatic parallelization of stream queries based on user provided stream metadata was discussed in [13], where a parallelized implementation of LRB was shown to achieve L = 5 on a 4-core PC. By contrast, in parasplit, metadata is expressed as queries in *rfn* and *bfn*.

The use of physical windows called SigSegs in XStream [16] was shown to reduce tuple passing overhead substantially. Similarly, we also save communication cost by operating on physical windows of stream events in the window router of parasplit. While entire SigSegs were distributed in XStream, parasplit allows massive parallelization based on hierarchical window routing and parallelized customized distribution and replication of tuples. This is shown to maintain network bound stream rates independent of the cost of splitting.

Recently [28], event detection using regular expressions was implemented on an FPGA, which achieved gigabit wire speed. By contrast, parasplit allows parallelization of arbitrary CQs in software with no need for special hardware.

MapReduce [12] can be seen as a form of parallelized group-by over large data sets. Dryad [17] allows more flexible parallelization schemes by implementing an explicit process graph building language. By contrast, SCSQ does not require the user to explicitly construct process graphs, since the process graphs of SCSQ are automatically generated by the parallelization functions. SCOPE [8] and Map-Reduce-Merge [30] provide an SQL-like query language over large distributed files. However, Dryad, Map-Reduce, and SCOPE are all batch systems, operating on data at rest (sets), while SCSQ continuously processes streaming data. MapReduce was recently extended with streaming capabilities [9] [10]. The problem of scalable stream splitting is not handled by streaming MapReduce.

A MapReduce wrapper was recently added to the DSMS System S [22], combining data at rest with streaming data. However, calling MapReduce from a DSMS is different from scaling out the execution of a DSMS, which is the focus of this paper.

6. Conclusions and future work

Scalable splitting of streams is necessary to achieve high stream rates in a parallel DSMS. We have introduced parasplit, which enables splitting input streams of high volume into a high number of output streams by parallelizing user defined stream splitting specifications. Parasplit is shown to enable a network bound stream rate independent of communication protocol (e.g. 93% of a 1 Gbps interface) for parallelization of expensive continuous queries over streams. This is achieved by (i) automatic parallelization of the execution of the stream splitting specifications, and (ii) by hierarchically routing of physical windows of sufficient size. Based on a cost model, we devised a heuristic that automatically chooses physical window size and parallelization of the stream splitting specifications for close-to-optimum efficiency according to the cost model. By scaling out stream splitting with parasplit in the *scsq-plr* implementation of the Linear Road Benchmark, we achieved an order of magnitude higher stream processing rate over previously published results, allowing 512 expressways.

As future work, we plan to investigate alternatives for scaling out a parallel database to combine high volumes of data at rest with high volumes of data in motion. Furthermore, it should be investigated how to push down selection predicates of Q into *rfn*, effectively saving communication cost by increasing omit percentage o in the window splitters.

Our experiments have been performed in a cluster of up to 70 compute nodes with 8 cores each connected by a 1 Gbps switched network. The behavior of parasplit should be investigated for higher network speeds, more cores, and more compute nodes.

It should be investigated if the efficiency of parasplit can be improved by using hardware acceleration such as FPGAs, by comparing the costs of hardware accelerated parasplit to that of standard hardware parasplit.

The query plan of parasplit is optimized, parallelized, and scheduled when the CQ is started. Although this approach was shown to work well in our evaluations, it would be worthwhile to extend it with methods for adaptive parallelization and scheduling of execution over streams after the CQ has been started, as in [24] [27] [29] [33]. For example, it should be investigated if p and W can be set adaptively based on system load.

Acknowledgements

This project is supported by VINNOVA, grant 2007-02916 and the Swedish Foundation for Strategic Research, grant RIT08-0041. The experiments were performed using UPPMAX resources. UPPMAX staff is acknowledged for assistance. We appreciate the insightful comments by the anonymous reviewers.

References

1. Ali, M.H. et al. Microsoft CEP server and online behavioral targeting. *Proc. VLDB 2009*.
2. Andrade, H., Gedik, B., Wu, K.-L., Yu, P. S. Scale-Up Strategies for Processing High-Rate Data Streams in System S. *Proc. ICDE 2009*.
3. Arasu A. et al. Linear Road: A Stream Data Management Benchmark. *Proc. VLDB 2004*.
4. Bai, Y., Thakkar, H., Wang, H., and Zaniolo, C. Optimizing Timestamp Management in Data Stream Management Systems. *Proc. ICDE 2007*.
5. Balazinska, M., Balakrishnan, H., Stonebraker, M. Contract-Based Load Management in Federated Distributed Systems. *Proc. NSDI 2004*.
6. Botan I. et al. Extending XQuery with Window Functions. *Proc. VLDB 2007*.
7. Brenna, L., Gehrke, J., Hong, M., Johansen, D. Distributed event stream processing with non-deterministic finite automata. *Proc. DEBS 2009*.
8. Chaiken, R., et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB 2008*.
9. Cheng, Q., Hsu, M., Zeller, H. Experience in Continuous analytics as a Service (CaaaS). *Proc EDBT 2011*.
10. Condie, T., et al. Online aggregation and continuous query support in MapReduce. *Proc. SIGMOD 2010*.
11. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V. Gigascope: A Stream Database for Network Applications. *Proc. SIGMOD 2003*.
12. Dean, J., Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI 2004*.
13. Fischer, P.M., Esmaili, K.S., and Miller, R.J. Stream schema: providing and exploiting static metadata for data stream processing. *Proc. EDBT 2010*.
14. Gedik, B., Andrade, H., Wu, K.-L., Yu, P.S., and Doo, M. SPADE: the system s declarative stream processing engine. *Proc. SIGMOD 2008*.
15. Gidofalvi, G., Pedersen, T. B., Risch, T., Zeitler, E. Highly scalable trip grouping for large-scale collective transportation systems. *Proc. EDBT 2008*.
16. Girod, L., et al. XStream: A Signal-Oriented Data Stream Management System. *Proc. ICDE 2008*.
17. Isard, M., et al. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, Volume 41, 59–72, 2007.

18. Ivanova, M., Risch, T. Customizable Parallel Execution of Scientific Stream Queries. *Proc. VLDB 2005*.
19. Jain, N., et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. *Proc. SIGMOD 2006*.
20. Johnson, S., Muthukrishnan, Shkapenyuk, V., Spatscheck, O. Query-Aware Partitioning for Monitoring Massive Network Data Streams. *Proc. SIGMOD 2008*.
21. Kazemitabar, S.J. et al. Geospatial stream query processing using Microsoft SQL Server StreamInsight. *Proc. VLDB 2010*.
22. Kumar, V., Andrade, H., Gedik, B., Wu, K.-L. DEDUCE: at the intersection of MapReduce and stream processing. *Proc. EDBT 2010*.
23. Liarou, E., Goncalves, R., Idreos, S. Exploiting the Power of Relational Databases for Efficient Stream Processing. *Proc. EDBT 2009*.
24. Liu, B., Zhu, Y., Jbantova, M., Momberger, B., and Rundensteiner, E.A. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. *Proc. VLDB 2005*.
25. Özsu, M.T., Valduriez, P. Principles of Distributed Database Systems, Second Edition. Prentice-Hall (1999)
26. SCSQ-LR home page. http://user.it.uu.se/~udbl/lr.html, February 2011.
27. Shah, M.A., Hellerstein, J. M., Chandrasekaran, S., Franklin, M. J. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. *Proc. ICDE 2002*.
28. Woods, L., Teubner, J., Alonso G. Complex Event Detection at Wire Speed with FPGAs. *Proc. VLDB 2010*.
29. Xing, Y., Zdonik, S., Hwang, J.-H. Dynamic Load Distribution in the Borealis Stream Processor. *Proc. ICDE 2005*.
30. Yang, H., Dasdan, A., Hsiao, R.-L. Parker, D.S. Map-reduce-merge: simplified relational data processing on large clusters. *Proc. SIGMOD 2007*.
31. Yu, Y., Gunda, P.K., Isard, M. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. *Proc. 22nd ACM Symposium on Operating Systems Principles*, 2009.
32. Zeitler, E. and Risch, T. Scalable Splitting of Massive Data Streams. *Proc. DASFAA 2010*.
33. Zhou, Y., Aberer, K., and Tan, K.-L. Toward massive query optimization in large-scale distributed stream systems. *Proc. Middleware 2009*.