Scalable Splitting of Massive Data Streams

Erik Zeitler, Tore Risch

Department of Information Technology
Uppsala University
Sweden
erik.zeitler@it.uu.se
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 *split-stream 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. Fig. 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.

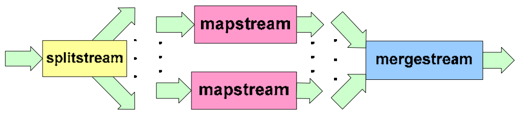

Fig. 1. Splitstream, mapstreams, and mergestream.

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()*.

Fig. 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.

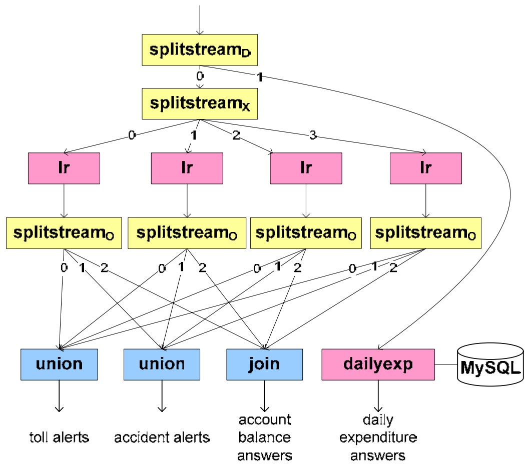

Fig. 2. The parallelization strategy in *scsq-plr*. $L = 4$.

2.2 Single Process Splitstream

A splitstream function is naïvely implemented by a single process splitstream operator *fsplit*, its modules being shown in Fig. 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…w–1.

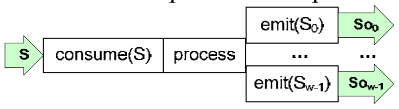

Fig. 3. Modules of *fsplit*.

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

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

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 $So_0…So_{w-1}$, 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…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	$\Phi_X = 99.5\% \cdot \Phi$	$\Phi_D = 0.1\% \cdot \Phi$
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*.

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

The denominator $\lambda_{\ell\text{-}1}$ 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

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

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 Fig. 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.

$$\Phi_{\max} = \frac{E}{\max\limits_{\ell}(C_\ell)}. \tag{7}$$

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 $\lambda_{\ell\text{-}1}$.

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

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 $w = 64$. Fig. 4 shows the estimated performance.

In the scale-up evaluation, the estimated throughput was plotted in Fig. 4 (a) as the percentage of emit capacity E. The estimated total CPU cost was plotted in Fig. 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*.

When scaling *b* in the robustness evaluation, Fig. 4 (d) shows that the CPU cost of *maxtree* increases sharply when *b* increases. If $b \approx 1$ in to 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. Fig. 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*.

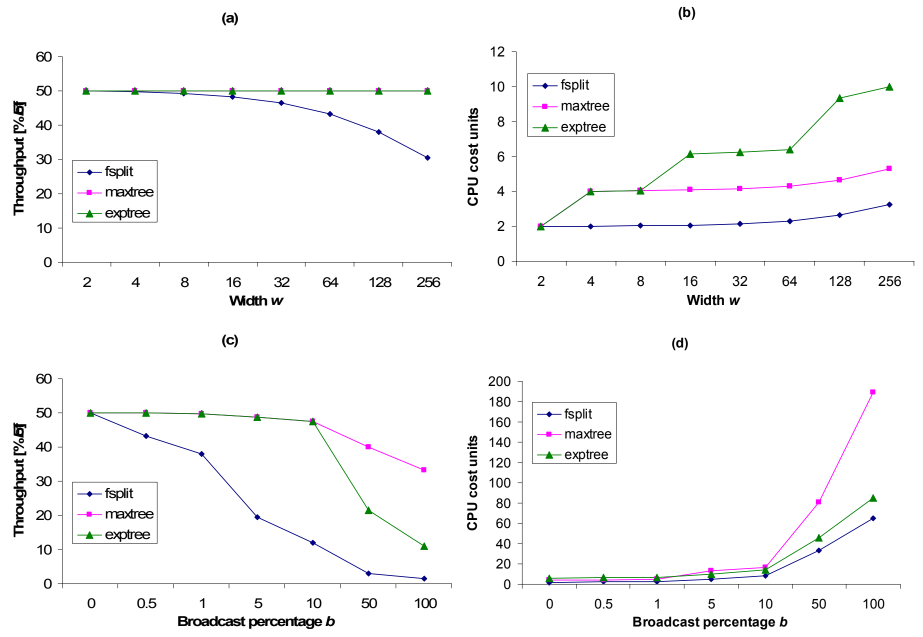

Fig. 4. (a) Estimated throughput and (b) total CPU cost, $b = 0.5\%$. (c) Estimated throughput and (d) total CPU cost, $w = 64$.

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 Fig. 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*.

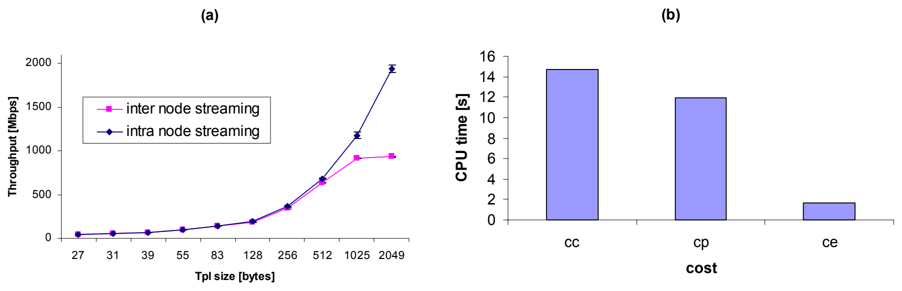

Fig. 5. (a) Inter and intra node emit capacity, (b) CPU time breakdown for *fsplit* with $w = 1$.

The CPU times obtained from the *fsplit* SP are shown in Fig. 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 $L > 65$.

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.

Fig. 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 Fig. 4 with one exception: The measured throughput of *maxtree* shown in Fig. 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.

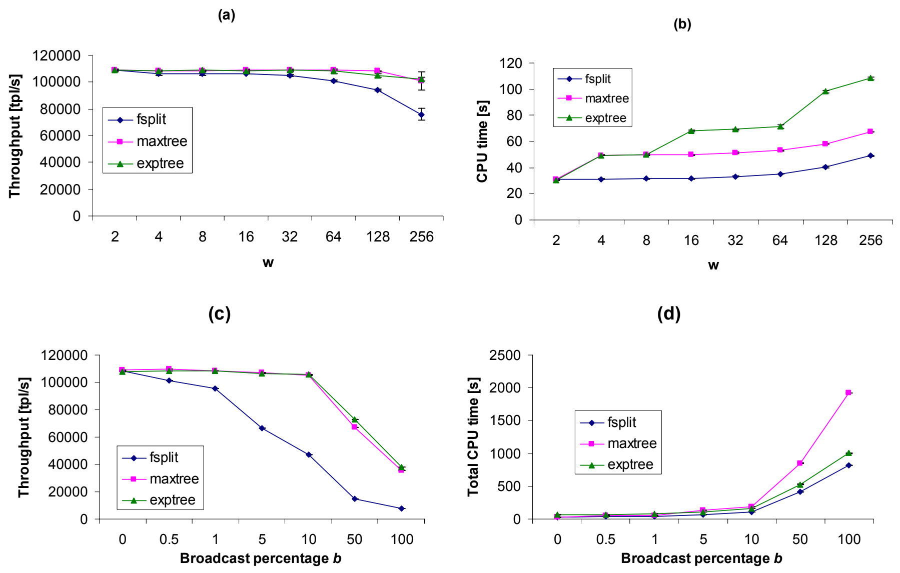

Fig. 6. (a) Measured throughput and (b) measured total CPU cost for $b = 0.5\%$. (c) Measured throughput and (d) measured total CPU cost for $w = 64$.

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 Fig. 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 Fig. 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 Fig. 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.

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 Fig. 4 (a) and experimentally in Fig. 6 (c), the gain will be bigger if the broadcast percentage b is greater.

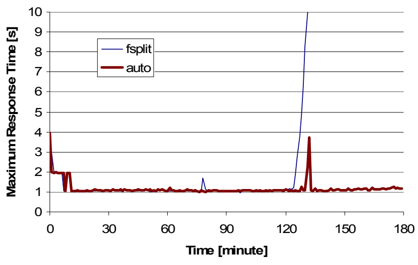

Fig. 7. Maximum response time for $L = 64$.

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 anti-correlations 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 re-scheduling 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].

Acknowledgments. 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.

9 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 supercomputer. 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)