Optimizing Query Processing in Cache-Aware Wireless Sensor Networks

Mario A. Nascimento
Univ. of Alberta, Canada
mn@cs.ualberta.ca

Romulo A.E. Alencar
Univ. of Fortaleza, Brazil
romulo@assessus.com.br

Angelo Brayner*
Univ. of Fortaleza, Brazil
brayner@unifor.br

ABSTRACT

It is a well known fact that minimizing energy consumption in Wireless Sensor Networks (WSNs) is crucial for its usability; and minimizing the flow of data is one way to achieve that. Most WSN models assume the existence of a base station where query results could in principle be cached, however, the opportunity for re-using such cached data for minimizing data traffic in the WSN has not been well explored thus far. Aiming at filling this gap, we investigate the *cache-aware query processing problem in WSNs*. We propose an approach that first clips an original rectangular query area into a polygon by selecting a suitably good subset of the cached queries for reuse. Next, this polygon is partitioned into rectangular sub-queries that are then submitted to the WSN. Finding the cost-wise best combination of polygon clipping and rectangular partitioning amounts to a highly combinatorial problem that justifies the use of efficient and effective heuristics. This paper presents algorithms that are used within a cost-driven optimization search to find a good "query-plan". Algorithms for maintaining the cache consistency are also presented. Our proposal does not dependent on any particular algorithm for processing queries in a WSN; as long as there is a well-defined a cost-model for the same, any proposal can be used. Experimental results show that our heuristic algorithms are orders of magnitude faster than an exhaustive search, yield no more than 10% loss compared to the optimal query cost, and are never worse than the two obvious alternatives, i.e., not using the cache at all or using all of it.

1. INTRODUCTION

A typical Wireless Sensor Network (WSN) is comprised of a set of several identical sensor nodes with limited CPU and storage capacity plus one base station which is assumed to have more resources, e.g., more CPU power, additional storage space and a relatively large, or possibly continuous, energy supply. All nodes are able to communicate wirelessly within a certain range and are assumed to be aware of its neighbors, i.e., other nodes within its wireless communication range. No node is assumed to have full knowledge of the network, including the location of other nodes. The base station is a partial exception in the sense that it is assumed to have some high-level knowledge of the WSN, e.g., node density, wireless range, etc., which is used in the query cost model. All decisions, such as data packet routing and collision resolution, are to be made locally by the nodes involved. Each node is capable of observing its surroundings and capture one or more measures relative to the same. Many applications for WSN have been discussed in the literature, e.g., [4, 15, 22] to name but a few. In a typical example, nodes are capable of sensing environmental attributes, such as temperature, light and humidity, allowing researchers to monitor an area of interest remotely, inconspicuously and continuously.

Once the WSN is active and nodes are collecting data, a number of queries can be issued, e.g., spatio-temporal range queries, join queries and aggregate queries, for which a large number of algorithms have been proposed, e.g., [1, 3, 11, 14, 20, 19, 25]. Most of those algorithms also assume the existence of a base station from where queries are injected into the WSN and to where the queried data is returned. In this scenario, there exists a clear opportunity for reducing query cost through caching query results at the base station and subsequently re-using such cached data when new queries are posed.

While in a traditional database system the use of cache is for maximizing the system's throughput, we aim at minimizing the energy cost of query processing. Nonetheless, even though it is not our main goal, the approaches we propose will likely also yield some improvement in the WSN throughput. As well, since our main interest is not in the in-network query processing itself, we do not make any strong assumption on how in-network query processing is done, rather we only assume that a cost model for the adopted technique is available. Typically, algorithms for query processing in WSN borrow the minimum bound rectangle abstraction from spatial databases and assume that the query is a rectangle within the monitored area; we follow suit and make similar assumptions.

Our main problem in this paper is to minimize the energy cost of processing a rectangular query Q with respect to a WSN that requests the most recent values of (possibly all) sensed attributes of sensor nodes located within Q[1]. To

*On sabbatical leave at the University of Alberta.

Technical Report TR 09-05. March 2009. Dept. of Computing Science. University of Alberta. Canada.

[1]To simplify notation we refer to a *query* as well as to its

illustrate the potential gain of using cached data consider for instance the scenario depicted in Figure 1, where Q is a new query and $P = \{P_1, P_2, P_3\}$, represents previously processed queries. Assuming the cached data is still valid, it is clear that only nodes located in the "clipped" query region Q', need be contacted. Since, the number of nodes within the area of Q' is bound to be smaller than those within Q's area, processing Q' is bound to be energy-wise less expensive than processing Q.

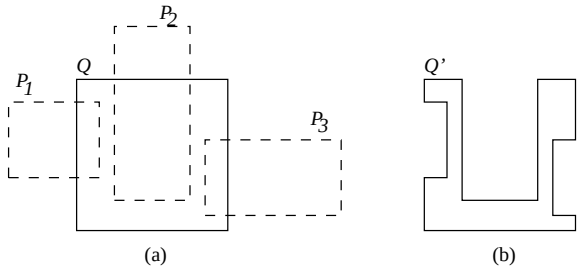


Figure 1: Query Q, and clipped query Q' after considering previous queries P_i.

One trivial way to solve this problem is to find the minimum bounding rectangle of Q', denoted by $R(Q')$, and process the query using $R(Q')$ instead of Q. Clearly, that would not necessarily minimize query processing cost as $R(Q')$ would often be likely equal to $R(Q)$ (as in the case of Figure 1).

Another straightforward way to solve this problem is to "flood" the query area and have only the sensors within the query area respond to the query. There are two main drawbacks in this approach. One is that in order to decide whether it should respond or not to the query, a sensor would need to know whether it is actually inside the query area. Even though efficient linear time algorithms exist to answer the point-in-polygon problem [17], they are linear in the size of the polygon description. In general the contour of the clipped query Q' can be very complex and also have holes, that is, its description may be quite large. Recall that a node's CPU is assumed to be fairly limited, i.e., even a linear time algorithm on a fairly large input can be sufficiently demanding to be executed at the node. Another problem is that the query area may contain a large number of nodes, and each one (plus possibly others sufficiently close to the query's boundary) would receive the potentially large description of the query's polygon. This would imply a potentially large number of large messages flowing in the network. Given that the the larger the message the more energy is needed to transmit it, this flooding-based approach is also non-practical.

Thus a more sophisticated solution is needed to adequately address the *cache-aware query processing problem in WSN* motivated above. The solution we propose assumes a query processing framework, illustrated in Figure 2, located on the WSN's base station, and whose functioning is as follows. The user poses a query Q at the base station. The Query Processor requests from the Cache Manager the set P of previous queries that intersect Q. The Cache Manager uses the Cache Index to quickly obtain the set P of relevant (cached) queries. (Although other possibilities could be used, an obvious alternative for the Cache Index would be an R*-tree [2].) The Cache Manager inspects whether any of the queries in P contain stale data. If needed it updates the Cache Index, and returns the set P' of valid relevant queries that intersect Q, along with the union respective cached datasets $D(P'_i), \forall P'_i \in P'$. Having Q and P', the Query Processor uses the services of a Query Optimizer[2] for two inter-dependent tasks aiming at reducing the cost of processing Q: (1) to determine which set $P'' \subseteq P'$ to use, and thus determine the clipped query Q' and, considering Q' and P'', (2) to partition Q' in order to determine a set of sub-queries Θ. Finally, the Query Processor receives Θ from the Query Optimizer, submits them to the WSN, combines their results with the cached datasets of the queries in P'' and returns the final query result to the user.

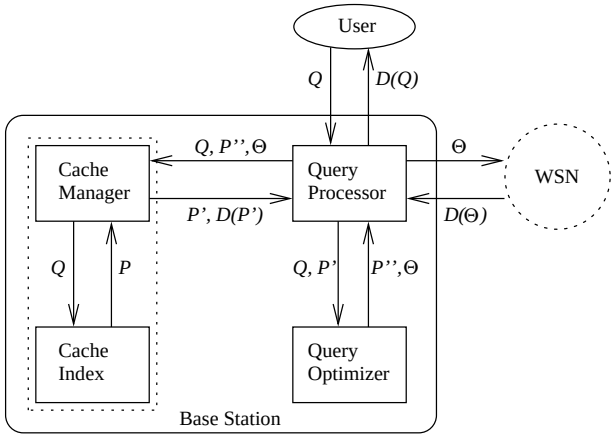


Figure 2: Query processing framework within the base station of a cache-aware WSN.

In the context of this framework, we focus mainly on the Query Optimizer module, i.e., given a query Q and the set P', we aim at determining the sets P'' and Θ that minimize the energy cost for processing Q. To the best of our knowledge this cache-aware query processing problem has not been addressed in the related literature. In summary, this paper presents the following contributions:

- We define the cache-aware query processing problem and argue that its highly combinatorial nature justifies the development of efficient sub-optimal solutions.
- We propose cost-oriented heuristic algorithms that are able to rapidly find good solutions (and often the best one) to the cache-aware query processing problem.
- Finally, we propose low-overhead algorithms for maintaining the WSN's cache current and consistent.

The remainder of the paper is structured as follows. In the next section we briefly review some of the related research. In Section 3 we discuss in detail the cached data selection problem and its inherently complex nature in the context of in-network query processing. Heuristic algorithms that are combined to find a cost-wise good strategy for solving

area using the same variable

[2]We do not claim that this is a full-fledged query optimizer in the usual sense, we use this term simply for the sake of argumentation.

the cache-aware query processing problem are presented in Section 4. Next, Section 5 introduces algorithms for maintaining cache consistency and freshness. Section 6 presents empirical evidence that our proposed solutions can be both effective and efficient when compared to the obvious alternatives of not using the cache at all or using all of it. Finally, section 7 concludes the paper.

2. RELATED WORK

Data caching has been a well-know mechanism for enhancing query throughput in database systems in general for quite some time [9]. In a typical setting, the cache offers much faster access (e.g., nsec. vs. msec.) but is much smaller than the main storage (e.g., a few MBytes vs. many GBytes), and it is used so that requests to frequently accessed data can performed more efficiently. In distributed database systems data caching can also reduce communication costs, since the amount of data flowing in the network is minimized. Our problem is similar but, as shall be clear from our discussion in the forthcoming sections, it has some important differences. Several new issues, such as data replacement policy and validation, as well as their close relations to lower level transaction management, need be carefully considered when caches are used. A thorough discussion on the topic with respect to traditional database systems is beyond the scope of this paper. We note that in the context of WSN many of these issues have not been carefully considered yet.

Cache strategies can be classified into two groups [12]: physical and logical caching. Physical caching, arguably the most common one, replicates in the (faster) cache pages or tuples identified as hot spots. In order to identify hot spots, heuristics based on access frequency and cost/benefit analyses can be used. Logical caching, on the other hand, replicates data that belong to query results. For example, a logical caching mechanism where the main idea is to use query semantics for organizing the cache was proposed in [7]. Our approach may be likened to that one in the sense that the authors model cached data and queries as "geometric" constraints onto the data space, whereas in our case we deal with queries as actual polygons in the Euclidean space. Another difference is that our approach aims at minimizing data flowing in the WSN using past queries whereas in [7] the authors aim at optimizing cache usage and replacement in data servers using the query semantics.

The context in which we consider the use of cache, namely WSN, is fairly novel itself, and a relatively small number of papers have been published in this area. Most of them are related to the networking rather than the database aspect of the problem. In that context data caching can be applied to minimizing packet transmissions in the network and consequently reducing power consumption. For instance, in [18] the authors propose a strategy that emulates a data caching mechanism by predicting when a sensor's observed data will change. Such a technique aims at avoiding requests to redundant data but it might impact negatively the correctness of query results, because some unpredicted change may not be propagated in the WSN. Another strategy proposed in the same paper is to aggregate the flow of redundant (replicated) values in a single message. Again, this may have an adverse effect on the query result as a single message failure becomes responsible for several values from different nodes being lost. While link failures can always affect query processing in a WSN, we minimize their effect by placing the cache on the base station which needs to relay the query results to the user anyway.

In [8] the authors discuss how to explore a natural hierarchy of entities within a WSN using an XML framework. One of the main focus of that work is on the so-called Query-Evaluate-Gather technique which enables one to not only find relevant queried data within a node but also how to gather the missing parts. We, on the other hand, assume that no data is cached on the actual resource-challenged sensor nodes. More importantly the authors focus only on improving query throughput, ignoring the energy cost factor, whereas in this paper, by virtue of having the cache at the base station, we focus on minimizing the energy cost of query processing in the WSN.

A work that assumes a context closer to ours, i.e., it is database-oriented in nature and geared towards an WSN has been presented in [16]. The authors assume that the WSN uses a tree-based routing protocol and propose that several nodes in the network to be used for caching data. The problem of choosing these so-called cache-nodes reduces to finding a Steiner Minimal Tree (SMT), i.e., a tree that connects all points with minimal length. Since finding an SMT is a NP-Hard problem, the authors propose a sub-optimal solution, called Steiner Data Caching Trees (SDCT). Unfortunately the bottleneck of the proposal is that it is as robust as the cache-nodes, i.e., once they become unavailable (which can happen for a variety of reasons), not only data is lost but also another SDCT needs to be reconstructed from scratch. By relying on the base-station our approach is relatively free from side-effects when individual nodes become unavailable.

3. CACHE-AWARE QUERY PROCESSING

3.1 Background

Before proceeding further we need to decide on a good model for the cost of processing queries in a WSN to be used within the Query Optimizer. In the remainder of this paper we re-use the SWIP framework for processing rectangular queries within a WSN [5]. While other query processing algorithms could be used, the main advantage of using the SWIP framework is that it offers an intuitive and accurate model that can be used for guiding the optimization problem we have. Due to limited space we skip the details of the cost model and refer to the interested reader to [5]. SWIP's query processing approach can be summarized in the following four major tasks, which form the main components of SWIP's cost model:

- S1: The query is sent from the base station[3] node to a coordinator node.
- S2: The query is partially flooded within the query area.
- S3: All nodes in the query area send their data to the coordinator node.
- S4: The coordinator node returns the gathered data to the Base Station node (likely to the same path used in step (S1)).

[3]In [5] the query was allowed to be posed from any node.In our case we assume the more typical case of a single node, the base station, being always query originator.

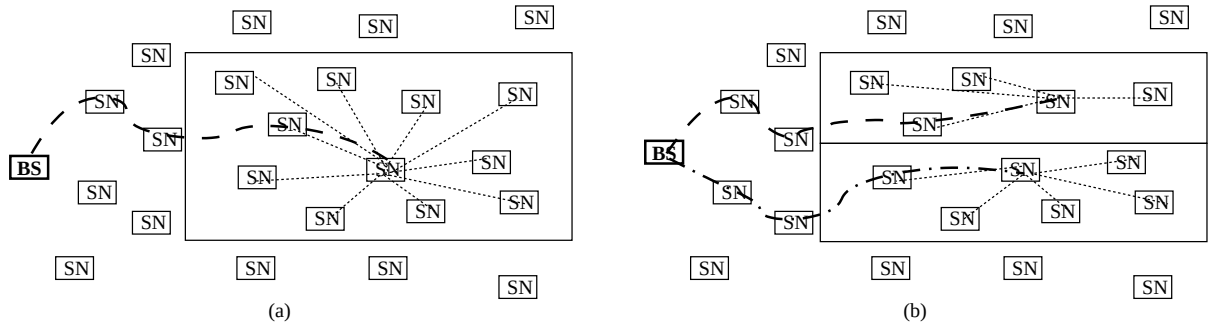


Figure 3: Illustration of the SWIF approach, where SN is a Sensor Node, BS is the base station, the dashed line is the path traversed by the query (and subsequently by its results) and the query region is denoted by the marked rectangular areas.

An important observation to be made at this point is that steps S1 and S4 can be considered (necessary) overheads to the query cost. Consider Figure 3(b), where the total queried area is the same as in Figure 3(a), but instead of one query we use two queries. Each and every node that is contacted and contributes to the answer in case (a) is also contacted in case (b). Given that during the local flooding shortest paths are found we consider the costs of steps S2 and S3 to be comparable in both cases. The main difference between those two cases is in the costs of steps S1 (mainly) and S4. In case (a) only one message is sent in step S1, whereas in case (b) two messages of the same size and routed through different paths are needed. In step S4, the query result is returned in a single message for case (a) but also in two messages, though each one carries less data. It should be intuitive by now that a larger number of smaller queries is probably not as good as a smaller number of larger queries due to the overhead that each query imposes As we shall see shortly this observation plays a key role in the optimization process we propose.

Let P be the set of all cached queries each having a possibly distinct validity period. We assume that all cached queries are pair-wise disjoint. This is a natural assumption because only the most recent data of any sensor needs to be cached. In fact, as it shall become clear throughout the paper, this assumption is enforced by our cache consistency algorithms presented in Section 5. The set $P' = \{P'_1, P'_2, \ldots, P'_M\} \subseteq P$ of queries which are still valid and such that $P'_i \bigcap Q \neq \emptyset, \forall P'_i \in P'$, is the set of relevant valid queries with respect to Q. We make the natural assumption that the resulting data set $D(P'_i)$ for each query P'_i is cached in the base station as well. The case of cached queries no longer valid is discussed in Section 5, but for now it suffices to ignore them.

The following definitions and Table 1 summarizes some of the notation used throughout the rest of the paper. Note that, unless otherwise noted, whenever we refer to a polygon we mean its boundaries as well as its interior points.

Definition 1. *Given two polygons R and T, $R \bigcup T$ denotes the polygon given by the union of all points interior to R or to T as well all as their boundaries. Similarly, given a set P of polygons P_i, their union is denoted as $\mathcal{U}(P) = \bigcup_{P_i \in P} P_i$.*

Definition 2. *Given a polygon R and a set of polygons S, such that $R \bigcap S_i \neq \emptyset, \forall S_i \in S$, we denote as $R \oplus S$ the clipping of R with respect to S. Informally, $R \oplus S$ is the portion of R not overlapped by any $S_i \in S$.*

Definition 3. *Given a rectilinear polygon R we denote as its partition a set $\rho(R)$ of rectangles such that their pair-wise intersection is null and their union is equal to R.*

Notation	Meaning
Q	A rectangular query wholly contained within the monitored area
P	Set of all relevant queries in the cache, i.e., $\{P_1, P_2, \ldots P_M\}$
P'	Set of relevant valid queries wrt Q ,i.e., $\{P'_i$ s.t. $P'_i \in P \wedge P'_i \bigcap Q \neq \emptyset\}$
P''	Subset of P' (to be determined by the Query Optimizer module)
Θ	Set of queries given by $\rho(Q \oplus \mathcal{U}(P''))$ to be submitted to the WSN
$D(X)$	Data set associated with a query X
$C(X)$	Cost (energy-wise) for processing query X in the WSN

Table 1: Notation.

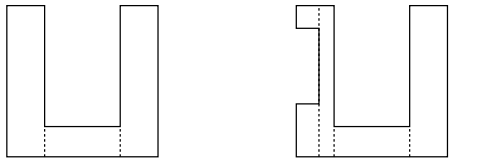
3.2 The Main Problem(s)

The problem we investigate is how to minimize the cost of processing a query Q which requests the *current observations of all sensor nodes within Q's area.* Considering only the set P', this amounts to a request for the current data from the sensors in the area denoted by $Q' = Q \oplus P'$. We can take advantage of previously proposed algorithms for processing rectangular queries in WSN by determining $\Theta = \{\theta_1, \theta_2, \ldots, \theta_T\} = \rho(Q')$. Once this is done, one can use suitable algorithms to execute each subquery in Θ, and obtain the answer for Q, i.e., $D(Q) = (\bigcup_{\theta_i \in \Theta} D(\theta_i)) \bigcup (\cup_{P'_i \in P} D(P'_i))$. Under the reasonable assumption that data for previous queries can be obtained at no cost within the base station, the cost of processing the original query Q is $C(Q) = \sum_{\theta_i \in \Theta} C(\theta_i)$. Hence, our ultimate goal is to determine the set Θ that minimizes $C(Q)$.

However, before setting out to determine Θ one must question whether it is wise to use the full set of relevant valid queries P' or rather a subset P'' thereof. We claim that very often one may be better off not using P', due to the overhead that each additional query imposes. For each element

$P'_i \in P'$ considered for re-use the query area is reduced but the number of sub-queries in Θ is further increased. However, each sub-query carries the burden of an inherent overhead, namely that of dispatching and receiving each query, thus adding sub-queries in Θ—an unavoidable consequence of adding P'_i to P''—is worth it if and only if the amount of in-network flow saved is larger than the overhead added.

We illustrate the intuition behind this argument through Figure 4 which shows two equally feasible alternatives for clipping and partitioning the instance problem in Figure 1. For each Q' we have a partitioning, i.e., a set Θ already defined. Note that this is just for the sake of illustration, and in general the set Θ is not unique.



(a) $Q' = Q \oplus P_2$ (b) $Q' = Q \oplus (P_1 \bigcup P_2)$

Figure 4: Possible partitioning schemes with respect to Figure 1.

From Figure 4(a) it is quite clear that using P_2 reduces Q's area considerably and therefore it is likely worth paying the overhead imposed by the three resulting sub-queries. However, when P_1 and P_2 are used at the same time not only query area is reduced by only a relatively small amount as compared to the previous case, but it increases the number of sub-queries from three to five. Thus, in this case, one would likely be better off not re-using both P_1 and P_2 but rather only P_2 as the set P'' instead. This scenario, although simplistic, suggests this problem should be explored from an optimization perspective, and that is exactly the main point of this paper.

However, the reasoning above assumes one is able to solve two sub-problems in an efficient manner, namely: clipping a polygon with respect to another one and partitioning a rectlinear polygon. We discuss those in the following.

3.3 Query Clipping and Partitioning

For the polygon clipping problem we use the well-known GPC public implementation[4] which is based on a polygon clipping algorithm presented in [24]. Furthermore, we need to evaluate the query cost yielded by a given set P''.

There are many goals one can aim at when partitioning a polygon. As discussed earlier each obtained rectangle in the partitioning of the clipped query will correspond to a sub-query, and in order to minimize the query processing overhead we restrict ourselves to finding a minimum cardinality decomposition of Q'. Interestingly, this produces a positive side-effect; it helps improving query performance from a networking perspective as well, e.g., by minimizing the effect of packet collisions in the network.

Covering a rectlinear polygon is a well-known problem in computational geometry [13] but is not a desired solution for our context. A *covering set* allows non-null intersections among the covering rectangles, which in our setting would imply that some nodes may be requested to participate in the same query more than once, thus expending more energy than needed. While this could be circumvented by somehow making nodes not respond twice to the same query, there is a more severe problem. When the polygon has holes, which may well be the case of Q', the covering problem has been proved to be NP-complete [6]. Therefore we settle for finding a *partition* of the polygon Q', i.e., a set of non-intersecting rectangles whose union is equal to Q'. It is important to note that in the related literature there is a number of proposals for similar problems under the same denomination. However, most aim at minimizing other objective functions, e.g., the edge length of the edges of the partition set, instead of minimizing the *cardinality* of the partition set as we do.

There exists a $O(v^{1.5} \log v)$ time optimal algorithm for this problem, where v is the number of vertices of the polygon to be partitioned [21]. Unfortunately its implementation is quite complex. An alternative approximative algorithm, with complexity $O(v \log v)$ and much simpler to implement has been presented in [23]. Furthermore it has also been argued in [10] that the partitioning problem has "an $\Omega(v \log v)$ lower bound on the time-complexity. The result holds for any decomposition, optimal or approximative." That is, the algorithm we chose, albeit not guaranteed to provide the optimal partition is as efficient as it can possibly be in terms of time complexity. Considering the fact that base station may not be computationally resourceful, and furthermore that within a WSN environment one should use every chance to save energy as opportunistic as it might be, we consider that settling for a sub-optimal but effective and efficient algorithm is an worthy trade-off.

4. THE CACHE-AWARE QUERY OPTIMIZATION PROBLEM

In order to address the problems discussed in the previous section, we begin by assuming Q and P' are given. How one determines the set $P' \subseteq P$ bears no impact on the following discussion, and thus we defer a detailed discussion of that task to Section 5. If $C(X)$ is the cost of processing a query X, and assuming the use of cached data at the base station can be done at null cost, the optimization problem at hand is to find the subset P'' and the rectangular partitioning Θ that minimizes $C(Q) = \sum_{\theta_i \in \Theta} C(\theta_i)$. As it shall become quite clear in the following, this is a highly combinatorial problem, for which heuristic solutions are justified in the interest of query processing time.

We now have two problems for which we seek effective and efficient solutions. One problem is to find the set P'' that yields the optimal query cost assuming a particular polygon rectangular partitioning algorithm, which turns itself out to be the second problem.

Given the arguments above, the main problem is we need to solve is finding which set $P'' \subseteq P'$ of previous queries to use. A trivial but impractical way to solve this problem is by brute-force. In this case one would to consider all elements of P''s powerset, whose cardinality is $2^{|P'|}$, and for each one find a partition Θ—we discuss how to do this in the next section—and then select the set P'' that minimizes $C(Q)$. A more practical approach is to start with an empty (full) set P'' and judiciously increase (decrease) its size until no improvement can be achieved, at which point the best solution found so far is adopted. This idea forms the core of the branch-and-bound approach we present in this section.

In our preliminary experiments we rarely saw a case where

[4]http://www.cs.man.ac.uk/∼toby/alan/software/

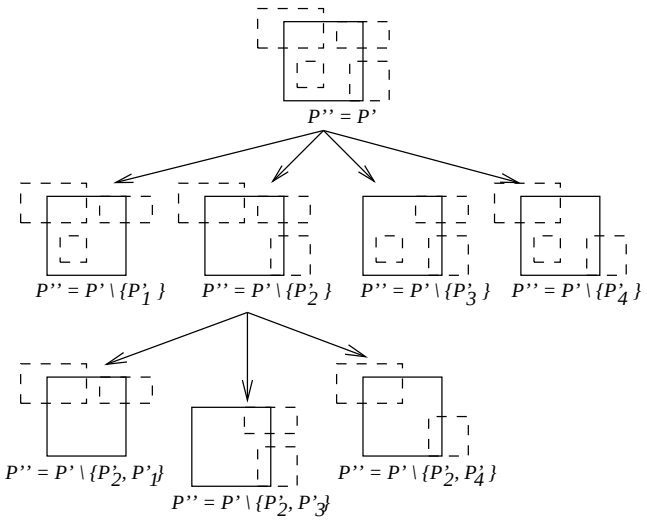


Figure 5: Illustration of the branch-and-bound search for a good set P'' and corresponding set Θ.

using no available cache would be the best option. This suggests that instead of starting with the original query and incrementally evaluating the solutions obtained by considering more and more intersections, it may be more productive to work the other way around.

We build our search tree by considering $P'' = P'$, i.e., all possible $|P'|$ intersections between P' and Q in the root node; we define the root node's height to be 0. Then at height 1 we consider all possible intersections using $|P'| - 1$ elements from P'. In general, at height $K < |P'|$ of the tree, each node represents the intersection between Q and a set P'' containing a unique set of $|P'| - K$ elements of P' Furthermore, each such node will branch out yielding $P - K$ sub-trees. Clearly, if no branch is pruned *and memoization* is used properly, then all elements of P''s powerset will be evaluated, and the one with minimum cost can be chosen as the best solution. It is important to note that for each one of $2^{|P'|}$ possibilities, one needs to compute a Q' as well as its partitioning. While such an exhaustive search is conceptually correct and guaranteed to find the optimal solution, it is not practical. Thus we propose to use a branch-and-bound technique to find a solution, possibly sub-optimal, to this search problem.

The search works as follows (we use Figure 5 to support the explanation). The root node uses $P'' = P'$, we compute its cost $C(Q)$ and save it as the *incumbent cost*. Note that it implies performing both appropriate clipping as well as partitioning operations. Next we "open" the node corresponding to the incumbent solution, i.e., obtain all the possibilities using $|P'| - 1$ intersections and compute their cost. Any un-opened node, i.e., one not yet explored, that has a cost lower than the incumbent is a candidate node to be opened as it indicates a better solution that the current incumbent one. Following a greedy mode, we choose the node with lowest cost, update the incumbent cost to that of the chosen node, and open it, leading us to one level further down the search tree. In Figure 5, the chosen node is the one corresponding to using $P' \setminus \{P'_2\}$. As long as there are un-opened nodes with cost lower than the current incumbent, the search proceeds down the tree. Once all un-opened nodes fail this test, we return the best solution thus far, i.e., the set P'' and partitioning Θ associated with the node which yielded the current incumbent cost. In the case of Figure 5, if none of the newly opened nodes has a better cost than the current incumbent then the search would stop and report $P'' = P' \setminus \{P'_2\} = \{P_1, P_3, P_4\}$ and the corresponding partition of $Q \oplus \mathcal{U}(P'')$ as the solution for the search.

Algorithm 1 shows the pseudo-code for the searching procedure just discussed. As we shall confirm shortly if provides very good solutions at a very small fraction of the exhaustive search cost.

Algorithm 1 B+B: a branch-and-bound search for P''

Input: Q and set P'
1: create a node and set it as an open incumbent node
2: set $P'' = P'$
3: compute a set of sub-queries $\Theta = \rho(Q \oplus \mathcal{U}(P''))$
4: compute the incumbent cost $C^* = \sum_{\theta_i \in \Theta} C(\theta_i)$
5: set the incumbent solution $\langle P'', \Theta \rangle$
6: **while** there is at least one open incumbent node **do**
7: select as incumbent node the node closer to the root and set it as closed
8: **for** each element $P'_i \in P''$ **do**
9: set $P'' = P'' \setminus \{P'_i\}$
10: set $\Theta = \rho(Q \oplus \mathcal{U}(P''))$
11: compute the cost $C = \sum_{\theta_i \in \Theta} C(\theta_i)$
12: **if** $C \leq C^*$ **then**
13: set the incumbent solution to $\langle P'', \Theta \rangle$
14: set the current node as an open incumbent node
15: set $C^* = C$
16: **else**
17: set the current node as closed
18: **end if**
19: **end for**
20: **end while**
21: **return** the incumbent solution

While Algorithm 1 will stop at a locally optimum solution and typically be much faster than the exhaustive search, one can be even more aggressive. Another possibility is to start the search as the branch-and-bound approach does, i.e., using all intersections. Then at each step remove the smallest one, i.e., the intersection that contributes the least to the savings. If this improves the query cost, i.e., the overhead yielded by the removed query was larger than its savings, we proceed removing the next smallest intersection, and so on and so forth as long as the total query cost is not increasing. This greedy approach is reflected in Algorithm 2.

With Algorithm 2 in mind, one can think of yet another fairly intuitive alternative. A previous query in set P' is worth using only if it saves more cost than it induces through the overhead of the resulting sub-queries. Then if one uses $P'_i \in P'$ that yields the largest intersection with the current Q' chances are that the gain may be larger than the added overhead. This can be repeated iteratively in a greedy mode. Using this observation we propose Algorithm 3 which is a rather simple modification of Algorithm 2.

By the way they start, it is easy to see that the solution by both the B+B and the GrF algorithms above cannot be worse than using all of the cache without any further optimization. Likewise the GrE algorithm cannot deliver a worse solution than not using any cache at all. Furthermore, one

Algorithm 2 GrF: a greedy search for P''

Input: Q and set P'

compute a set of sub-queries $\Theta = \rho(Q \oplus \mathcal{U}(P'))$
compute the incumbent cost $C^* = \sum_{\theta_i \in \Theta} C(\theta_i)$
set the incumbent solution to $\langle P', \Theta \rangle$
set $Q' = Q$
repeat
  find $P'_i \in P'$ that minimizes the area $(Q' \bigcap P'_i)$
  set $P' = P' \setminus \{P'_i\}$
  compute a set of sub-queries $\Theta = \rho(Q' \oplus \mathcal{U}(P'))$
  compute the cost $C = \sum_{\theta_i \in \Theta} C(\theta_i)$
  if $C \leq C^*$ **then**
    set $C^* = C$
    set the incumbent solution to $\langle P', \Theta \rangle$
    set $Q' = Q' \oplus P'_i$
  end if
until $P' = \emptyset$ or $C > C^*$
return the incumbent solution

Algorithm 3 GrE: another greedy search for P''

Input: Q and set P'

compute the incumbent cost $C^* = C(Q)$
set $P^* = \emptyset$
set $\Theta = \{Q\}$
set the incumbent solution to $\langle P^*, \Theta \rangle$
repeat
  find $P'_i \in P'$ that maximizes the area $(Q \bigcap P'_i)$
  set $P' = P' \setminus \{P'_i\}$
  set $P^* = P^* \bigcup P'_i$
  compute a set of sub-queries $\Theta = \rho(Q \oplus \mathcal{U}(P^*))$
  compute the cost $C = \sum_{\theta_i \in \Theta} C(\theta_i)$
  if $C \leq C^*$ **then**
    set $C^* = C$
    set the incumbent solution to $\langle P^*, \Theta \rangle$
  end if
until $P' = \emptyset$ or $C > C^*$
return the incumbent solution

can anticipate that a bad (greedy) decision by the GrE algorithm is going to be more costly than a bad decision by the GrF algorithm. In the GrF algorithm the change in the set P'' from one iteration to the next is by construction relatively small whereas in the case of the GrE algorithm it is exactly the opposite case, hence yielding a bigger change in the solution. Indeed, this will be confirmed in the experiments we present in Section 6.

5. MAINTAINING CACHE CONSISTENCY

In order for the cache be effective its data needs to be kept consistent and as fresh as possible. In this section we discuss the assumptions made about the cached queries and data, as well as algorithms used within the Cache Manager module proposed in the query processor's overall architecture (Figure 2).

We assume that whenever a query is posed the dataset returned has a *validity period* (or equivalently an expiration timestamp). Although each sensor can be autonomous enough to decide on the validity time of its own observations, we assume that all observations returned by a query bear the same validity time. In case a query span sensors

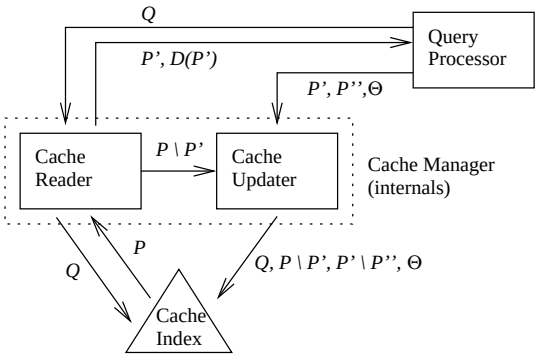


Figure 6: Detailed view of the Cache Manager's architecture

which have decided on different validity period, we assume the shortest of them to be the validity period for all the returned dataset. This conservative approach facilitates cache re-use and allows us to focus on optimizing access to cached data at the query level, instead of a single observation/single sensor level.

We envision the internal working of the Cache Manager (Figure 6) to be as follow. Given a query Q, the Cache Reader inside the Cache Manager uses an index to quickly find the set P of all relevant previous queries that intersect Q. (We make no assumption about which index structure one uses, though a classical R*-tree [2] would likely suffice for the task.) At this point each query $P_i \in P$ needs to have its validity asserted. The relevant and valid queries from P are put together in a set P'. The set of relevant but invalid queries, i..e, $P \setminus P'$, is removed from the index by the Cache Updater to avoid spurious recovery later on and also to avoid wasting index resources. Note that this cache purging can be done in parallel to the rest of the query processing. P' and the corresponding datasets $D(P')$ is then passed on to Query Processor module for further processing.

As discussed in Section 3.1, not all relevant and valid cached queries in P' will be necessarily used at query processing time, i.e., it is possible that the Query Optimizer module decides to drop elements from P'. This presents the opportunity for partially *refreshing* cached data. Consider a valid query $P'_k \in (P' \setminus P'')$ that was dropped by the Query Optimizer. By construction $P'_k \bigcap Q \neq \emptyset$, and then obtaining the data for Q (in whichever way the Query Optimizer decides to do so), implies that data for the area $P'_k \bigcap Q$ will necessarily be recovered as well. Therefore the Query Processor can take advantage of the opportunity to request the Cache Manager to partially refresh the data cached for each P'_k. For that the Cache Updater within the Cache Manager only needs Q and each of the dropped queries P'_k. Figure 7 illustrates a sample scenario, where P_1 is partially refreshed. Note that it has to be partitioned into smaller datasets in order to preserve our simplifying assumption that all the data within a previous query has the same validity time. Once this is done, all sub-queries in Θ and respective answers are also inserted into the Cache Index.

The pseudo-code for this task, comprised of basically two processes, cache reading and cache updating is described in Algorithms 4 and 5. Note that the later will require P'_k to be properly decomposed. Fortunately, this can be easily accomplished by using the algorithms discussed in Section 4.

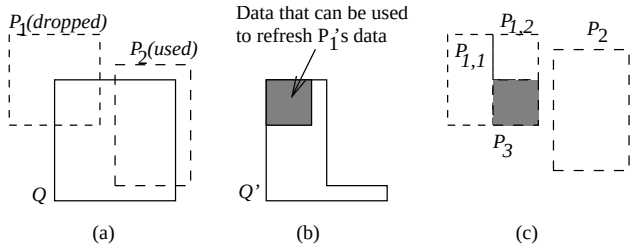


Figure 7: (a) cache status before query processing, (b) Recovered data overlapping dropped cached query P_1, and (c) status of updated cached after query processing and P_1's refreshing.

Algorithm 4 Cache reading.

Input: query Q, Cache Index (CI)
1: set $P' = \emptyset$
2: use CI to obtain $P = \{P_i$ such that $P_i' \bigcap Q \neq \emptyset\}$
3: **for** each element P_i in P **do**
4: **if** P_i is valid **then**
5: $P' = P' \bigcup P_i$
6: **else**
7: remove P_i from the cache and update CI
8: **end if**
9: **end for**
10: **return** set P'

Algorithm 5 Cache updating.

Input: sets P', P'', Θ and Cache Index (CI)
1: **for** each element θ_i in Θ **do**
2: insert θ_i into cache and update CI accordingly
3: **end for**
4: **for** each element $P_i' \in P' \setminus P''$ **do**
5: remove P_i' from the cache and update CI accordingly
6: set $R = P_i' \oplus Q$
7: compute all rectangles in $\rho(R)$, insert them into the cache and update CI accordingly
8: **end for**

Another possibility for refreshing data is to be more flexible in the filtering of expired (invalid) data. For instance one may chose to replace data which is still valid but which will expire relatively soon. The trade-off to be considered in this case is the increased query processing cost which may be offset by having data which is fresher for a longer period. A final remark regarding the Cache Manager is related its replacement policy. For the time being, we adopt a simple policy: queries closest to expiration are chosen for eviction first. Although a few ideas can be immediately suggested, a policy's usefulness is determined by the usage pattern, e.g., spatial distribution of the queries, which is an aspect outside the scope of this paper. Therefore, we defer a thorough investigation of these particular topic for future research.

6. EXPERIMENTAL RESULTS

In order to evaluate the quality of our solutions we considered the two orthogonal dimensions of efficiency and effectiveness with respect to a number of different parameters.

Efficiency is related to query processing time. Query processing time can be divided in time spent at the base station devising the best query scheme, i.e., which (sub-)queries to actually submit to the WSN, and time spent within the WSN forwarding the query and collecting the results. To simplify our analysis we assume the latter is proportional to the query cost, i.e., a higher cost means more traffic in the network which implies more complex scheduling of messages and the like. Thus, we concentrate on the time spent within the base station, which turns out to be dominated by the Query Optimizer module. Since searching for the best configuration $\langle P'', \Theta \rangle$ is the most intensive process during query optimization, we measure efficiency by the number of states explored during the search for the cost-wise optimal (or good) configuration.

Regarding effectiveness we used the estimated energy cost, as per the cost model, as the measure of interest. We investigated the solutions obtained by heuristic approaches presented in Section 4, as well as by using an exhaustive search, which served as a baseline reference. Given that the search space is relatively large and in order to make the experiments practical we stopped the exhaustive search when it reached 2^{13} states, which was typically at least one order of magnitude larger than the number of states explored by the heuristic searches. In the few cases where this maximum limit was reached the best solution found thus far was adopted as the optimal one. We compared the cost of all solutions against two straightforward choices: not using any cached data at all (i.e., submitting the original query without any further processing) or using all of the relevant cached data. As expected, only in very rare and extreme cases not using cache at all was the best option. Thus, in the interest of conciseness, we do not detail the results obtained by not using cached data. Nonetheless, recall that Algorithms 2 and 3, cannot, by their very design, yield a solution that is worse than using all or using none of the relevant cached data, respectively.

In all of our experiments we assume the sensor nodes are uniformly distributed in the monitored area, and so are the centroids of the queries. The location of the base station is fixed in the center of the monitored area. (Experiments not shown here revealed the location of base station does not have a qualitative influence in the results.) For the sake of completeness Table 2 presents the values (borrowed from [5]) which are used in the cost model for the WSN in our experiments.

Parameter	Used Values
Monitored area	1000 m×1000 m
Cost to transmit 1 bit over d meters	$50 + 10 \times d^2$ nJ
Cost to receive 1 bit	50 nJ
Wireless range radio	50 m
Query message size	32 bytes
Answer tuple size ⟨value, timestamp⟩	8 bytes

Table 2: Parameters used in WSN model.

We focus on studying the impact of the following parameters in our solutions. By varying the number of sensors (N) we influence the sensor density of the WSN, and consequently amount of data that flows when a query is processed. The larger the N the more important role of the query optimizer will become and the heavier its workload. Even though the base station can be realistic considered to be less constrained in terms of resources, one must consider

a limited amount of storage for cached data. We denote this parameter by M and investigated its effect on the simulated scenarios. The average size of the queries (S) play an important role in our scenario. Larger queries will yield a larger number of intersections with cached queries, thus more opportunity for optimization, which, however, comes at a cost (searching process). We assume that the size (area) of the queries follows an exponential distribution in order to accommodate for eventual relatively larger queries. The last parameter we investigate is the validity period of cached data (V), which reflects how dynamic and fresh the cached data is. In order to evaluate how each of those affected the algorithms' performance we adopted a *ceteris paribus* assumption, i.e., when varying one parameter all others were kept constant at their default values. All values used for the parameters above are listed in Table 3.

Parameter	Used Values
N (# of sensors × 1,000)	1, 2, **3**, 4, 5
M (# of cached queries × 100)	1, 2, **3**, 4, 5
S (query size as % of total area)	0.01, 0.25, **1**, 4, 16
V (validity period in timestamps)	10, 20, **30**, 40, 50

Table 3: Parameters investigated and respective values (bold face denotes the default values).

Before we started accumulating statistics about the algorithms performance we run the experiments for a "cold-start" period where enough queries to fill the cache were posed. Whenever a query was posed, i.e., during the cold-start of afterwards, the algorithms presented in Section 5 for maintaining cache consistency were used, and when the cache run out of space the query closest to expiration, along with its respective dataset, was evicted.

Finally, effectiveness is measured by the degree of suboptimality yielded by each approach as compared to the solution obtained by using an exhaustive search. We measure efficiency as the number of nodes explored during the optimization process. Obviously, this measure is not applicable when the full cache is used without any optimization. In the figures that follow we use the following convention to denote the origin of the data: "FC" (short for full cache) denotes results obtained by using the whole cache without any optimization, "OPT" denotes results obtained via the exhaustive search, "B+B" denotes results obtained using Algorithm 1, and "GrF" and "GrE" denote results obtained using Algorithms 2 and 3 respectively. The reported figures are average values obtained for each setting during consecutive timestamps, where 10 queries were posed in each such timestamp.

In our first set of experiments we set all parameters to their default values and compared how often the cost obtained by each of the heuristic algorithms lost to the optimal cost found by the exhaustive search and much faster it was. Thus, although it was possible that the cost reported by an heuristic algorithm was better than the "optimal" cost found by the exhaustive search, this happened only a handful of time during our experiments. Table 4 shows the statistics we obtained. The table shows B+B's unarguable superiority, as in only 7% of the cases its solution was worse than OPT's, while while still being two orders of magnitude faster and not substantially slower than the other greedy heuristics. GrF and GrE are not as effective as they lose to OPT in about 1/3 of the experiments. Looking further into this data, Figure 8 shows the histograms of the losses by all heuristics with respect to OPT. Note that the first bar of the histogram is for losses in the range of only (0-1%], i.e., practically negligible losses. Almost half of the time B+B is not able to tie with OPT, which happens only 7% of the time, it loses less than 1% in optimality. The other approaches are, as one would expect, less robust, in particular GrE—recall our previous discussion on the effect of bad choices by GrE being more pronounced than bad choices by GrF. In fact, note that in about 8% of the cases GrE lost to OPT it did so by over 100%.

Algorithm	Tied cost	Worse cost	Average speedup*
B+B	93%	7%	98.3%
GrF	73%	27%	99.8%
GrE	66%	34%	99.6%

Table 4: How well each approach performed with respect to the exhaustive search (OPT). *The exhaustive search explored on average 1,230 states.

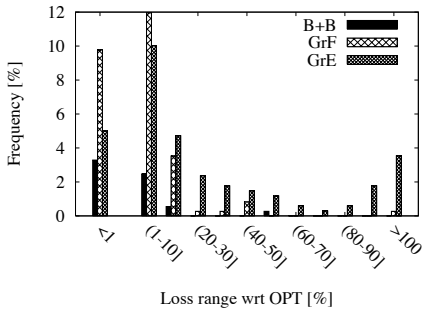


Figure 8: Distribution of all approaches' loss with respect to the exhaustive search. Note the first and last columns have different ranges in order to clarify performance at low and high ranges.

Algorithm	Better	Tied	Worse
B+B	71%	28%	1%
GrF	67%	30%	3%
GrE	70%	30%	N/A

Table 5: How well each approach performed with respect to using no cache.

Algorithm	Better	Tied	Worse
B+B	46%	54%	N/A
GrF	27%	73%	N/A
GrE	30%	46%	24%

Table 6: How well each approach performed with respect to using all of the cache.

Tables 5 and 6 show how the heuristic approaches compare with respect to the two straightforward optimization-free

alternatives to process Q: not using the cache ($P'' = \emptyset$) or using all of it ($P'' = P'$), while Figures 9 and 10 show the distribution of the gain. We make two observations from these results. On the one hand we confirm that using no cache is very rarely a worthwhile option. On the other hand, none of the heuristics was able to be as consistently superior to using all of the cache (Table 6), although B+B was strictly better almost half of the time and over 40% of the time it obtained gains upwards of 20%. This leads one to conclude that even though using $P'' = P'$ is clearly not always the best choice it remains an option that should to be considered depending on the application scenario.

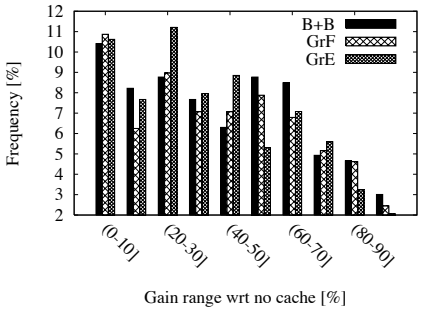


Figure 9: Histogram of all approaches' gain with respect to using $P'' = \emptyset$, i.e., no cache.

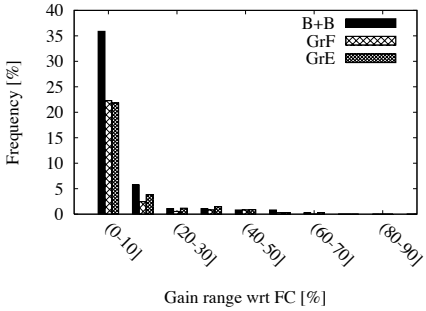


Figure 10: Histogram of all approaches' gain with respect to using $P'' = P'$, i.e., all relevant cached data.

The main conclusion at this point is that it is clear that there is merit in our claim, i.e., that it is worth performing the optimization search in order to determine good sets P'' and Θ. Next we investigate how robust each of the proposed solutions are with respect to the parameters in Table 3.

The results obtained by varying N are displayed in Figures 11 and 12. On average B+B's solution is sub-optimal by a factor smaller than 2% for all values of N, and it obtained exploring typically less than 20 states as opposed to over 1,000 ones explored by the exhaustive search. Both greedy algorithms are driven by the size of the intersections only. In scenarios with low sensor densities (lower N) this turns out to be somewhat misleading and affects their effectiveness noticeably. Nonetheless they are both faster than B+B, requiring no more than a few states to reach a local optimum.

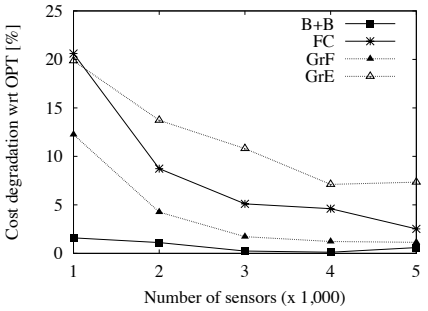


Figure 11: Query effectiveness when varying number of number of sensors in the WSN.

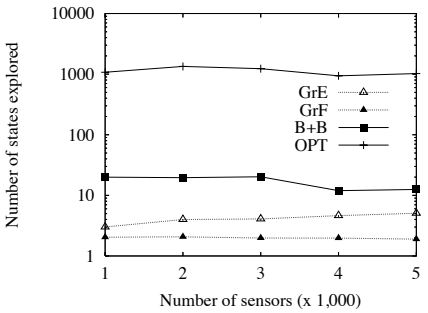


Figure 12: Query efficiency when varying number number of number of sensors in the WSN.

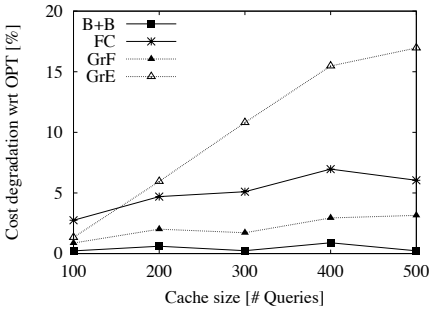


Figure 13: Query effectiveness when varying number of previous cached queries.

The results also seem to suggest that while B+B is a good compromise in general, for very dense networks (large values of N) GrF may actually be a better one (though one must consider that greedy approaches are typically less stable).

In order to simplify our analysis we measure the cache size (M) as number of queries. Note that given the values in Tables 2 and 3, it is easy to estimate the average actual size (in Bytes) of queries and their answers. When varying M we observe that while B+B is again stable and very effective, GrE tend to lose more when dealing with larger caches (Figure 13). The reason is that while a larger cache offers more opportunities for optimization it also opens the door to more bad choices due to greedy short-sightedness.

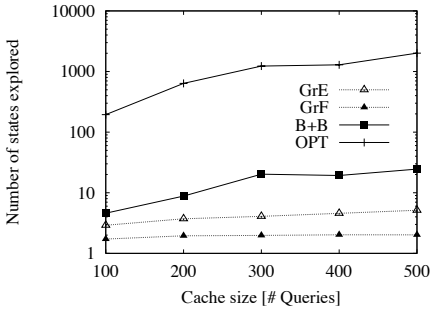


Figure 14: Query efficiency when varying number of previous cached queries.

More interestingly however, using FC becomes a less attractive alternative with the increase of M, confirming our claim that trivially using $P'' = P'$ may often not be worthwhile. Given that ideally one would like to use as much storage as possible (within reason) for the cache, our results suggest that the optimization becomes increasingly important as the cardinality of P' grows. Among the greedy approaches GrF is actually an interesting option, unlike GrE. In terms of efficiency (Figure 14), the number of states explored grows with M since a larger cache yields larger sets of candidates queries for re-use, thus more choices to be evaluated in the optimization process. As expected the greedy approaches are practically not affected in terms of efficiency.

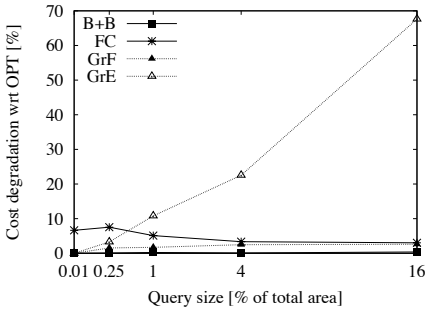


Figure 15: Query effectiveness when varying query size.

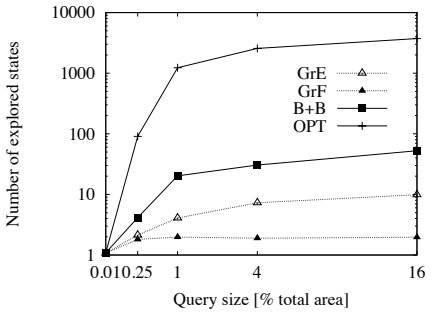


Figure 16: Query efficiency when varying query size.

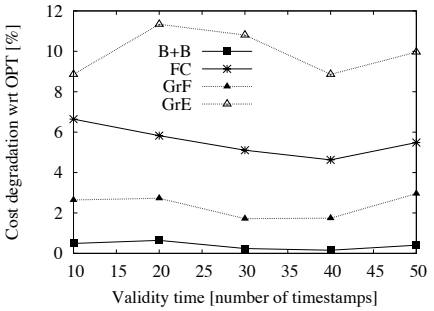


Figure 17: Query effectiveness when varying queries' validity time.

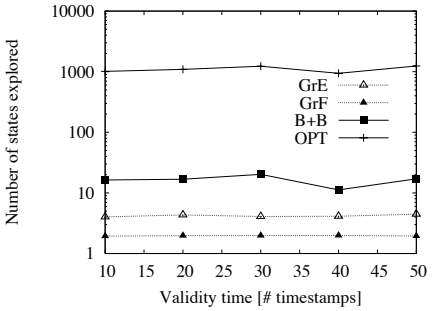


Figure 18: Query efficiency when varying queries' validity time.

Figures 15 and 16 show the effect of varying the size of the query. As before B+B offers very good effectiveness with best overall efficiency. With the increase of S it is clear that GrE's effectiveness becomes rather unacceptable, due again to poor (greedy) optimization choices. On the other hand, using FC turns out to be a not bad choice. As the size of the queries increase the optimization quickly becomes a harder problem. This is due to a larger number of intersections and a large number of configurations to be considered, and this is clearly reflected in Figure 16. One aspect that is not transparent in our experiments is that when queries become very large at some point the monitored area will be eventually fully covered by cached queries (unless they have a very short validity period). Hence, a rather trivial solution to the optimization problem is to use all of the cache the area of the new query will be fully covered by cached data.

Our last experiment was varying the validity period (V). Figures 17 and 18 do not show a very clear dependence between this parameter and overall performance. This is not unexpected, since while having long lived queries can potentially allow for better optimization, the number of cached queries is limited (by M), i.e., queries still valid will eventually need to be evicted, forcefully limiting the search space.

In summary, it is clear that the branch-and-bound optimization process is able to almost always offer identical or very close to optimal query plans at the low cost of exploring only a very small number of configurations $\langle P'', \Theta \rangle$. While on the one hand one cannot deny that using all the relevant valid cached data is often a reasonable compromise, in

a domain, such as WSNs, the chance of saving every bit of energy, as opportunistic as it may be, must be taken. Even relatively small savings over the long term are worth the optimization overhead (which incidentally has no impact in the energy budget of the network as it is performed in the base station).

7. CONCLUSIONS

In this paper we have investigated the problem of how to effectively exploit a data cache at the base station of a WSN. The formalization of the problem calls for answers to two sub-problems: (1) how to select which queries to use, and depending on those, (2) how to create the sub-queries that will be submitted to the WSN. Given the highly combinatorial nature of the problem we proposed a few heuristic approaches. The best alternative, which is based on a branch-and-bound optimization search is very efficient (typically two orders of magnitude faster than an exhaustive search) and also effective (typically less than 2% and no more than 10% sub-optimal). Finally, even though this work reused the query processing framework presented in [5], the solution we propose does not depend on the same. Rather, any approach can be used as long as it provides the cost model that is used to guide the search for an optimized query "plan". We are currently working on extending the ideas in this paper to address other types of queries, e.g., aggregate queries.

Acknowledgements

Research partially supported by NSERC (Canada), CBIE (Canada), CAPES (Brazil) and FUNCAP (Brazil).

8. REFERENCES

[1] D. Abadi, S. Madden, and W. Lindner. REED: Robust, efficient filtering and event detection in sensor networks. In *Proc. of VLDB*, pages 769–780, 2005.

[2] N. Beckmann et al. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of SIGMOD*, pages 322–331, 1990.

[3] A. Brayner et al. Toward adaptive query processing in wireless sensor networks. *Signal Processing*, 87(12):2911–2933, 2007.

[4] R. Brooks, P. Ramanathan, and A. Sayeed. Distributed target classification and tracking in sensor networks. In *Proc. of the IEEE*, pages 1163–1171, 2003.

[5] A. Coman, J. Sander, and M. Nascimento. Adaptive processing of historical spatial range queries in peer-to-peer sensor networks. *Distributed and Parallel Databases*, 22(2-3):133–163, 2007.

[6] J. Culberson and R. Reckhow. Covering polygons is hard. *J. Algorithms*, 17(1):2–44, 1994.

[7] S. Dar et al. Semantic data caching and replacement. In *Proc. of VLDB*, pages 330–341, 1996.

[8] A. Deshpande et al. Cache-and-query for wide area sensor databases. In *Proc. of SIGMOD*, pages 503–514, 2003.

[9] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[10] J. Gudmundsson, T. Husfeldt, and C. Levcopoulos. Lower bounds for approximate polygon decomposition and minimum gap. *Inf. Process. Lett.*, 81(3):137–141, 2002.

[11] C. Intanagonwiwat et al. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[12] D. Kossmann, M. Franklin, and G. Drasch. Cache investment: integrating query optimization and distributed data placement. *ACM Trans. Database Syst.*, 25(4):517–558, 2000.

[13] V. Kumar and H. Ramesh. Covering rectilinear polygons with axis-parallel rectangles. In *Proc. of STOC*, pages 445–454, 1999.

[14] S. Madden et al. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proc. of OSDI*, 2002.

[15] A. Perrig, J. Stankovic, and D. Wagner. Security in wireless sensor networks. *Commun. ACM*, 47(6):53–57, 2004.

[16] S. Prabh and T. Abdelzaher. Energy-conserving data cache placement in sensor networks. *ACM Trans. on Sensor Networks*, 1(2):178–203, 2005.

[17] F. Preparata and M. Shamos. *Computational Geometry - An Introduction*. Springer, 1985.

[18] M. Rahman and S. Hussain. Effective caching in wireless sensor network. In *Proc. of AINA Workshops (1)*, pages 43–47, 2007.

[19] A. Silberstein, R. Braynard, and J. Yang. Constraint chaining: on energy-efficient continuous monitoring in sensor networks. In *Proc. of SIGMOD*, pages 157–168, 2006.

[20] A. Silberstein, K. Munagala, and J. Yang. Energy-efficient monitoring of extreme values in sensor networks. In *Proc. of SIGMOD*, pages 169–180, 2006.

[21] V. Soltan and A. Gorpinevich. Minimum dissection of a rectilinear polygon with arbitrary holes into rectangles. *Discrete & Computational Geometry*, 9:57–79, 1993.

[22] R. Szewczyk et al. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, 2004.

[23] I.-L. Tseng and A. Postula. An efficient algorithm for partitioning parameterized polygons into rectangles. In *Proc. of GLSVLSI*, pages 366–371, 2006.

[24] B. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, 1992.

[25] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proc. of CIDR*, 2003.