# Efficient Time Aggregation and Querying of Flashed Streams in Constrained Motes

Pedro Furtado

Univeristy of Coimbra,
Portugal
pnf@dei.uc.pt

**Abstract.** We propose and evaluate efficient, low-memory and low-consumption organization and query processing algorithms for a tiny Stream Management Engine (SME). The target sensor devices have low memory and computation capabilities, and high wireless data transmission costs. The SME represents data as streams, we discuss the approach and study how to optimize group-by aggregation over time-ordered data in that context, and to provide simple all-purpose group-by and join algorithms. We used an experimental testbed to evaluate the findings and prove the advantage of the alternatives and studies that we made.

## 1 Introduction

Low-cost autonomous wireless sensing devices can be deployed to collect sensor data, either logging it for later retrieval or sending it wirelessly, possibly in multi-hop fashion, to some computerized systems. Applications of such systems include environmental, medical, industry, smart buildings, warehouse tracking, transport logistics and surveillance. The devices themselves, a.k.a motes, have computational and wireless communication capabilities, being able to sense, store, read and route the data along one or more hops, and to interface with a collection device, typically through wireless or USB connections. Operation is frequently supported on top of a tiny operating system such as TinyOS [26] or Contiki [28]. Motes also have their own battery power source that provides autonomy and mobility. An example of a mote is the TelosB [25], with up to 48KB of code memory, which must fit the OS and applications, and up to 10KB or data memory.

Most mote designs come with support for an external flash memory with MB or GB of capacity, which enables logging and storage of large quantities of data. In that context, consider streams of sensed data being acquired and logged into the flash. We build a tiny Stream Management Engine (SME) that allows users to submit queries for the data. It does the usual relational algebra data manipulations of data management systems, such as selections, projections, aggregations, joins. The SME must fit into the tiny code base and be efficient. Questions that immediately pops-up are: whether a stream management system fits well into the memory and computational constraints of motes; whether it has acceptable performance? Are there advantages over pushing or pulling detailed log data to a collecting PC?

We propose and evaluate compact and efficient algorithms, in particular time-ordered group-by aggregations (GBTime) over time-ordered storage, as well as more generic group-by and join algorithms over the tiny devices. In the process, we provide answers to the above questions.

Efficiency is measured according to three major metrics: code size, since it must fit constrained devices, query execution time, and energy consumed to execute the queries. Query execution efficiency guarantees that clients will have their responses quick enough, and consumed energy is paramount, since low consumption avoids frequent battery replacement, which is very undesirable in many practical deployments.

We propose and test the mechanisms for stream-based data management with the aggregation and joins over stream data, and we present the SME and query processing approach. The code size, efficiency and energy savings are evaluated experimentally, showing that, in spite of the very small data memory, it is advantageous to be able to store and query locally in the motes, reducing the amount of data that needs to be sent to PC. We also show that algorithms such as time-ordered aggregation have major performance and energy saving advantages.

This paper is structured as follows: section 2 reviews related work. Section 3 presents the SME model and shows how it is used to query over sensor networks. Sections 4 and 5 discuss query-processing algorithms, and Section 6 show experimental results. Section 7 concludes the paper.

## 2 Related Work

A mote is a node in a wireless sensor network that is capable of gathering sensory information, performing some computations, and communicating with other connected nodes in the network. For instance, TelosB features a IEEE 802.15.4/ZigBee compliant RF transceiver with integrated onboard antenna, achieving a 250 kbps data rate. It has a 8 MHz TI MSP430 microcontroller with 10kB of RAM and a 1MB external flash for data logging. It also has a programming and data collection USB interface, integrated sensors and interfaces for adding other sensors and actuators. TelosB runs tiny operating systems such as TinyOS [26] or Contiki [28] and is programmed using some C-based dialect that is compiled and loaded with the full code image. TelosB has a line-of-sight reach of about 100 meters. More generically, mote communication ranges go from anywhere between a few meters to kilometers, with a corresponding bill to pay concerning energy consumption. Power autonomy is an important aspect in Motes, since it gives them both breadth of deployment in any place and complete mobility. Communication is by far the costliest part in terms of power consumption. To decrease consumption, communication should be reduced. The radio is turned-on only long enough for the node to be able to participate in multi-hop network communication, if and when necessary. Motes typically have small data memories, but if the data is logged into flash and queries are posed against that flash memory, it is possible to reduce the amount of communication significantly. For instance, in TelosB [25], writing data to flash consumes 15 to 30 times less than sending the same data to another node. This means that logging the data locally and aggregating it are important strategies for the SME.

Writing and reading data from flash is 10 to 100 times faster than exchanging and storing the data from the mote where it was acquired to a computer logging the data. In most cases, the data also needs to go through other nodes in a sensor network before it arrives at a sink node connected to the computer. If the data is logged locally and aggregated efficiently before it is sent over the air, considerable speedup can be achieved. Logging the data locally also results in further autonomy, with the possibility of using the motes as data loggers for longer spans of time. In certain cases, data indexes may reduce operation times significantly. While indexes designed for large databases assume block-based I/O, flash devices have specificities that modify the design, such as access characteristics and restricting modifications of written data. Indexes designed for flash memory include FlashDB [13], Lazy-Adaptive Tree [3], and MicroHash [19]. Because of NAND write restrictions, these approaches use log structures [26], which need large amounts of memory. We provide a simple and efficient approach for time-wise data and queries.

Sensor data management by means of middleware approaches is also related to this work. For clarity, we classify middleware as either intra-sensor network approaches, such as ours, and internet-based sensor data management, which pick sensor data from a sensor data source, then use internet-connected non-constrained computers with full sensor data management engines to integrate, compute and share the data. They do not instrument motes or work inside wireless sensor networks at all.

Intra-sensor network: In [1], the authors share a vision of storage-centric sensor networks where sensor nodes will be equipped with high-capacity and energy-efficient local flash storage, arguing that the data management infrastructure will need substantial redesign to fully exploit the presence of local storage and processing capability in order to reduce expensive communication. There are several works surveying middleware managing data over wireless sensor networks, such as [21, 22, 23]. Intra-network approaches include SQL-based solutions, such as TinyDB [12,8,18], Cougar [5] or PerLa[24]. These approaches provide a database front-end to a sensor network. For example, TinyDB runs a small database-like query engine at a sink node. All the remaining nodes in the WSN load the code that allows them to receive commands from the sink and reply with the data. These approaches do not provide a stream management engine for individual nodes, and do not focus on keeping the data in the nodes for longer.

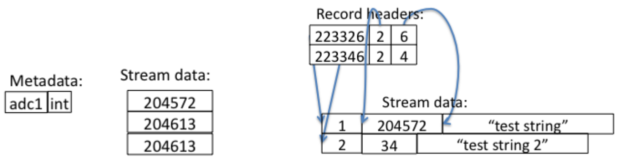
Most other sensor network middleware approaches aim at simplifying programming and deployment, therefore they do not provide a local stream data management engine as ours does. The approaches typically allow users to express computations using a simplified model, or to load pieces of code (agents) for extending functionality. For instance, Kairos [10] offers a network-programming model that allows the programmer to express, in a centralized fashion, the desired global behavior of a distributed computation on the WSN. The Abstract Task Graph (ATaG) [4] is a data driven programming model for end-to-end application development on networked sensor systems. SINA (Sensor Information Networking Architecture) [14, 16] is a middleware architecture that abstracts the network of a sensor node as a distributed object for query operation, and task allocation. Data Service Middleware (DSWare) [11] takes a data-centric approach by defining the common data service and group based service parts of various applications. SensorWare [6] is a general middleware framework based on agent technology, where the mobile agent concept is exploited. Agents migrate to destination areas performing data aggregation reliably.

Internet-based: an example of an internet-based system is GSN [2], whose goal is efficient integration of multiple heterogeneous sensor data sources, with the capability of posing complex queries on the underlying data. GSN gets the streaming data from sensor sources using wrappers, and GSN's stream processing engine (which resides entirely outside of WSNs) is built on top of a relational database engine, storing and retrieving the streaming data during GSN's data processing. Other internet-based middleware architectures include Hourglass [15], HiFi [7] and IrisNet [9], which provide internet-based infrastructures for connecting sensor networks and sensor data sources. Semantic Streams [20] allows users to pose declarative queries over semantic interpretations of sensor data. Other popular internet-based sensor data management systems also include Pachube [29] and SensorCloud [30].

## 3 Overview of Stream Model and Its Use

A Stream Management Engine (SME) is installed in nodes and implements a query processor over ram and flash, and data exchanges between nodes. A stream has a metadata structure that is stored in flash or ram memory and defines attributes. Streams stored in memory are arrays of tuples, while flash-resident streams are stored in files (each stream is stored in one file). One typical use scenario is to log a stream of sensor data into flash and retrieve it later using queries. Another scenario is to acquire sensor data into a window in memory and to send the data to the PC when the window fills-up.

Physical storage on flash has two main possible organizations: constant- and variable-sized tuples. Figure 1 shows the metadata and data corresponding to both alternatives. Although SME can handle both, we focus on the simpler constant-sized tuples for our implementations on motes with small code and data memory.

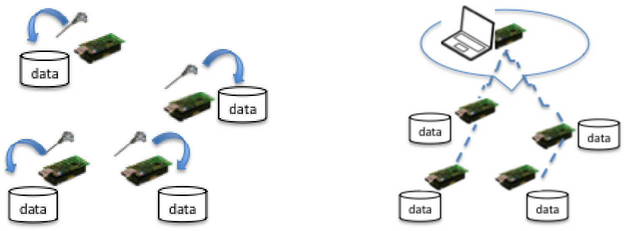


(a) Constant-sized (b) Variable-sized

**Fig. 1.** Metadata and Data: Constant and Variable-sized

If desired, it is possible to store one timestamp value per stream row. However, for streams representing periodic acquisitions (e.g. every minute) it is enough to store the pair (starting timestamp, acquisition rate). The timestamp of every tuples is determined from those. If the stream stops execution and restarts later, a new pair needs to be added. This timestamp information of a stream is stored in a companion file that is called the timestamp index. In the case of variable-sized tuples, a b-tree index is used over the timestamp.

Queries are submitted using SQL dialect. Consider sensors deployed in a sensor data logging application, where nodes store collected information for some period of time. Since the amount of data may increase significantly over time, it needs to be logged into the flash. Figure 2 illustrates the logging (a) and querying (b) operations.



(a) Data acquisition and Logging (b) Querying and Retrieving data

**Fig. 2.** Logger Application Example

The next two commands set the timestamps of sensor nodes to a date and time and create the stream to log data. From then on, the sensors start logging temperature and light data into the stream that is resident on the flash.

```
Set timestamp of SensorNodes to ('10-SEP-2011-14:10:10','DD-MON-
YY-HH24:MI:SS');

Create stream environmentData in SensorNodes as
Select temperature, light From me
Sample every 1 minute
Window 1 year
Storage flash;
```

The following commands exemplify retrieving all the data for a specific month, and retrieving per-day temperature statistics (minimum, maximum and average).

```
Select nodeID, temperature, light, timestamp
from environmentData
Where time between todate(('01-DEC-2011-00:00:00','DD-MON-YY-
HH24:MI:SS') and todate('01-JAN-2012-00:00:00','DD-MON-YY-
HH24:MI:SS')

Select nodeID, min(temperature), max(temperature),
avg(temperature), todate(timestamp, ''DD-MON-YY')
from environmentData
group by todate(timestamp, ''DD-MON-YY');
```

A stream can be defined with a window of time-ordered tuples, and a query that is run against the window with a predefined period (the window size). A stream without a window is equivalent to a relational table with a set of time-ordered tuples in it. For instance, the window may collect 60 seconds of sensor data. Then, every 60 seconds, it computes a per-10 seconds summary of the sensor data (average, variance and maximum), sends the computed summary to some destination, and empties the window for the next period of 60 seconds. The definition of the stream for this example is shown next:

```
Create stream pressure in SensorNodes as
Select avg(value),stdev(value),max(value)
From adc0
Sample every 1 second
Window 60 seconds
Group by 10 seconds;
```

This stream with the last 60 seconds of data fits nicely in the small memories of embedded devices. Every 60 seconds, its contents is sent to data consumers. A consumer stream is a stream that specifies this stream in the from clause, as shown next:

```
Create stream collectPressure in CollectionPC as
Select nodeID, *
From pressure;
```

The command syntax is summarized in the appendix.

## 4 Stream Relational Algebra and Algorithm

The sensor network is a distributed system with at least one SME in a sensor node and an SME with catalog and a Java console application in one PC. The catalog maintains all information on node configurations and status. Queries are submitted through the console. The query is pre-parsed into a query bytecode and nodes run the query and return the result to the caller. In the case of a stream with a window, when the window fills-up the query is ran and results forwarded to registered consumers (other streams).

The constrained SME version should occupy very small amounts of code and data memory, we will describe its query processing algorithms.

The base query-processing algorithm of Figure 3 works on a row-by-row fashion, retrieving one tuple at a time, applying selection and projection restrictions on the row and outputting the results if the row is not excluded by evaluated conditions. The select clause contains a set of expressions (e.g. stream attributes, parameters, constants, function calls such as todate(), aggregation functions applied to attributes, or simple expressions). These are pre-parsed in the console application into a bytecode that represents the select fields to be interpreted by the mote. Examples of node parameters that can be included in queries include "nodeID" or sensor identifiers. Where conditions are either "operand operator operand" expressions (binary) or "operand operator" expressions (unary). Operands are (simple) expressions, and operators are a set of possible operators (e.g. ">","<","=","!=",">=","<="). Multiple where conditions can be "anded" or "ored".

In the figure, the temporary aggregation structure A maintains additive quantities (sum s, square sum ss, maximum, minimum and number of tuples processed n) that allow aggregations to be computed after all the tuples were processed. For instance, the maximum and minimum are given directly from the current maximum and minimum in the structure, the average is a sum divided by the number of tuples, and the variance is (ss-(s*s)/n)/n.

The query processing algorithm shown in the figure requires only a minimal amount of memory. It needs one tuple for input stream data, about 100 B for keeping metadata for each stream, few bytes for local variables used during query processing, space for the aggregation structure A (less than 50 B), and space for the output buffer O that holds result tuples. This buffer is flushed into network messages as soon as there are enough tuples to fill a packet payload, to be sent to the destination computer. This way, O needs only a packet payload size (about 100 B in telosB). We show results on the memory space that was consumed in the experimental section.

```
O= temporary tuple space for output tuples;
A=Aggregation structure, a temporary structure for computing aggrega-
tions;
1. Scan stream, tuple-by-tuple:
For each tuple,
     Apply selection operations (early-select) (where clause conditions)

     If selection operations evaluate to false (tuple will not contri-
     bute to output),
        go to step 2 with next tuple

     For each select clause field,
        If field is a constant, output it to a temporary output tuple
        space O;
        If the field is attribute, copy its value in current tuple to O;
        If the field is a function applied to an attribute, call the
        function with the attribute value of current tuple, output the
        result to O;
        If the field is an aggregation (e.g. sum, count, avg, max, min),
        the attribute value of the current tuple updates A, a temporary
        aggregation computation structure for that attribute (an aggre-
        gation hashmap);
        If (O already fills a network packet), fill the packet and send
        the results, emptying O)
2. End of query:
If the query is an aggregation, compose final output from aggregation
structure.
```

**Fig. 3.** Base Query Processing Algorithm

## 5 Constrained Group by and Join

The objective concerning constrained group by and join algorithms is to devise efficient solutions that may be run entirely in very small amounts of data memory, and the code should fit into the code memory of motes.

Sensor data is stored in stream format in monotonically increasing timestamp order, and it is very frequent to aggregate by time units. Therefore, we take advantage of the timestamp-order to define a simple Group-By Time approach with minimal memory requirements that is based on ordered aggregation. Only for the more generic case when the group-by attributes are not ordered we apply an external sort prior to using the same ordered aggregation algorithm. In the case of Join, a sort-merge join is implemented, with both relations participating in the join being sorted, followed by a merge-join. In this section we describe the algorithms, which are evaluated in the experimental section.

### 5.1 Time-Interval File Seek Index

Consider a query retrieving logged data for a time interval. Since the stream data is time-ordered and the stream has a timestamp index (section 3), the appropriate offset in the stream is calculated from the index and the query timestamp interval start. Tuples are then read until the timestamp interval end is reached. In the experiments we include a comparison of this with full table scan (fts) in the context of processing queries over datasets in TelosB motes.

### 5.2 GroupBy Time (GBTime) and GB-ALL

The objective of this algorithm together with the time-interval file seek is to run fast, save battery and to use the minimum possible amount of data memory, so that the algorithm can run efficiently in 2KB or less of data memory. The algorithm keeps a single aggregation computation structure (A) in data memory and is useful for grouping into time-intervals and to aggregate all the dataset. It is also used as the second step of the all-purpose group-by algorithm given in the next sub-section.

Consider an acquisition stream, that is, a stream that results from acquiring sensor data periodically, such as the example of section 3. The time granularity of the stream is given by the sensor-sampling rate (or the rate at which it receives data from another stream), and time-aggregated queries aggregate into some other time granularity. The Group-By Time algorithm of Figure 4 executes when a time argument is used in the group by clause. If the where clause contains a time interval condition (alone or “anded” with other conditions), the file seek index (section 5.1) is used to avoid full table scans. Then the algorithm simply scans the dataset tuples within the time interval specified in the where condition, while updating the aggregation computation for the current group. When the group changes (group time boundary is passed), the group aggregation is computed and it switches to compute the next group. The algorithm is described next.

```
  timeF: the time format string used in the query groups
                             ('DD-MON-YY,HH' in the above example);
  timeG: the current group identified as a string;
  aggregationStructure: A(timeG, s=0,ss=0,max=-1,min=MAXVAL,n=0);
                       (sum s, square sum ss, maximum max, minimum min,
                       and number of tuples processed n)

GBTime Algorithm:
0. timeG="";
1. If a time interval specified in the where clause restricts the inter-
val that must be considered,
       Seek the position on flash corresponding to the start of the
       time interval specified in the where clause.
2. Scan the tuples one-by-one while the tuple timestamp is lower than
the upper bound on time interval or the end of the dataset is met. Eva-
luate where conditions on the tuple, if tuple is excluded continue (2.)
with next tuple;
       if todate(timestamp, timeF) for the tuple equals timeG
              update aggregation structure variables by adding the val-
              ue(s) from the tuple;
```

```
      else // ended computing group aggregation for timeG
                compute select aggregations and expressions from A and
                output to O;
                if O fills packet, send packet and empty O;
                reset A structure for next group;
                timeG= new timeF;
  3. Send O;
  4. End.
```

**Fig. 4.** Group-By-Time Algorithm (GBTime)

GBTime is evaluated in the experimental section and its performance and energy consumption is compared with alternatives.

**Resource use**:

I/O (flash): n = number of tuples in dataset, nI in time interval

Constant-sized = nI (Variable-sized = log n + nI)

Minimum data memory:

sizeOf(A) + sizeOf(Tuple) + sizeOf(O), where O is the temporary output buffer that can be flushed whenever needed.

### 5.3 All-Purpose GroupBy (GB) and Join

An all-purpose Group-By is given for processing aggregations over generic non-stream-ordered attributes. A Sort-Group By algorithm is given. Similarly, an all-purpose Join is given with the Sort-Merge-Join algorithm. These will be slower when temporary flash-space is needed, but will handle generic aggregation and join operations. Figure 5 shows the Sort-GroupBy algorithm that was implemented. In Step 1 (external sort), the data set is sorted by the group-by attributes using an external sort (flash memory). Step 2 (re-)uses the GBTime algorithm of section 5.2, replacing time attribute values by the group-by attribute values in the algorithm. This way, the grouping of the sorted data can be done with small amounts of data memory, and re-utilizes the aggregation algorithm of GBTime (small code image).

We denote as GB-fts the GB algorithm doing a full-table scan and GB-idx a version using the timestamp index when one exists and a where clause restricts retrieval over a time interval.

```
GB-fts and GB-idx Algorithms:
Step 1. External Sort (simplified for the sake of brevity):
S= Sort buffer, should fit in memory, S=empty initially
For all tuples of dataset
  Apply where conditions, if tuple excluded by where conditions, con-
  tinue (1.) with next tuple;
  Project attributes and add remaining tuple values to S;
  If S full, apply in-mem sort algorithm, store as runfile and empty S;
For each input tuple from all runfiles
  Output next tuple in sort order and read next tuple from the runfile
  of the chosen tuple;
  Output tuples are flushed to flash when output buffer fills up, emp-
  tying the buffer;
```

**Fig. 5.** Sort-Group By Algorithm for SME

**Resource use:**

IO (flash): n = number of tuples in dataset, σ is where selectivity

n + 3σ(n) (read dataset, write runfiles, read runfiles, write sorted).

Minimum data memory: sizeOf(S)

For step 2, Sorted Group-By, see section 5.2, replacing time by group-by attribute values.

In the case of the Join algorithm shown in Figure 6, for the sort-merge join both datasets need to be sorted. The same external sort algorithm is used, then the algorithm reads sequentially tuples from both datasets simultaneously, outputting matches.

```
Sort-Merge-Join of datasets A and B
Run External Sort on A and on B to order by join attribute(s), resulting
in sortedA and sortedB (external sort Algorithm given above)
While there are input tuples from sortedA or sortedB
   If join attribute values for sortedA and sortedB match,
      Compose output tuple to O, from the sortedA and sortedB tuples;
      If O fills a packet, compose the packet and send it, then empty O.
      Retrieve next sortedA and sortedB tuples;
   Else
      Retrieve next tuple from either sortedA or sorted, by getting the
      smallest of the two based on the sort order. Replace the corres-
      ponding input tuple;
```

**Fig. 6.** Sort-Merge Join for SME

**Resource use**:

IO (flash): n = number of tuples in dataset, σ is where selectivity

IO(sort A) + IO(sort B) + σAnA+ σBnB

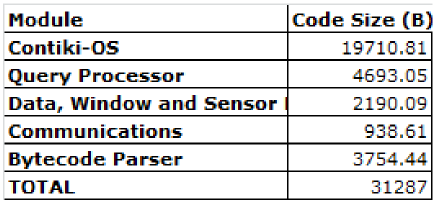
Minimum data memory: sizeOf(S) for sorts, then sizeOf(A Tuple) + sizeOf(B Tuple) + sizeOf(O).

## 6 Experiments

We have setup a set of experiments to test the approach and algorithms. Those were based on developing and running the SME for a TelosB mote with flash-stored streams, queried from a PC. Each TelosB has 48KB of ROM, 10 KB of data memory, 1024 MB of flash and 2 AA 800 mAh batteries. The PC is 2.53 GHz Intel Core 2 Duo, 4 GB 1067 MHz DD33 memory, running Windows 7, a Java Virtual Machine and a Java version of SME. The commands and data go through a TelosB sink node connecting to the gateway PC.

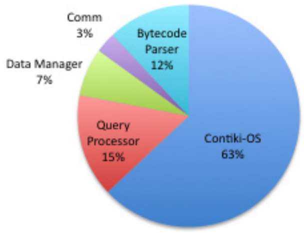
### 6.1 Code size and Data Memory

Figure 7 shows the size of the code composing the SME developed for TelosB. As shown in the figure, the total code of the SME occupies about 11KB, which together with about 20KB of the Contiki code size results in 31 KB for the total code size. This fits well into the 48KB of the code memory of this particular type of mote. Figure 7(b) shows that 63% of the code is occupied by the OS and that the second largest amount is the query processor (15%), followed closely by the query parser (12%).



| Module | Code Size (B) |
|---|---|
| Contiki-OS | 19710.81 |
| Query Processor | 4693.05 |
| Data, Window and Sensor I | 2190.09 |
| Communications | 938.61 |
| Bytecode Parser | 3754.44 |
| TOTAL | 31287 |

(a) Code Size of Modules (Bytes)



(b) code Size (%)

**Fig. 7.** Code Size of Modules

Figure 8 shows the minimum and maximum amount of memory occupied by the SME while it was running a query aggregating flash data using the algorithm GBTime. The temporary buffer (O) was set to 500 Bytes. We can see that the memory occupied was between 1.1 KB and 3 KB, fitting in the available data memory.

## 6.2 Time to Aggregate in Different Contexts

Figure 9 shows the time taken to aggregate in SME TelosB in memory (SME-ContikiTelosB) versus the time to aggregate when the dataset is on the flash (SME-TelosBflash) versus the time taken to aggregate the same amount in the computer Java version of the SME. Window sizes of 10, 50 and 100 were tried. The results show that memory operation was 5 to 6 times faster than flash on the TelosB and that aggregation on the PC was much faster.

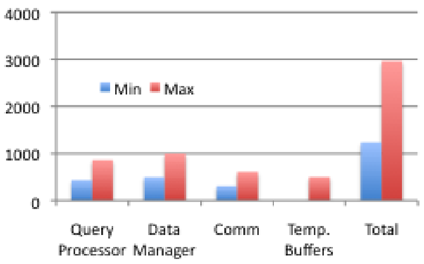


**Fig. 8.** Memory Occupancy SME (B)

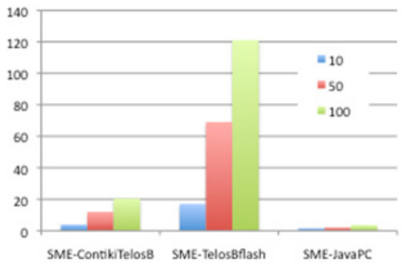


**Fig. 9.** Op. times (sec) vs windows size

## 6.3 GroupBy (GBTime, GB) over Time-Order

This experiment evaluates both efficiency and energy consumption of GBTime (section 5.2), GB-fts and GB-idx (section 5.3), and it also compares those with retrieving the data and sending it directly to be processed in the PC (Read&Send). We have loaded a dataset representing 6 months of per-minute sensor data. The "full table scan" alternatives (GB-fts and Read&Send(All)) ran over one-year dataset. Battery lifetime is measured in number of times the query can be submitted repeatedly before the battery is depleted. To get it, energy consumption was measured for the query execution using Energest on Contiki, then the number of queries for battery depletion was calculated considering two typical 1.5V 800mA AA batteries and a cutoff voltage of 1.8 V.

Figure 11 shows the execution times of aggregation plus sending the results to the PC. The per-minute dataset was aggregated per-hour over a time interval of three months.

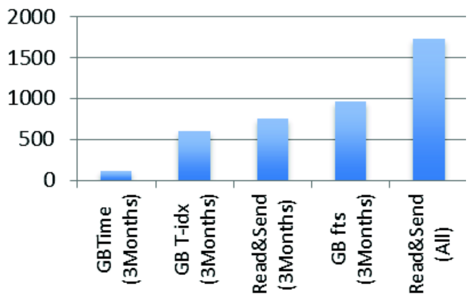


**Fig. 10.** Execution Times for Experiment 1 (secs)

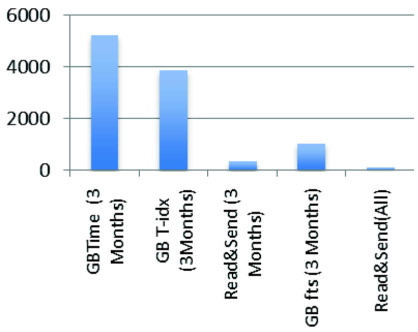


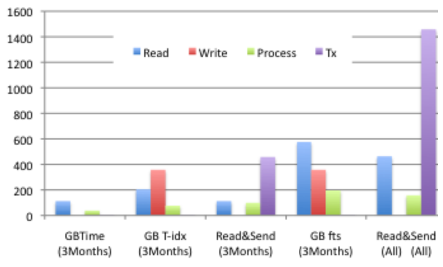
**Fig. 11.** Lifetime for Experiment 1

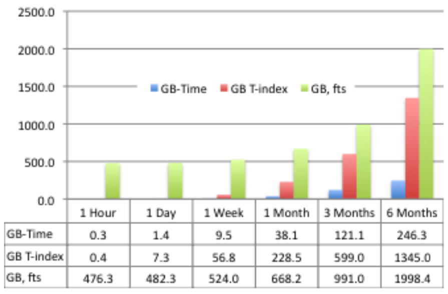
The GBTime algorithm took 121 seconds (about 2 minutes) to compute the three months results and send them to the collecting PC. This compares very favorably with GB-idx. It is also shown that if the data is simply retrieved to the PC without aggregation, it takes more than 12.5 minutes (748 secs), and full year data takes almost half an hour to collect. The time taken by GB-fts is worse than the time taken to Read&Send over 3 months of data because GB-fts is scanning the whole dataset (1 year), so GB-fts should be compared with Read&Send (All) that is also doing an fts.

As conclusion, GBTime is very efficient, and it is important to have a timestamp index to handle queries over time intervals faster.

Figure 12 shows the expected lifetime measured in number of runs of the query before batteries are depleted. The results prove that there is a great advantage in terms of lifetime in executing queries locally to summarize data before sending to the PC. The Read&Send queries resulted in orders of magnitude lower lifetimes than the aggregating queries. This is because data transmission consumes much more energy than computing or accessing flash memory.

Figure 13 is a breakdown of the time spent in each type of operation for each query type. Read&Send spends most time transmitting a large number of tuples to the collecting PC, and group-by algorithms (GB) other than GBTime spend significant time reading and writing to the flash, due to sort operations and, in the case of GB fts due to the full scan of the stream on flash. Figure 14 shows the time taken to execute an aggregation query using each alternative (GBTime, GB-idx, GB-fts) when the time interval (in the where clause) increases from 1 hour to 6 months. Once more the results show the advantage of GBTime and also show the relevance of the timestamp index (GB-idx vs GB-fts) in (time-)selective queries.





| | 1 Hour | 1 Day | 1 Week | 1 Month | 3 Months | 6 Months |
|---|---|---|---|---|---|---|
| GB-Time | 0.3 | 1.4 | 9.5 | 38.1 | 121.1 | 246.3 |
| GB T-index | 0.4 | 7.3 | 56.8 | 228.5 | 599.0 | 1345.0 |
| GB, fts | 476.3 | 482.3 | 524.0 | 668.2 | 991.0 | 1998.4 |

**Fig. 12.** Breakdown of Time in Operations (secs) **Fig. 13.** Varying Where Conditions

### 6.4 All-Purpose Group-By and Join

For evaluation of performance and battery lifetime of GB and Join algorithms (section 5.3), we ran experiments grouping and joining non-ordered datasets A and B with 5K, 10K, 25K and 50K rows. The group-by attribute in the datasets was chosen to generate a number of distinct groups that is 10% of the number of tuples. Figure 15 shows that Read&Send was faster than GB&Send for this test. This is because GB&Send has more overhead with sorting the dataset, computing aggregations and sending the results to the PC than Read&Send retrieving all tuples and sending them to the PC. Figure 16 shows lifetime for the same experiment. It is interesting to see that although GB&Send was slower than Read&Send, in terms of lifetime it lasts more than twice. This is because Read&Send transmits 90% more tuples, and transmission consumes much more power than computation or accessing the flash memory.

In what concerns the comparison with Join, it is interesting to compare the height of the upper limit of the dashed line on top of GB&Send with the execution time for Join. Join is processing two datasets with the same size (e.g. 5K with 5K), while the other alternatives are processing only one such dataset (e.g. 5K). The dashed lines are twice the execution time, which corresponds to processing (5K plus 5K). The execution time of GB&Send on the two datasets is very similar to the execution time of Join. Both cases sort both datasets, and both have to retrieve then every row of both datasets and do some processing with them. Finally, Join has to send the joined tuples, and GB has to send the aggregated tuples.

## 7 Conclusion

In this paper we proposed algorithms for a stream management engine to process data efficiently in constrained mote devices, including a very efficient time-ordered group-by processing solution, and all-purpose group by and join algorithms. We also proposed the Stream Management Engine approach to deal with the data. We have argued that the approach is very useful to allow applications to log data and to query the data efficiently, and that it is prepared to process streams in-memory and in flash and to send data between nodes. We evaluated the approach experimentally and against alternatives, showing that the devised algorithms and Stream Management approach fits the memory of constrained devices and processes efficiently, while saving energy.

We have proposed and tested a base set of algorithms. In the future, we expect to test other forms of indexing and processing algorithms.

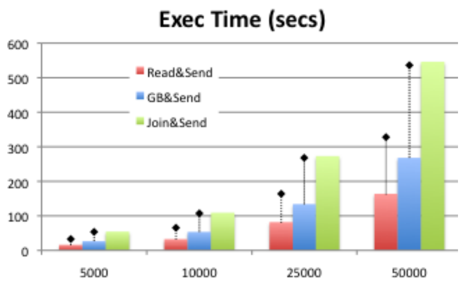


**Fig. 14.** Performance – GroupBy and Join

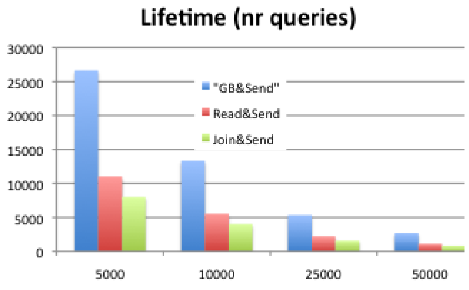


**Fig. 15.** Lifetime – GroupBy and Join

## References

1. Diao, Y., Ganesan, D., Mathur, G., Shenoy, P.J.: Rethinking Data Management for Storage-centric Sensor Networks. In: CIDR, Asilomar, USA, pp. 22–31 (January 2007)
2. Aberer, K., Hauswirth, M., Salehi, A.: Infrastructure for data processing in large-scale interconnected sensor networks. In: Mobile Data Management, Germany (2007)
3. Agrawal, D., Ganesan, D., et al.: Lazy- adaptive tree: An optimized index structure for flash devices. In: Proceedings of IC Very Large Data Bases (VLDB), Lyon, France (August 2009)
4. Bakshi, A., et al.: The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems. In: Proc. EESR (2005)
5. Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. In: Proceedings of the Second International Conference on Mobile Data Management (2001)
6. Boulis, A., et al.: Design and implementation of a framework for efficient and programmable sensor networks. In: Proc. MobiSys (2003)
7. Franklin, M., Jeffery, S., Edakkunni, A., Hong, W., et al.: Design Considerations for High Fan-in Systems: The HiFi Approach. In: CIDR (2005)
8. Gehrke, J., Madden, S.: Query Processing in Sensor Networks. IEEE Pervasive Computing 3(1), 46–55 (2004)
9. Gibbons, P.B., Karp, B., Ke, Y., Nath, S., Seshan, S.: IrisNet: An Architecture for a World- Wide Sensor Web. IEEE Pervasive Computing 2(4) (2003)
10. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using *kairos*. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 126–140. Springer, Heidelberg (2005)
11. Li, S., et al.: Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In: Proc. Int. Workshop on Information Processing in Sensor Networks (2003)
12. Madden, S., Franklin, M., et al.: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. on Database Systems 30(1), 122–173 (2005)

13. Nath, S., Kansal, A.: FlashDB: Dynamic self-tuning database for NAND flash. In: International Conf. on Information Processing in Sensor Networks Cambridge, USA (April 2007)
14. Shen, C.C., et al.: Sensor Information Networking Architecture and Applications. E Personal Communications Magazine 8(4), 52–59 (2001)
15. Shneidman, J., Pietzuch, P., et al.: Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard University, EECS (2004)
16. Srisathapornphat, C., et al.: Sensor Information Networking Architecture. In: Proc. Int. Workshops on Parallel Processing (2000)
17. Rosenblum, Ousterhout, J.: The design and implementation of a log structured file system. In: ACM Sympo. on Operating Systems Principles, Pacific Grove, USA (1991)
18. Woo, A., Madden, S., Govindan, R.: Networking support for query processing in sensor networks. Commun. ACM 47(6), 47–52 (2004)
19. Zeinalipour-Yazti, Lin, S., et al.: MicroHash: An efficient index structure for flash-based sensor devices. In: USENIX FAST 2005, San Francisco, CA, USA (2005)
20. Whitehouse, K., Zhao, F., Liu, J.: Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. Wireless Sensor Networks, 5–20 (2006)
21. Yoneki, E., Bacon, J.: A survey of Wireless Sensor Network technologies: research trends and middleware's role. Tech. R. of Univ of Cambridge, UCAM-CL-TR-646 (2005)
22. Wang, M.M., Cao, J.N., Li, J., et al.: Middleware for wireless sensor networks: A survey. Journal of Computer Science and Technology 23(3), 305–326 (2008)
23. Mottola, L.: Programming Wireless Sensor Networks: From Physical to Logical Neighborhoods. PhD Thesis, Politecnico di Milano, Italy (2008)
24. Schreiber, F.A., et al.: PERLA: a Data Language for Pervasive Systems. In: Sixth International Conf. on Pervasive Computing and Communications, Hong Kong, pp. 282–287 (2008)
25. Polastre, J., Szewczyk, R., Culler, D.E.: Telos: enabling ultra-low power wireless research. In: IPSN 2005. IEEE, Los Angeles (2005)
26. Levis, P., Madden, S., et al.: The Emergence of Networking Abstractions and Techniques in TinyOS. In: NSDI 2004, pp. 1–14. USENIX (2004)
27. `http://www.arduino.cc/`
28. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: LCN 2004 (2004) ISBN 0-7695-2260-2
29. Pachube [Pachube], `https://cosm.com/`
30. SensorCloud [SC], `http://www.sensorcloud.com/`

## A Appendix – SME Query Expressions [nodeID| nodeSet] =NODES

```
[Create Stream Xpto [in NODES as]
Select [select expressions][in NODES ]
From [ sensorID | streamName | me ] [Where clause]
[Group by clause][sample clause] [window clause] [storageclause]

Update stream sensorID | stream [in NODES ]
set [set expressions] [Where clause];

Insert into stream sensorID | stream [in NODES] values [values];

Del stream sensorID|stream [in [nodeID|nodeSet]][Where clause];
```