Discovery of Frequent Distributed Event Patterns in Sensor Networks*

Kay Römer

Institute for Pervasive Computing, ETH Zurich, Switzerland
roemer@inf.ethz.ch

Abstract. Today it is possible to deploy sensor networks in the real world and collect large amounts of raw sensory data. However, it remains a major challenge to *make sense of sensor data*, i.e., to extract high-level knowledge from the raw data. In this paper we present a novel in-network knowledge discovery technique, where high-level information is inferred from raw sensor data directly on the sensor nodes. In particular, our approach supports the discovery of frequent distributed event patterns, which characterize the spatial and temporal correlations between events observed by sensor nodes in a confined network neighborhood. One of the key challenges in realizing such a system are the constrained resources of sensor nodes. To this end, our solution offers a declarative query language that allows to trade off detail and scope of the sought patterns for resource consumption. We implement our proposal on real hardware and evaluate the trade-off between scope of the query and resource consumption.

1 Introduction

Systems research in sensor networks has reached a point where we can build and deploy medium-sized sensor networks and collect large amounts of raw or preprocessed sensor data during months of unattended operation. However, it remains a major challenge to make sense of the collected data, i.e., to extract the relevant knowledge from the raw data. Most existing techniques for knowledge discovery from sensor data are centralized and require the extraction of raw sensor data from the network. However, this can be very costly due to the large data volume and does not scale to large networks.

In a previous position paper [20] we sketched an in-network knowledge extraction technique that supports the discovery of frequent distributed event patterns. In this paper, we turn this idea into a complete system, implement it on sensor nodes and study important performance metrics. The key advantage of our in-network approach is that the extracted knowledge is directly available to the sensor nodes and can be used to control the behavior of the sensor nodes (e.g., to prioritize processing of event patterns that occur infrequently). Also, the extracted knowledge is often much more compact than raw sensor data and can therefore be more efficiently extracted from the sensor network than raw sensor data.

Our approach is based on events, that is, each sensor node locally analyzes the output of its sensors to find relevant real world occurrences. In many applications it is

* The work presented in this p aper was partially supported by the Swiss National Science Foundation under grant number 5005-67322 (NCCR-MICS).

important to put such events into a spatial and temporal context, i.e., to consider the correlation of an event observed by a sensor node with events observed by surrounding sensor nodes in the recent past. In an equipment monitoring application (e.g., [1]), for example, one is interested in understanding if abnormal vibration signatures are correlated with nearby abnormal temperature readings. In a bird monitoring application (e.g., [21]), one is interested in understanding if certain events in the neighborhood of a nesting burrow (e.g., noise, motion) are correlated with birds leaving their nests.

Our approach supports this type of application by providing a framework to analyze the correlation of a certain type of event on a sensor node with *context events* observed by nodes in a confined neighborhood of this node in the recent past. For example, we might find that in 30% of the cases where a bird left its nest, motion has been detected by at least one sensor located within 10 meters of the nest no more than 3 minutes in the past. We call such a correlation of events a *distributed event pattern*. Our technique discovers such distributed event patterns that occur with a frequency not less than a user-specified minimum. Besides a minimum frequency, a user has to specify local events of interest and certain temporal and spatial constraints using a declarative query language.

The discovered set of frequent event patterns can be considered as a compact characterization of the "common behavior" observed by a set of sensor nodes over long periods of time. Likewise, an event pattern that is not frequent can be considered as an exceptional occurrence. Frequent event patterns can be used in two primary ways: Firstly, by a user to learn about the common behavior, or to be notified of exceptional behavior or of significant changes to the common behavior. Secondly, as the event patterns are computed on the sensor nodes, the latter can use this information to control or adapt their behavior, for example, to allocate more resources for the processing and communication of rare event patterns than for more common ones.

In this paper we focus on how frequent event patterns can be efficiently computed on resource-constrained sensor nodes. Although sensor nodes are becoming more powerful over time, constrained node resources are the primary challenge in designing and implementing our in-network knowledge extraction technique. Our approach to deal with this challenge lies in the query language, which allows the user to define the detail (e.g., granularity of temporal and spatial relationships between local events) and scope (e.g., minimum pattern frequency, involved local events, maximal temporal or spatial distance between local events) of sought patterns: the more detailed or the larger the scope of sought patterns, the more expensive is pattern discovery. Thus, we offer the user a turning knob to trade off detail and scope for resource consumption.

Note that the above *discovery of frequent events patterns* is different from *detection of event patterns*. For the latter, the user needs to specify in advance and exactly which event patterns the system should detect. With our approach, the system itself identifies event patterns that occur frequently given certain constraints on the sought event patterns. As such, our approach can be considered as a relaxation of detection of event patterns.

We begin with an overview of the system in Sect. 2 and introduce patterns and queries in Sect. 3, before presenting the core algorithms in Sect. 4. Important implementation aspects are discussed in Sect. 5. We evaluate our proposal in Sect. 6.

2 System Overview

The overall architecture of the proposed system is as follows. A user can pose a query to the system using a declarative language. Such a query defines the local events of interest and additional constraints on the sought frequent distributed event patterns, see Sect. 3 for details. A query is compiled into executable code (containing both the query parameters and the pattern discovery algorithm) at the gateway of the sensor network and the resulting executable is distributed to each node in the sensor network using a code distribution protocol.

Using the query parameters, the pattern discovery algorithm executing at a sensor node continuously collects event notifications from nodes in a confined network neighborhood and computes the set of frequent distributed events patterns as detailed in Sect. 4. This set can now be used in a number of different ways as discussed in Sect. 1.

Depending on the application scenario, the pattern discovery algorithm may be executing at some sensor nodes (e.g., only on nest nodes in the bird monitoring example in Sect. 1) or on every node of the sensor network.

3 Patterns and Queries

We will illustrate the notion of distributed event patterns using the bird monitoring example given in Sect. 1. Here, sensor nodes are deployed in and around the nest and can detect two types of events: motion (of creatures) and a bird leaving its nest. We are interested in understanding how *leave* events are correlated with *motion* events in the vicinity of the nest. Here, our system might find a frequent distributed event pattern such as

```
(motion, <10m, <3min, >=1) : leave [30%]
```

This pattern has to be read as follows: In 30% of the cases where a bird left its nest, motion has been detected by at least 1 sensor located within 10 meters of the nest sensor no more than 2 minutes in the past.

In general, a pattern consists of a *local event* (right of the pattern's column) and a term that summarizes occurrences of *context events* (left of the pattern's column) in a spatial and temporal neighborhood of the above event. The frequency or *support* of a pattern equals the number of occurrences of the local event for which the left-hand term of the pattern also applies, divided by the number of occurrences of the local event (regardless if the left-hand side of the pattern applies or not). A frequent pattern is a pattern whose support is greater than or equal to a given *minimum support*.

In order to discover such patterns, a user has to specify a query. The query defines events and context events of interest and a number of constraints on the sought patterns. These constraints are needed to cut down the otherwise huge search space for possible patterns to allow an implementation of the pattern discovery algorithm on resource-constrained sensor nodes. Fig. 1 shows a possible query for our bird monitoring example.

In our system, time is divided into epochs of fixed length. Nodes are synchronized such that epochs begin and end at approximately the same real-time instants at all nodes across the network. Since typical epoch durations are in the order of seconds or tens of

seconds, required synchronization is rather loose and easily achieved with existing synchronization protocols. In our example query, epoch length is 60 seconds as specified in line 2. Epochs are identified by monotonically increasing integer numbers starting with zero.

```
1  // epoch length
2  epoch = 60 // [seconds]
3  // event definitions
4  event motion { max:accel[0] > threshold }
5  event leave { max:pir[0] == 0 &&
6                max:pir[1] == 1}
7  // events and context events
8  levents {leave}
9  cevents {motion}
10 // temporal and spatial scope
11 neighborhood = 1 // [hop]
12 history = 6 //epochs
13 // minimum support and error bound
14 minsupport = 30 // [%]
15 error = 5 // [%]
16 // distance partitions [meters]
17 distance { near =(0,10], far =(10,20] }
18 // time interval partitions [epochs]
19 time { now=0, recent =[1,3], old =[4,6] }
20 // frequency partitions [number]
21 frequency { none=0, some=[1,infty ] }
```

Fig. 1. An example query.

In each epoch, a sensor node can generate at most one instance of each possible event type. In our sample query, two event types `motion` and `leave` are defined in lines 4 to 6, respectively. A *motion* event (i.e., ground is vibrating) is generated in an epoch if the maximum output value of the accelerometer sensor in that epoch is greater than a threshold (i.e., `max:accel[0] > threshold` is true, where the "0" in square brackets refers to the current epoch). A `leave` event is generated in the current epoch if the passive infrared sensor (PIR) detected presence of a bird in the previous epoch (i.e., `max:pir[1] == 1` is true, where the "1" in square brackets refers to the previous epoch), but not in the current epoch (i.e., `max:pir[0] == 0` is true). We assume that PIR is a binary sensor that outputs either zero or one. The set of all events given in a query will be denoted by E.

The builtin event definition language only supports simple predicates over aggregated sensor values in the current and past epochs. In every epoch, sensor values are aggregated in predefined ways (e.g., minimum, maximum, average). For more complex and realistic events, the query language supports external events whose detection is implemented outside of our system (e.g., using more elaborate sensor signal processing techniques).

Lines 8 and 9 in our sample query define `leave` as a local event and `motion` as a context event, respectively. Note that an arbitrary numbers of local events and context events can be specified and event types may be declared as both local events and context events. The set of all context and local events defined in a query will be denoted by E_c and E_l, respectively.

Lines 11 and 12 define the spatial and temporal scope that should be considered for the correlation analysis. The spatial scope, denoted by $SSCOPE$, is given as a maximum hop count, such that only correlations between events generated by nodes at most $SSCOPE$ hops apart are considered. The temporal scope, denoted by $TSCOPE$ is given as a number of epochs, such that only correlations between events that occurred within a time window of $TSCOPE$ epochs are considered. In the example, if the node in the nest executing the pattern discovery algorithm observes a local event during a given epoch, only context events generated no more than 6 epochs in the past by nodes

no more than one hop away (including the nest node itself) will be considered for the correlation analysis.

Recall that our system will only discover patterns that occurred with a given minimum support, which is given in line 14 of the sample query. Further, we allow a certain error (given in line 15 of the sample query), such that a pattern may be reported as being frequent by our system if its true support is greater than or equal to minimum support minus error bound. We will denote minimum support and error bound as MS and MS_e, respectively.

Finally, the query contains a quantization of Euclidean distances between nodes, time intervals (between event occurrences, where time is measured in epochs), and frequency of event occurrences into a set of discrete partitions. Each partition is an interval that is either open (parenthesis) or closed (bracket). Note that the set of distance (time, frequency) partitions does not need to cover the whole domain of distances (time, frequency). By this, a user can constrain the search space for patterns to certain distances and time intervals between events as well as to certain frequencies of events. We assume the existence of an implicit, possibly non-continuous or empty partition $\perp$ that covers the part of the domain that is not covered by partitions that have been explicitly defined. The sets of all distance, time, and frequency intervals defined in a query will be denoted by DP, TP, and FP, respectively. We assume the existence of mapping functions map_d, map_t, and map_f, which map a given distance, time interval, and frequency to elements of $DP \cup \{\perp\}$, $TP \cup \{\perp\}$, and $FP \cup \{\perp\}$, respectively.

We can now specify the general form of a pattern in terms of the query parameters as follows:

$$\bigwedge_{i=1..N} (e_i^c, dp_i, tp_i, fp_i) \quad : \bigwedge_{j=1..M} e_j^l \quad [s] \tag{1}$$

Here, $e_i^c \in E_c$ is a context event, $dp_i \in DP$ is a distance partition, $tp_i \in TP$ is a time partition, and $fp_i \in FP$ is a frequency partition. $e_j^l \in E_l$ is a local event and s is the support of the pattern. Note that the above pattern is equivalent to M patterns with only one local event and identical terms on the left-hand sides, but with possibly different support values. The pattern is frequent if $s \geq MS$. One example pattern would be:

```
(motion, near, recent, some) AND
(motion, near, now, some) : leave [30%]
```

Note that while the above discussion is based on the notion of events (defined as a state change), patterns can also be used to reason about correlations between different *states* as well as between states and events. In our bird monitoring example, we could define an event *present* as follows: `event present { max:pir[0] == 1 }`. This event would fire in every epoch as long as a bird is in the nest, thus implementing the *state* "a bird is in the nest".

4 Discovery of Frequent Patterns

The pattern discovery algorithm executing at a sensor node consists of several components which will be discussed in this section. Firstly, a sensor node collects event

occurrences from a confined network neighborhood and transforms this information into a pattern for each epoch (Sect. 4.1). These patterns are represented as sets of small integers, so-called itemsets (Sect. 4.2). From the resulting stream of itemsets, frequent itemsets are discovered (Sects. 4.3 and 4.4).

4.1 Data Collection and Pattern Generation

The pattern discovery algorithm is executing at one or more sensor nodes (as specified by the user using mechanisms outside of the scope of this paper) to discover frequent event patterns. We will denote such sensor nodes as *discovery nodes*. Sensor nodes that are within the spatial scope $SSCOPE$ of a discovery node are called *client nodes*. Note that a single sensor node may both act as a discovery node and as a client node to one or more other discovery nodes. Also note that the set of client nodes may change over time due to fluctuation of wireless links and due to nodes dying or being added. Throughout this section we consider a single discovery node.

The pattern discovery algorithm executing on the discovery node proceeds as follows. After each epoch t, the algorithm checks if any local events occurred locally during t. If so, a pattern is constructed for epoch t. Otherwise, nothing needs to be done.

To construct the pattern, a request message is sent to all client nodes containing the identity and location of the discovery node and epoch t. Client nodes reply a message containing the event occurrences during $TSCOPE$. Essentially, a reply message from node i contains values $freq_i(e, dp, dt)$ for each context event e, distance partition dp, and time partition dt that have been defined in the query. This value equals 1 iff event e occurred on node i in the distance partition dp during time partition dt with respect to the requesting discovery node and is zero otherwise. The discovery node computes the sums $freq(e, dp, dt) = \sum_i freq_i(e, dp, dt)$ over all client nodes to obtain the following pattern for epoch t:

$$\bigwedge_{\forall e \in E_c, dp \in DP, tp \in TP} (e, dp, tp, map_f(freq(e, dp, tp))) \ : \ \bigwedge E_l(t) \qquad (2)$$

where $E_l(t)$ refers to the set of local events that occurred at the discovery node during epoch t.

If $SSCOPE = 1$ (i.e., a spatial scope of one hop), then the request is implemented by a broadcast message from the discovery node to all child nodes and the replies are implemented by unicast messages from the child nodes to the discovery node. If $SSCOPE > 1$, then networking abstractions such as Abstract Regions [23] may be used which support the above communication pattern also for multi-hop neighborhoods. Also, in-network aggregation [18] may be used to compute the sums $freq(e, dp, tp)$ in the network rather than at the discovery node.

4.2 Pattern Representation

Patterns are represented by so-called *itemsets*, i.e., a set of items. Conversion of patterns to itemsets and vice versa is accomplished as follows. Each term on the left-hand side of a pattern is mapped to an item by concatenating the event identifier, distance

partition identifier, time partition identifier, and frequency partition identifier. Each local event is mapped to an item consisting of the respective event identifier. The reserve mapping is analogous. In the remainder of the paper, we will use the terms pattern and itemset synonymously. For example, the pattern `(motion, near, recent, some) AND (motion, near, now, some) : leave` maps to the itemset {motion.near.recent.some, motion.near.now.some, leave}. It is easy to see that the maximum size of an itemset is

$$|E_c| \times |DP| \times |TP| \times |FP| + |E_l| \tag{3}$$

Hence, itemsets can be implemented as sets of small integers by mapping each possible item to an integer between 1 and the above maximum size. In our system, itemsets are implemented as bitvectors.

4.3 Frequent Patterns

The procedure described in Sect. 4.1 produces a stream of patterns (one for each epoch where a local event occurs), each of which is represented as an itemset as described in Sect. 4.2. We now need to find itemsets which are frequent with respect to this stream S of itemsets.

We will constrain our search to frequent itemsets is which contain only one local event $e \in E_l$. The support of such an itemset is defined as the number of itemsets in S of which is is a subset, divided by the number of itemsets in S which contain e as a local event. An itemset is frequent if its support is greater than or equal to the minimum support MS given in the query. Note that frequent itemsets are not necessarily elements of S, but they are subsets of one or more elements of S. Also note that every subset of a frequent itemset is also frequent and its support is greater than or equal to the support of the superset.

Several algorithms have been proposed to discover frequent itemsets from a stream of itemsets (e.g., [7, 10, 15]). The difficulty of this problem lies in the fact that only one pass over S is possible as S grows without bounds over time and hence cannot be stored completely on resource-constrained devices. Much better algorithms exist if multiple passes over S are possible. Typical single-pass algorithms therefore use a so-called *synopsis data structure*, which is essentially a compressed version of the data stream. Frequent itemsets can then be computed from the synopsis data structure which can be randomly accessed. However, synopsis data structures used by the above algorithms are still too large to fit into the constrained memory of a sensor node. Also, as we are ultimately interested in frequent itemsets (and not in the synopsis), memory is needed for both the synopsis data structure *and* frequent itemsets.

We therefore developed an algorithm that directly generates frequent itemsets without using a separate synopsis data structure. The algorithm basically splits S into small blocks B of fixed size which fit into main memory. An efficient multi-pass algorithm is used to discover frequent itemsets FIB_i in each block B_i. Each itemset $is \in FIB_i$ is associated with a counter $is.c$ that holds the number of itemsets in B_i of which is is a subset. That is, the support of is in B equals $is.c \times 100\%/|B|$. The frequent itemsets in all blocks are then merged in an incremental fashion to obtain the frequent itemsets FIS of S. Initially, FIS is empty. To merge FIB_i into FIS, we merge each frequent

itemset $is \in FIB_i$ into FIS by either inserting is into FIS if $is \notin FIS$, or by adding $is.c$ to the counter value of the existing itemset in FIS. The support of an itemset $is \in FIS$ then equals $is.c \times 100\%/|S|$.

For the ease of exposition we assume that the query contains only a single local event, i.e., $|E_l| = 1$. If more than one local event has been defined, the data stream will be split into $|E_l|$ data streams each of which contains only patterns with a single type of local event. These streams will then be processed separately as described above, but the resulting frequent itemsets will all be merged into a single instance of FIS.

Although simple, it is not clear that the above approach obtains the correct result, as an itemset that is frequent in S may not be frequent in some blocks B_i, such that the support of an itemset in S is not computed correctly. To fix this problem, we will use a smaller support value $\overline{MS} < MS$ when discovering frequent itemsets in a block. We will select $\overline{MS}$ such that we meet the error bound MS_e given in the query. That is, the support $\overline{s}$ we compute for an itemset with respect to S will be not less than the true support s of that itemset minus MS_e. Due to this, all itemsets in FIS with $\overline{s} \geq MS - MS_e$ will be considered frequent. This set of itemsets includes all true frequent itemsets (i.e., for which $s \geq MS$) plus additional ones that are actually not frequent with bounded error $s - \overline{s} \leq MS_e$.

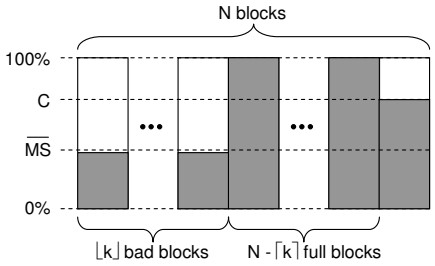

Fig. 2. Stream of itemsets that maximizes $s - \overline{s}$. Grey bars indicate the support s in a block on a scale from 0 to 100%.

How do we compute $\overline{MS}$ given MS and MS_e? For this, let us assume we have chosen some value $\overline{MS} < MS$. Let us consider an itemset is which is frequent in the stream S with support s. Our algorithm will output a value $\overline{s} \leq s$ for the frequency of is. We are interested in computing an upper bound for the error $s - \overline{s}$. For this, let us assume for now that S is split into N equal-sized blocks. We will see later that the actual length of the stream is irrelevant. Let us further consider a worst-case stream S that maximizes $s - \overline{s}$, i.e., which minimizes $\overline{s}$. Note that our algorithm only makes an error if the support of is in a block is smaller than $\overline{MS}$. In this case, is will not be considered a frequent itemset in this block and its support in this block will not be considered when computing $\overline{s}$. That is, to maximize the error, there should be as many blocks as possible where the support of is is just below $\overline{MS}$. To maximize the number of such "bad" blocks, is must have a support of 100% in the remaining blocks ("full" blocks), such that the global support of is for the whole stream is s. It follows that the (fractional) number k of bad blocks in the worst-case stream must satisfy the constraint $k \times \overline{MS} + (N - k) \times 100\% = N \times s$, which can be solved for k to obtain

$$k = N \frac{100\% - s}{100\% - \overline{MS}} \tag{4}$$

It follows that the worst-case configuration includes $\lfloor k \rfloor$ bad blocks, $N - \lceil k \rceil$ full blocks, and optionally one block with support $C = (k - \lfloor k \rfloor) \times \overline{MS} + (\lceil k \rceil - k) \times 100\%$ with $\overline{MS} < C < 100\%$ unless k is an integer. If k is an integer then that block doesn't

exist and $C = 0$. The worst-case configuration of S is illustrated in Fig. 2. Note that the order of the blocks in the stream is irrelevant.

We can now express the error $s - \overline{s}$ as follows:

$$s - \overline{s} = s - \frac{(N - \lceil k \rceil) \times 100\% + C}{N} = s - \frac{(N - k) \times 100\% + (k - \lfloor k \rfloor) \times \overline{MS}}{N}$$
$$\leq s - \frac{(N - k) \times 100\%}{N} \tag{5}$$

The "$\leq$" holds because $(k - \lfloor k \rfloor) \times \overline{MS} \geq 0$. Inserting Eq. 4 into Eq. 5 and rearranging terms we obtain

$$s - \overline{s} \leq s - (s - \overline{MS}) \frac{100\%}{100\% - MS} \tag{6}$$

Note that N has been eliminated, that is, the derived error bound is independent of the actual length of the stream. It can be easily verified that the right-hand side of Eq. 6 is monotonically decreasing in s for all $MS > 0$. That is, we obtain the largest error $s - \overline{s}$ for the smallest possible value of s, which is MS. s cannot be smaller than MS, since itemset is would then not be frequent in contradiction to our assumption. Replacing s by MS and rearranging terms we obtain

$$s - \overline{s} \leq MS - \frac{MS - \overline{MS}}{100\% - \overline{MS}} 100\% \tag{7}$$

To ensure that $s - \overline{s} \leq MS_e$, we require that the right hand side of Eq. 7 equals MS_e. Solving for $\overline{MS}$ we obtain

$$\overline{MS} = \frac{MS_e \times 100\%}{100\% + MS_e - MS} \tag{8}$$

Using Eq. 8, we can compute the minimum support $\overline{MS}$ to be used for discovering frequent itemsets in a block, such that the resulting error $s - \overline{s}$ is never greater than MS_e.

4.4 Closed Patterns

Since there may be a large number of frequent itemsets, we will consider so-called *closed itemsets* instead [22, 24]. A closed itemset is a frequent itemset which has only proper supersets with smaller support than itself. In can the shown that the set of closed itemsets of S contains the same information as the set of frequent itemsets of S. In practice, the number of closed itemsets can be orders of magnitude smaller than the number of frequent itemsets. In our algorithm, FIB and FIS will be sets of closed itemsets.

The algorithm for discovering closed itemsets of S proceeds as follows. It collects itemsets (generated as described in Sect. 4.1) into a block B until a complete block of size BN has been filled. Then, the block is compressed by removing duplicate itemsets $B[i] = B[j]$ (equality does not consider the counter values $B[\_].c$): we set $B[i].c \leftarrow B[i].c + B[j].c$ and remove $B[j]$.

We can now implement a function $isfreq(is)$ as depicted in Fig. 3 that computes the support of an itemset is for B and checks whether or not that itemset is frequent in B by iterating over the compressed block and computing the number of itemsets in B of which is is a subset (lines 7 and 8). The function terminates early (lines 9 and 10) if the number of unprocessed itemsets in B is too small to make is frequent.

```
1  bool isfreq (itemset &is) {
2    int rest ← BN;
3    int minc ← BN × MS/100%
4    is.c ← 0;
5    foreach bis ∈ B {
6      rest ← rest − bis.c;
7      if (is ⊂ bis)
8        is.c ← is.c + bis.c;
9      else if (is.c + rest < minc)
10       return false ;
11   }
12   return true ;
13 }
```

Fig. 3. Algorithm to compute the support of itemset is in the current block.

To find closed itemsets in block B that contain local event e_l, function $block()$ is used as depicted in Fig. 4 (left). After execution, FIB will hold all closed itemsets. With the help of $traverse()$ (also given in Fig. 4), $block()$ basically enumerates all possible itemsets which contain local event e_l, computes their support and stores closed itemsets in FIB. However, two properties of closed itemsets are exploited to prune the huge search space significantly. Firstly, if itemset is is not frequent, no superset of is is frequent. Secondly, if adding an item i to itemset is does not change the support of is, then is cannot be a closed itemset. Exploiting these properties for pruning is a standard technique [5]. Additional techniques exist, but these require significant amounts of memory [24] or do not have a large impact on runtime according to our experience.

In more detail, $block()$ creates the itemsets is which contain only local event e_l in line 27, computes the support of this itemset in line 28, and invokes $traverse(is, tail)$ in line 29, which recursively enumerates supersets of is by incrementally moving items from $tail$ to is. Initially, $tail$ contains all possible items except items that represent local events (line 25). The first loop in $traverse()$ implements pruning by moving all items from $tail$ to is that do not change the support of s (lines 5 - 8), exploiting the second of the properties mentioned above. Also, if adding an item from $tail$ to is makes is infrequent, then that item is removed from $tail$ (line 10). The second loop (line 12) implements the recursion step for all items remaining in $tail$. Finally, is is added to FIB if the latter doesn't contain an itemset which is a superset of is and has same support (lines 19-21). If such a superset exists in FIB, then is cannot be a closed itemset. Note that $traverse()$ implements a depth-first search of the itemset space, that is, before is is considered for insertion into FIB, all supersets of is are considered first.

Next, function $merge()$ as depicted in Fig. 4 (right) is used to merge the closed itemsets in FIB for the current block into FIS. The latter set of itemsets holds closed itemsets of the stream S seen so far.

Basically, merging is implemented by considering each possible intersection of itemsets from FIS and FIB (is in line 35). If $is \notin FIS$, then the support of is with respect to FIS equals the support of the superset $sis \supset is$ with maximum support

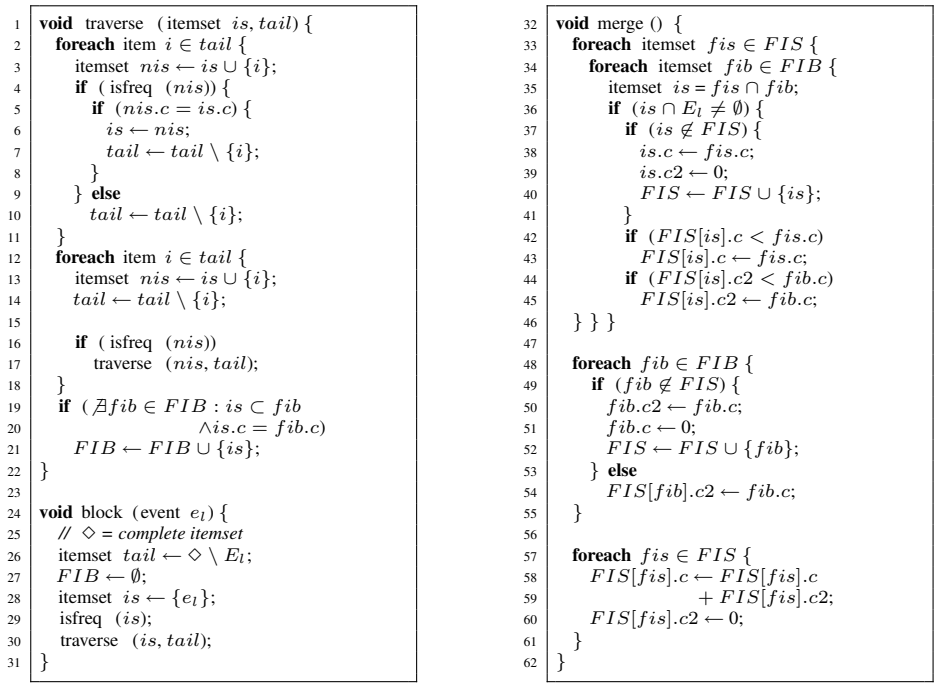

Fig. 4. Algorithm to discover closed itemsets.

among all supersets $sis \in FIS$. The analog applies for the support of is with respect to FIB. The support of each intersection is is incrementally computed in lines 37-45 and stored in the fields $is.c$ (support with respect to FIS) and $is.c2$ (support with respect to FIB). An separate loop in lines 48-55 is used to add all itemsets $fib \in FIB$ to FIS. To compute the support values $fib.c$ and $fib.c2$, we can assume that either $fib \in FIS$, or fib has support 0 in FIS. If fib has nonzero support in FIS and is not contained in FIS, then a superset of fib must be contained in FIS. However, in this case fib has already been added previously as an intersection, because the intersection of fib with a superset of itself equals fib.

Finally, the new support values of all itemsets in FIS are computed in lines 57-61 by adding the support in FIS (counter c) and the support in FIB (counter $c2$). All itemsets $is \in FIS$ that satisfy $is.c \geq (MS - MS_e) \times |S|/100\%$ afterwards are output as closed itemsets of the stream S seen so far.

4.5 Maximal Patterns

In some cases it is sufficient to know whether or not an itemset is frequent, that is, the exact support does not matter. In these cases, we can compute the set of *maximal itemsets* [11, 14] given the set of closed itemsets computed as described in the previous section. A frequent itemset is is maximal if there are no supersets of is which are frequent. Note that an itemset is frequent if and only if it is a subset of a maximal

itemset. Hence, knowing the set of maximal itemsets a user knows all frequent itemsets and he could send inquiries to the sensor network to learn the frequencies of specific frequent itemsets.

The number of maximal itemsets is often orders of magnitude smaller than the number of closed itemsets. Maximal itemsets are typically an extremely compact and human-readable summary of the "common behavior" observed by a sensor network, see Sect. 6 for an example. Note that

5 Implementation Aspects

We have developed two implementations of the proposed system. The first implementation is based on the BTnode [25] sensor node platform and supports a spatial scope of one hop. We chose the BTnode platform mainly because it provides 256 kB of RAM. Otherwise, the BTnode is similar to a MICA2: the microcontroller is an Atmel AT Mega 128L and the radio is a ChipCon CC1000.

The second implementation uses pre-recorded logs of sensor values instead of real sensors. The log contains sensor data also from neighbor nodes, such that communication between nodes is not required. This implementation runs both on BTnodes and on PCs and is mainly used for evaluation. Apart from these differences, the two implementations are identical. We will refer to these implementations as *DistributedImpl* and *SimulatorImpl*. Below we discuss some important implementation aspects that are shared by both programs.

5.1 Data Structures

The performance of our system significantly depends on an efficient implementation of itemsets and sets of itemsets (i.e., FIB and FIS), as operations on these data structures are frequently performed in the inner loops of algorithms for discovering and merging closed itemsets.

Itemsets are represented as bitvectors, which are implemented as an array of bytes. The size of the array is a compile-time parameter, such that the compiler can apply loop-unrolling to optimize itemset operations. Most operations on itemsets (such as union, intersection, subset tests) operate on bytes (i.e., 8 items at a time) rather than on individual bits and are thus efficient. Note that itemsets in S are densely populated. Eq. 2 implies that a fraction of about $1/|FP|$ of the bits are non-zero. That is, bitvectors are also a space-efficient representation.

Sets of itemsets (i.e., FIB and FIS) only need to support insertion, lookup, deletion of all elements, but *not* deletion of individual elements. Since FIS dominates the memory footprint of our system, it is also important that per-element memory overhead of these data structures is minimized. For example, many typical data structures (linked lists, trees) require one or more pointers per element. Since a typical itemset is rather small in the context of our work (typically < 10 bytes), this would represent a significant overhead. We therefore decided for hash tables that are implemented with a fixed-size array, which requires no per-element memory overhead. The hash function for FIS is based on the bitvector contents of the itemset, while the hash function for FIB is based on the support value of the itemset to support search for supersets (line 19 in Fig. 4).

5.2 Query Compilation

The query compiler reads a query as given in Fig. 1 and generates C code. The output consists of functions to generate events from sensor output according to the definitions given in the query (i.e., lines 4 to 6 in Fig. 1), mapping functions $map_d(), map_t(), map_f()$ that map distances, time intervals and event frequencies to partitions, as well several constant definitions (e.g., for minimum support, temporal and spatial scopes, epoch length). Among the latter is also the size of the bitvectors of itemsets, which is computed using Eq. 3.

6 Evaluation

We study code size, runtime, memory footprint, and the output of the pattern discovery algorithm for a typical query using sensor data logs [26]. In particular, we investigate the trade-off between the scope of the query (i.e., minimum support and number of local events) and resource consumption (i.e., runtime and memory footprint).

6.1 Code Size

We report the size of the code and data segments of *DistributedImpl* in Bytes. The program consists of two main parts. The first part includes algorithms and data structures for discovering frequent itemsets. The second part contains code for reading out sensors and generating events, the protocol for data collection from a one-hop neighborhood, as well as time synchronization. The latter simply uses the request messages of the data collection protocol which are broadcast by the discovery node to synchronize all nodes in the one-hop neighborhood to the time of the discovery node. Code has been compiled by avr-gcc 3.4.5 using optimization flags -O3 -funroll-loops.

Function	Code	Data
pattern discovery	10628	260
data collection, sensors, time sync	5498	707
total	16126	967

6.2 Runtime and Memory Footprint

To evaluate runtime and memory footprint of the pattern discovery algorithm, we use *SimulatorImpl* executing on a BTnode, using sensor data collected during one month from 54 sensor nodes in the Intel Research Lab Berkeley [26]. This dataset was collected with an epoch duration of about 30 seconds (resulting in a total of about 65000 epochs) and contains, among others, temperature and light readings. Using this dataset, we investigate how two key query parameters, namely minimum support and the number of local and context events, affect the resource consumption of our system in terms of runtime and memory footprint.

Fig. 5 shows the relevant query parameters. We consider two types of events: *warm* and *light* events. Each sensor node with a temperature reading > 23 degrees Celsius in an epoch emits a *warm* event in this epoch. Every sensor node with a light reading

$>$ 300 Lux emits a *light* event. Note that with these event definitions we are actually investigating correlations between *states* "it is light" and "it is warm" as discussed in Sect. 3. Both events are declared as both context and local events, which means that we are interested in how light on a node correlates with light and temperature in its neighborhood and how temperature on a node correlated with light and temperature in its neighborhood.

For our experiment, we selected the node with ID 1 as the discovery node executing the pattern discovery algorithm. We obtained very similar results when selecting other nodes as the discovery node. With the above settings, mote 1 generates a *warm* event in about 23% of all epochs and a *light* event in about 14% of all epochs.

```
1 epoch = 30
2 event warm { temp[0] > 23 }
3 event light { light [0] > 300 }
4 cevents {warm, light}
5 levents {warm, light}
6 history = 10
7 distance { near =(0,5), far =[5,10] }
8 time { now=0, recent =[1,4], old=[5,10] }
9 frequency { none=0, some=[1,infty] }
```

Fig. 5. Query used for evaluation.

To evaluate runtime and memory footprint of the pattern discovery algorithm, we compiled a preprocessed version of the sensor data for the first 30254 epochs into the text segment of the *SimulatorImpl* executable that is running on a BTnode (more data didn't fit into the program flash). We then repeatedly ran the pattern discovery algorithm on this data set with different values for MS and MS_e, measuring execution time, the number of itemsets in FIS, the number of closed itemsets among the itemsets in FIS, as well as the number of maximal itemsets among the itemsets in FIS. We also studied how the size of FIS grows over time as more and more blocks are processed. Fig. 6 shows the results. The runtime is given as the ratio between execution time of the algorithm and total time of data collection (i.e., 30254 epochs $\times$ 30 seconds). In all cases, the resulting CPU duty cycle is very small (subfigure (a)). The number of itemsets in FIS strongly depends both on minimum support MS and on error bound MS_e (subfigure (b)). Interestingly, the number of closed itemsets in FIS is also strongly dependent on MS, but less so on MS_e (subfigure (c)). This indicates that reducing MS_e results in the generation of additional itemsets that are frequent in some blocks, but not frequent for the whole data stream. Also note that the number of closed itemsets does also depend on structure of frequent itemsets, such that increasing the error bound MS_e may actually result in more closed itemsets as it is the case in subfigure (c). Also note that only about half of the itemsets in FIS are actually frequent. Again, this indicates that there are many itemsets which are frequent in individual blocks, but not for the whole data stream. The number of maximal itemsets is at most 10 in all experiments (diagram not shown). That is, a very small number of maximal itemsets is sufficient to characterize the "common behavior" observed by the sensor network.

The size of an itemset for the experiment query is 8 bytes (4 bytes for the bit vector, 2 bytes each for the counters c and $c2$). With this, the total RAM required for itemsets in FIS was at most 30 kB in the course of our experiments.

Fig. 7 shows the same diagrams for a simplified query, which considers only *light* as a context event and *warm* as a local event, but is otherwise identical to the query in Fig. 5. We observe a qualitatively similar behavior as in Fig. 6, but at much lower abso-

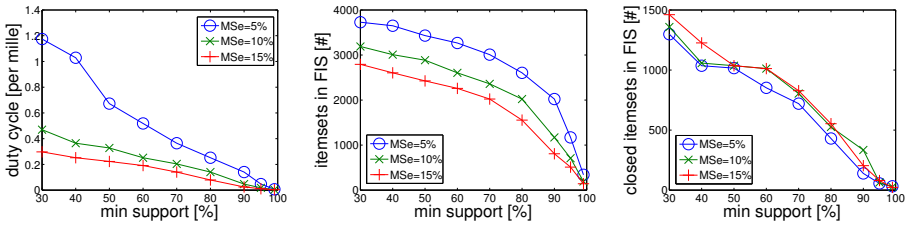

Fig. 6. Results for pattern discovery: (a) runtime (b) size of FIS (c) closed itemsets in FIS.

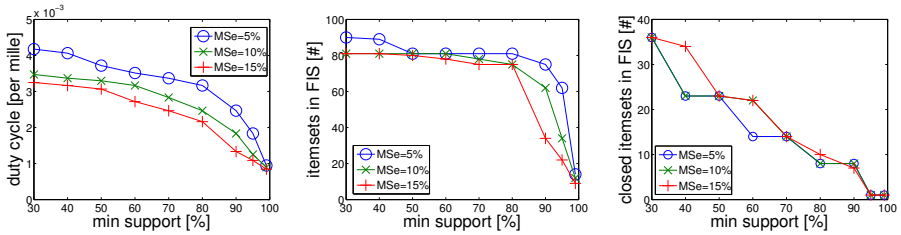

Fig. 7. Results for pattern discovery with simplified query: (a) runtime (b) size of FIS (c) closed itemsets in FIS.

lute values. This illustrates the capability of our system to trade off scope for resource consumption by constraining the search to fewer local events.

To understand the reason for the above quantitative differences, let us estimate how the number of discovered itemsets (i.e., patterns) increases when adding additional context or local events to a query. For this, let us define $Q(E_c, E_l)$ as the number of patterns discovered for a query with context events E_c and local events E_l. Since our algorithm only considers patterns with a single local event, adding a new local event results in an additive increase of patterns:

$$Q(\{e_1\}, \{e_2, e_3\}) = Q(\{e_1\}, \{e_2\}) + Q(\{e_1\}, \{e_3\}) \tag{9}$$

However, when adding an additional context event, we obtain a multiplicative increase in patterns in the worst case:

$$Q(\{e_1, e_2\}, \{e_3\}) \leq Q(\{e_1\}, \{e_3\}) \times Q(\{e_2\}, \{e_3\}) \tag{10}$$

The reason for this is that potentially every frequent pattern that contains only e_1 as a context event could be combined with every frequent pattern that contains only e_2 as a context event to obtain a frequent pattern which contains both e_1 and e_2 as context events. However, in practice this number is often significantly smaller as event patterns with e_2 and events patterns with e_1 have to occur during the same epochs in the data stream – otherwise the combined event pattern may not be frequent.

6.3 Communication Overhead

During the experiment duration of 30254 epochs, only in 8000 epochs a local event occurred on the discovery node (i.e., mote 1). Recall that only when a local event occurs, then the discovery node requests event occurrences during the last $TSCOPE$ epochs (i.e., 10 epochs for our experiment) from client nodes. In our experiment, event occurrences for 12418 epochs have been requested by the discovery node. As each event is represented by a single bit, every client node would have to transmit 24836 bits (since there are two context events defined in the query) or about 3 kB. Assuming that every sensor reading requires one byte, the raw sensor data generated by each node during the experiment would be about 60 kB. Also note that with our approach communication among nodes is constrained to small neighborhoods, whereas traditional data gathering applications require to transmit raw sensor readings through the whole network to the sink.

6.4 Discovered Maximal Patterns

We would expect a strong correlation of the occurrence of *light* and *warm* events on the discovery node with *light* and *warm* events on client nodes. The discovered patterns confirm this expectation. For $MS = 90\%$, for example, we obtain two maximal itemsets that map to the following patterns:

```
(W,now,far,some) AND (W,recent,*,some) AND
(W,old,*,some) : L [96%]

(W,*,far,some) AND (L,now,far,some) AND
(L,{old,recent},*,some) : L [92%]
```

Here, "W" and "L" refer to *warm* and *light* events as defined above. The notations "{... , ...}" and "*" mean that the enclosing term is valid for the set of given partition identifiers or for all possible partition identifiers, respectively.

Packet loss is an issue in most multi-hop data collection sensor networks [9]. In particular, the dataset used for the experiments also had missing entries. In the context of our work, packet loss may affect the correctness of discovered frequent patterns. In particular, three cases can be distinguished. Firstly, a wrong frequency may be reported for a frequent pattern. Secondly, an infrequent pattern may be reported as being frequent. Thirdly, a frequent pattern may not be reported as being frequent. The latter two problems apply predominantly to patterns with a frequency close to MS, where "close to" is a function of the amount of packet loss. Hence, the likelihood of missing frequent patterns due to packet loss can be decreased by reducing MS to a lower value. A quantitative study of this aspect is the subject of future work.

7 Related Work

While data stream mining techniques have been used for other purposes in sensor networks, we are not aware of similar in-network approaches to discover frequent distributed event patterns. There are systems that support *detection* of given distributed

event patterns (e.g., [2, 16]), but this is a fundamentally different problem as mentioned in Sect. 1.

The authors of [17] apply itemset mining to find sensors that show the same value concurrently for significant portions of time, which can be considered as a very specific instance of distributed event patterns. However, their approach is centralized. Resource requirements of their solution are too high to allow an implementation on sensor nodes.

In a more general context, data stream mining techniques have also been applied to outlier detection [4] or to in-network reduction of sensor data streams [3] such that certain properties of the original data stream are preserved. However, these are fundamentally different problems. Also, while the authors claim that their algorithms can be implemented on resource-constrained sensor nodes, they resort to simulations.

Another approach that is loosely related to our work is *distributed regression* [12], where sensor nodes cooperate locally to fit a global function to their measurements. Implicitly, such a global function represents the correlation between sensor data of different nodes. However, this work is based on continuous sensor time series and makes the fundamental assumption that sensor data is strongly correlated both spatially and temporally. In contrast, our approach is based on discrete events and we make no assumptions about the correlation of sensor data – instead, we want to find out whether and how events on different sensor nodes are correlated.

There is a large amount of work regarding discovery of frequent itemsets from a data stream. Many approaches are based on sliding windows (e.g., [6, 8]), where frequent itemsets are discovered from a small, moving fraction of the data stream. However, we are interested in discovering patterns from the whole data stream. Several proposals exist for this problem (e.g., [7, 10, 15, 13, 19]). However, these approaches use synopsis data structure in addition to the sought frequent itemsets, resulting in a memory footprint that led us to develop an approach that fits the specific constraints of sensor nodes. For discovery of closed itemsets from a small block of itemsets, we borrowed techniques from existing multi-pass algorithms, most notably [5].

8 Conclusions

We presented a novel in-network knowledge discovery technique that supports the discovery of frequent distributed event patterns in sensor networks, where event patterns characterize the spatial and temporal correlations between events observed by sensor nodes in a confined network neighborhood. To deal with the constrained resources of sensor nodes, our system offers a declarative query language to specify the level of detail and the scope of sought patterns, thus offering a turning knob to trade off detail and scope for resource consumption. We implemented our proposal on the BTnode platform and and showed that the resources of this platform are sufficient to handle typical problem instances. We also showed that by reducing the scope of the query we could decrease resource consumption.

References

1. R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, L. Krishnamurthy, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and Deployment of Industrial Sensor Networks: Ex-

periences from the North Sea and a Semiconductor Plant. In *Sensys 2005*, San Diego, USA, November 2005.
2. S. Ahn and D. Kim. Proactive Context-Aware Sensor Networks. In *EWSN 2006*, Zurich, Switzerland, February 2006.
3. H. Akcan and H. Brönnimann. Deterministic Data Reduction in Sensor Networks. In *MASS 2006*, Vancouver, Canada, October 2006.
4. J. Branch, B. Szymanski, C. Gianella, R. Wolff, and H. Kargupta. In-Network Outlier Detection in Wireless Sensor Networks. In *ICDCS 2006*, Lisboa, Portugal, July 2006.
5. D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In *ICDE 2001*, Heidelberg, Germany, April 2001.
6. J. H. Chang and W. S. Lee. Finding Recent Frequent Itemsets Adaptively over Online Data Streams. In *SIGKDD 2003*, Washington, USA, August 2003.
7. W. Cheung and O. R. Zaiane. Incremental Mining of Frequent Patterns Without Candidate Generation or Support Constraints. In *IDEAS 2003*, Hong Kong, China, July 2003.
8. Y. Chi, H. Wang, P. S. Yu, and R. M. Muntz. Moment: Mainatining Closed Frequent Itemsets over a Stream Sliding Window. In *ICDM 2004*, Brighton, UK, November 2004.
9. J. Choi, J. Lee, M. Wachs, and P. Levis. Opening the sensornet black box. Technical Report SING-06-03, Stanford Information Networks Group, 2006.
10. C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In *NSF Workshop on Next Generation Data Mining 2002*, Baltimore, USA, November 2002.
11. K. Gouda and M. J. Zaki. Efficiently Mining Maximal Frequent Itemsets. In *ICDM 2001*, San Jose, USA, November 2001.
12. C. Guestrin, P. Bodik, R. Thibaux, M. Paskin, and S. Madden. Distributed Regression: an Efficient Framework for Modeling Sensor Network Data. In *IPSN 2004*, Berkeley, USA, April 2004.
13. R. Jin and G. Agrawal. An Algorithm for In-Core Frequent Itemset Mining on Streaming Data. In *ICDM 2005*, New Orleans, USA, November 2005.
14. R. J Bayardo Jr. Efficiently Mining Long Patterns from Databases. In *SIGMOD 1998*, Seattle, USA, June 1998.
15. H.-F. Li, S.-Y. Lee, and M.-K. Shan. An Efficient Algorithm for Mining Frequent Itemsets over the entire History of Data Streams. In *First Intl. Workshop on Knowledge Discovery in Data Streams 2004*, Pisa, Italy, September 2004.
16. S. Li, S. H. Son, and J. A. Stankovic. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In *IPSN 2003*, Palo Alto, USA, April 2003.
17. K. K. Loo, I. Tong, and B. Kao. Online Algorithms for Mining Inter-Stream Associations from Large Sensor Networks. In *PAKDD 2005*, Hanoi, Vietnam, May 2005.
18. S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In *OSDI 2002*, Boston, USA, December 2002.
19. G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *VLDB 2002*, Kong Kong, China, August 2002.
20. K. Römer. Distributed Mining of Spatio-Temporal Event Patterns in Sensor Networks. In *Workshop on Middleware for Sensor Networks*, San Francisco, USA, June 2006.
21. R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a Sensor Network Expedition. In *EWSN 2004*, Berlin, Germany, January 2004.
22. J. Wang, J. Han, and J. Pei. Searching for the Best Strategies for Mining Frequent Closed Itemsets. In *SIGKDD 2003*, Washington, USA, August 2003.
23. M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *NSDI 2004*, Boston, USA, March 2004.
24. M. J. Zaki and C. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. In *SDM 2002*, Arlington, USA, April 2002.
25. BTnodes. www.btnode.ethz.ch.
26. Intel Lab Sensor Data. http://berkeley.intel-research.net/labdata/.