Massive Scale-out of Expensive Continuous Queries

Erik Zeitler
Department of Information Technology
Uppsala University
+46 18 471 3390
erik.zeitler@it.uu.se

Tore Risch
Department of Information Technology
Uppsala University
+46 18 471 6342
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. Scalable execution of such continuous queries with expensive computations requires input streams to 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 massive parallelization of query operators.

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 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* function 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.

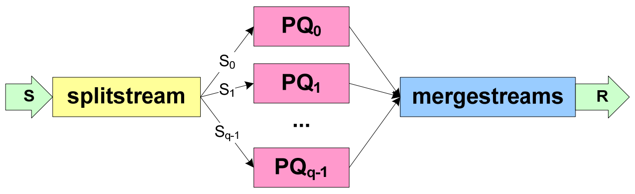

Figure 1. Streamed data parallelism.

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 stream 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 11.
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

bound stream rates, independent of the stream splitting cost 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 large enough 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.

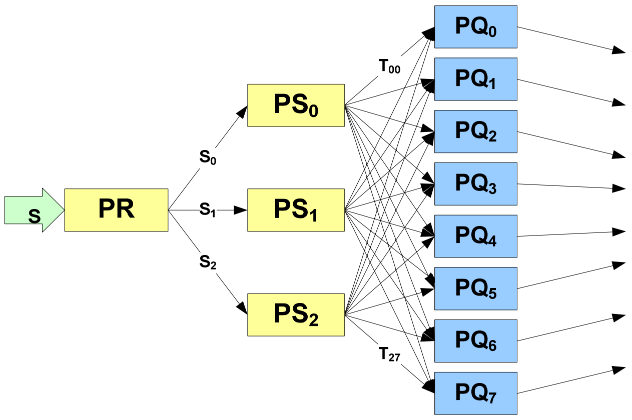

Figure 2. Parasplit.

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, $j = 0\ldots u–1$, is first read and demarshalled 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, $i = 0\ldots q–1$. 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.

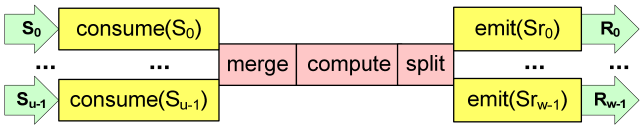

Figure 3. Modules and operators in a stream process.

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).

$$C = cr + (cp + cm)\cdot u + cq + \sigma\big(cs(o + r + q\cdot b) + ce(r + q\cdot b)\big) \quad (1)$$

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.

$$C_{PR} = cr_W + cs_W + ce_W \quad (2)$$

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.

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

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.

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

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 $\Phi_{PARASPLIT} = \min(\Phi_{PR}, \Phi_{PS}, \Phi_{PQ})$. 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 $\Phi_{PR} = 987$ 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 $p\cdot\Phi_{PS} \geq \Phi_D$ 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):

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

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.

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

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, $O = 0$.

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

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 $p \leq 64$. 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.

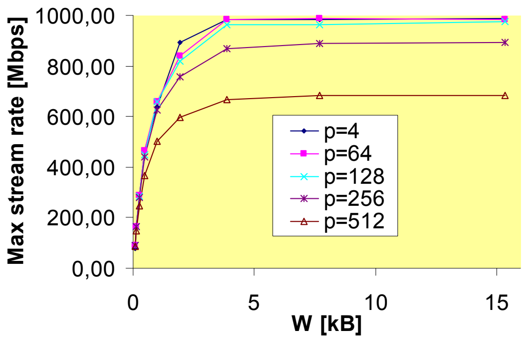

Figure 4. Φ_{PR} for different *W*, *p*.

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*.

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 splitstream 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.

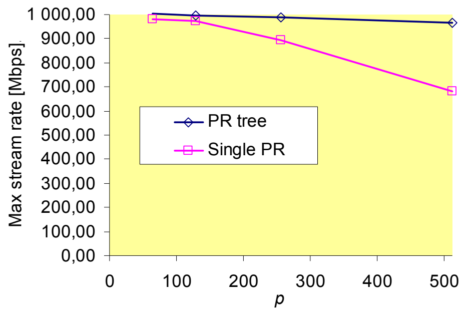

Figure 5. Φ_{PR} for different *p*.

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*.

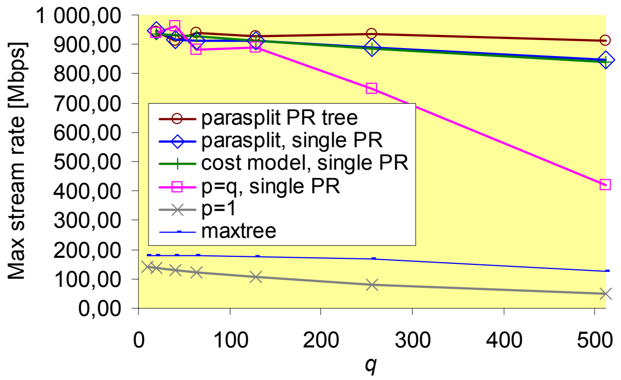

Figure 6. Scale-up.

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)).

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 **Error! Reference source not found.** 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*.

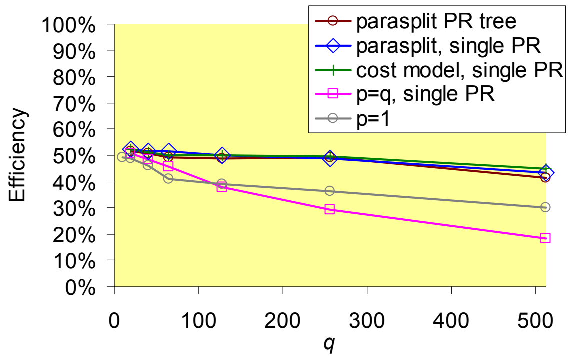

Figure 7. Parasplit efficiency for increasing *q*.

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 (MySQL) 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.

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.

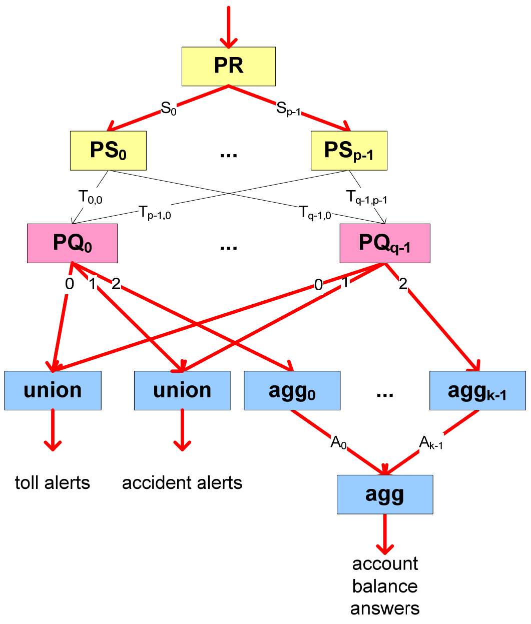

Figure 8. scsq-plr using parasplit.

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 $k = 22 \approx \sqrt{512}$ 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. 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	3GHz Xeon
XQuery [6]	2007	1.5	1	
scsq-lr [26]	2007	1.5	1	laptop
DataCell [22]	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 metadata 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 [19] 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.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] Ali, M.H., et al. Microsoft CEP server and online behavioral targeting. *Proc. VLDB 2009*, 1558–1561.

[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*, 1375–1378.

[3] Arasu A., et al. Linear Road: A Stream Data Management Benchmark. *Proc. VLDB 2004*, 480–491.

[4] Bai, Y., Thakkar, H., Wang, H., and Zaniolo, C. Optimizing Timestamp Management in Data Stream Management Systems. *Proc. ICDE 2007*, 1334–1338.

[5] Balazinska, M., Balakrishnan, H., Stonebraker, M. Contract-Based Load Management in Federated Distributed Systems. *Proc. NSDI 2004*, 197–210.

[6] Botan I., et al. Extending XQuery with Window Functions. *Proc. VLDB 2007*, 75–86.

[7] Brenna, L., Gehrke, J., Hong, M., Johansen, D. Distributed event stream processing with non-deterministic finite automata. *Proc. DEBS 2009*, 3:1–3:12.

[8] Chaiken, R., et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB 2008*, 1265–1276.

[9] Cheng, Q., Hsu, M., Zeller, H. Experience in Continuous analytics as a Service (CaaaS). *Proc EDBT 2011*, 509–514.

[10] Condie, T., et al. Online aggregation and continuous query support in MapReduce. *Proc. SIGMOD 2010*, 1115–1118.

[11] Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V. Gigascope: A Stream Database for Network Applications. *Proc. SIGMOD 2003*, 647–651.

[12] Dean, J., Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI 2004*, 107–113.

[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*, 207–218.

[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*, 1123–1134.

[15] Gidofalvi, G., Pedersen, T. B., Risch, T., Zeitler, E. Highly scalable trip grouping for large-scale collective transportation systems. *Proc. EDBT 2008*, 678–689.

[16] Girod, L., et al. XStream: A Signal-Oriented Data Stream Management System. *Proc. ICDE 2008*, 397–406.

[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*, 157–168.

[19] Jain, N., et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. *Proc. SIGMOD 2006*, 431–442.

[20] Johnson, S., Muthukrishnan, Shkapenyuk, V., Spatscheck, O. Query-Aware Partitioning for Monitoring Massive Network Data Streams. *Proc. SIGMOD 2008*, 1135–1146.

[21] Kazemitabar, S.J., et al. Geospatial stream query processing using Microsoft SQL Server StreamInsight. *Proc. VLDB 2010*, 1537–1540.

[22] Kumar, V., Andrade, H., Gedik, B., Wu, K.-L. DEDUCE: at the intersection of MapReduce and stream processing. *Proc. EDBT 2010*, 657–662.

[23] Liarou, E., Goncalves, R., Idreos, S. Exploiting the Power of Relational Databases for Efficient Stream Processing. *Proc. EDBT 2009*, 323–334.

[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*, 1338–1341.

[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*, 25–36.

[28] Woods, L., Teubner, J., Alonso G. Complex Event Detection at Wire Speed with FPGAs. *Proc. VLDB 2010*, 660–669.

[29] Xing, Y., Zdonik, S., Hwang, J.-H. Dynamic Load Distribution in the Borealis Stream Processor. *Proc. ICDE 2005*, 791–802.

[30] Yang, H., Dasdan, A., Hsiao, R.-L. Parker, D.S. Map-reduce-merge: simplified relational data processing on large clusters. *Proc. SIGMOD 2007*, 1029–1040.

[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, 247–260.

[32] Zeitler, E. and Risch, T. Scalable Splitting of Massive Data Streams. *Proc. DASFAA (2) 2010*, 184–198.

[33] Zhou, Y., Aberer, K., and Tan, K.-L. Toward massive query optimization in large-scale distributed stream systems. *Proc. Middleware 2009*, 326–345.