# Two-Tier Multiple Query Optimization for Sensor Networks

Shili Xiang Hock Beng Lim Kian-Lee Tan Yongluan Zhou
Department of Computer Science, National University of Singapore
{xiangshi, limhb, tankl,zhouyong}@comp.nus.edu.sg

## Abstract

*When there are multiple queries posed to the resource-constrained wireless sensor network, it is critical to process them efficiently. In this paper, we propose a Two-Tier Multiple Query Optimization (TTMQO) scheme. The first tier, called base station optimization, adopts a cost-based approach to rewrite a set of queries into an optimized set that shares the commonality and eliminates the redundancy among the queries in the original set. The optimized queries are then injected into the wireless sensor network. In the second tier, called in-network optimization, our scheme efficiently delivers query results by taking advantage of the broadcast nature of the radio channel and sharing the sensor readings among similar queries over time and space at a finer granularity. Our experimental results indicate that our proposed TTMQO scheme offers significant improvements over the traditional single query optimization technique.*

## 1. Introduction

Wireless sensor networks are increasingly being deployed in many important applications to enable users to query the physical world, such as environmental monitoring, healthcare monitoring, military surveillance, traffic monitoring, etc. To ease the deployment of such applications, researchers have proposed techniques to treat the wireless sensor network as a database which provides a good logical abstraction for sensor data management, and hence better realizes the potential of wireless sensor networks [16]. Users can issue declarative queries without having to worry about how the data are generated, processed, and transferred within the network, and how sensor nodes are (re)programmed. Query optimization techniques can also be applied to optimize the network operations.

In many applications, it is often necessary to process multiple user queries simultaneously. Unfortunately, most existing work has focused on the optimization and execution of a single long-running query. Consequently, multiple concurrent queries cannot benefit from each other by sharing their data acquisition, computation and communication cost. Moreover, running multiple queries in such an uncooperative manner will lead to bandwidth contention and even data loss as a result of transmission collisions (which may in turn require retransmission). Thus, in the resource constrained sensor network, it is critical to perform multi-query optimization in order to share the limited communication and computational resources.

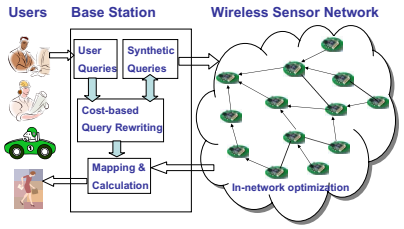



**Figure 1. The System Architecture**

In this paper, we propose a Two-Tier Multiple Query Optimization (TTMQO) scheme to minimize the average transmission time in the sensor network. Figure 1 shows the system architecture. TTMQO supports both aggregation and data acquisition queries. The first-tier optimization is performed at the base station. We adopt a cost-based approach to heuristically rewrite user queries into "synthetic" queries before injecting them into the sensor network, such that duplicate data requests from original queries can be eliminated as much as possible while guaranteeing the correctness of semantics of all queries. After the sensor network returns results for the synthetic queries, corresponding results for user queries can be easily obtained through mapping and calculation. Our algorithm is dynamic in that the set of running synthetic queries is continuously being updated by the arrival of new queries as well as the termination of existing queries.

Our second-tier optimization is done inside the wireless sensor network. The main idea is to focus on the data required by all (synthetic) queries during specific time interval, and design a good DAG over the sensor nodes with the base station as the sink point to gather the queried data. Our algorithm further reduces the number of radio messages and

27th International Conference on Distributed Computing Systems (ICDCS'07)
0-7695-2837-3/07 $20.00 © 2007 IEEE

saves the energy of sensor nodes in three ways. First, it schedules the communication among queries as a whole, which enables the combination of several query transmissions if these queries need data at the same time. Second, our algorithm dynamically determines the route to disseminate the query results, which enables data aggregation as soon as possible and involves fewer nodes. Finally, it tries to acquire and transmit the data to satisfy multiple queries if they need the same data, by taking advantage of the broadcast nature of the radio channel.

To study the effectiveness of the TTMQO scheme, we implemented it using TinyDB [6] and evaluated it under the TOSSIM [5] emulator. Our experimental results show that the TTMQO scheme can provide significant performance improvements, in terms of the cost of radio transmission and scalability with the number of queries.

The rest of this paper is organized as follows. In Section 2, we review the related work and background information on data query optimization in sensor networks. Section 3 presents our TTMQO scheme in detail. In section 4, we discuss the methodology and results of our experimental study. Finally, we conclude the paper in Section 5.

## 2. Background and related work

Several sensor database query systems, such as Cougar [16], and TinyDB [6], have been developed by the database research community. These works have established the foundation of sensory data management which our work is based on. In this paper, we consider queries with the semantics discussed in TinyDB, which consists of a SELECT-FROM-WHERE clause supporting selection, projection, and aggregation, in addition to EPOCH DURATION clause to define the frequency that the sensor data should be fetched from the network [6].

Besides these systems, a large amount of research has been conducted on various aspects of sensor query processing techniques. These include the design of energy-efficient routing protocols [10], in-network query processing techniques [1, 12], approximate data query processing [3], and adaptive techniques to adjust query strategies and optimize query plans over time [2]. These research have mainly focused on the optimization and execution of a single long-running query, which is complementary to our base station optimization.

The *multi-query optimization* (MQO) problem has long been studied by the database community and many heuristic techniques are proposed [9, 8]. However, these techniques are not directly applicable to wireless sensor networks, even at the base station, due to the following two reasons. First, sensor queries have different query semantics, with one more "dimension" specified by the EPOCH DURATION clause. Second, with data being ready to be processed, traditional MQO focuses on optimizing the physical level, but the logical level optimization is critical in our case.

Multi-query optimization has also been considered in the context of streaming database [7, 4], where the focus is to develop techniques to process multiple queries over the collected sensor data streams. The problem of deciding which set of data to be collected at the required frequency from a sensor network is not addressed.

There are only a few studies on multi-query optimization in wireless sensor networks. Trigoni et al. tackled multi-query optimization in wireless sensor networks in [13]. Their work studies region based aggregation queries, while our scheme aims to support more types of queries, including, for example, other types of aggregation queries (e.g., value-based aggregation queries) and data acquisition queries. Most recently, Silberstein et al. exploited the combination of multicast and in-network optimization to optimize many-to-many aggregations to achieve efficient in-network control of sensors [11], where the application context is different from ours.

## 3. Two-tier multiple query optimization

Since sensor nodes are resource-constrained, we endeavor to design a light-weight but effective scheme to support multiple queries running inside a wireless sensor network. The base station is the interface of a wireless sensor network. Moreover, it is usually much more powerful than sensor nodes, with abundant processing, disk, and memory capacity. Thus, we use the base station as a filter to reduce duplicate data access to the sensor network and as a screen to hide the query dynamics from the sensor network as much as possible. The objective here is to save the energy at sensor side instead of minimizing the response time or computation cost at the base station.

### 3.1. Base station optimization algorithm

As our base station optimization algorithm is based on query rewriting, we propose a cost model to measure the benefit of the rewriting. Then we propose a heuristic query insertion algorithm, which is guided by the cost model, to optimize the synthetic query set for each new incoming query. Finally, we look at how to re-optimize the synthetic queries when a query terminates.

#### 3.1.1 Basic data structures

Let us first introduce some basic data structures of our *user queries* and *synthetic queries*. We store each user query in the form of $\langle qid, attribute\_list|agg\_list, predicates, epoch\_duration, qid' \rangle$ in a query table. $qid$ is the unique

identifier of the query. The $attribute\_list$ field contains the list of attributes that a data acquisition query $qid$ retrieves from the wireless sensor network. $agg\_list$ is a list of $\langle operator, attribute\rangle$ that an aggregation query $qid$ acquires. We note that for a single query, either $attribute\_list$ or $agg\_list$ will be empty. $predicate\_list$ is a list of $\langle attribute, min, max\rangle$. The $qid'$ field is used by our algorithm to denote which *synthetic* query this query $qid$ has been rewritten into.

As for a synthetic query, besides the above fields, a few more fields are used. (a) A $count$ field is associated with the $epoch\_duration$ field as well as each entry in the various lists ($attribute\_list$, $agg\_list$ and $predicate\_list$), which denotes the number of user queries that require that piece of data. This is to facilitate the maintenance of the synthetic query when user queries terminate. (b) A $from\_list$ field contains the user queries which the synthetic query is responsible for. (c) A $flag$ field denotes the current status of this synthetic query. (d) A $benefit$ field indicates the benefit that can be gained by the synthetic query (in comparison to processing the individual user queries). It is worthy to note that all these enhanced fields of the *synthetic* query are stored in the base station to help with query rewriting and further mapping and calculation, and they are not contained in the query propagation message.

### 3.1.2 Benefit estimation

**Cost model.** Moore's law suggests that the memory density and processor speed will continue to grow at an exponential rate. Thus, we expect sensor networks to continue to be bandwidth and energy limited. Since radio transmission is the most energy intensive operation a node performs, we use the cost of radio transmission as our performance metric.

Radio messages consist of query result transmission messages, query propagation and abortion messages, and periodical network maintenance messages. For continuous queries, result transmission messages dominate, so we only count the result message transmission in our cost model. However, to be realistic, we also include the effect of other radio message transmission into the cost of radio transmission in the experimental study.

For a query $q_i$, assume the length of its result message is $len(q_i)$. The transmission cost of a result message from one node to another can be estimated as $C_{start}+C_{trans}\cdot len(q_i)$, where $C_{start}$ is the transmission startup cost and $C_{trans}$ is the transmission cost of each unit of data. To measure the average transmission cost incurred by $q_i$ for each unit of time, we have to estimate the number of per-unit time transmissions incurred by $q_i$, which is related to the number of result messages generated by the sensors as well as the number of hops required to forward the messages back to the base station.

First, we look at the per-unit time number of result messages generated by a set of sensor nodes $N_k$, which is denoted as $result(q_i, N_k)$. At the end of each epoch of $q_i$, one result message would be generated by a sensor node whose readings satisfy the predicates of $q_i$. Therefore, we have

$$result(q_i, N_k) = \frac{sel(q_i, N_k) \cdot |N_k|}{epoch_i} \quad (1)$$

where $sel(q_i, N_k)$ is the selectivity of the query predicates over $N_k$, which is equal to the percentage of sensor nodes in $N_k$ whose readings can satisfy the query predicates, $epoch_i$ is the epoch length of $q_i$.

Second, the forwarding hops of the result messages are determined by the message source nodes' location at the data routing tree. Based on Eq. (1), the number of message transmission incurred by $q_i$ can be estimated as

$$trans(q_i) = \sum_{k=1}^{max\_depth} result(q_i, N_k) \cdot k \quad (2)$$

where $N_k$ is the set of sensor nodes at the $k$th level of the routing tree and $max\_depth$ is the maximum depth of the routing tree. Note that messages may be retransmitted due to transmission failures, such as collisions. Here we assume the number of retransmissions is proportional to $trans(q_i)$ and can be omitted in our cost model because only relative value is necessary to guide our query rewriting. Again, retransmission messages are considered in our experimental study.

Eq. (2) provides an accurate estimation for acquisition queries where no in-network aggregation occurs. For aggregation queries, an internal node at the data routing path can forward aggregation values instead of the original detail values to reduce the number of message transmissions. Hence the actual number of transmissions would be a value within the range of $[result(q_i, N), trans(q_i)]$, where $N$ is the whole set of sensors in the network. The lower bound value happens if each node that receives a result message also generates a result itself and can aggregate the received result with its own result, while the upper bound value occurs when no in-network aggregation can be performed at all. Unfortunately, there is no straightforward way to estimate this actual value. That is because the places where in-network aggregation occurs is hard to predict unless we make much stronger assumptions, which is undesirable. In this paper, we just use the lower bound value. As we will see soon, this is conservative in that an aggregation query is integrated with an acquisition query only if it is guaranteed to be beneficial.

Now we can compute the cost of a query $cost(q_i)$ as

$$cost(q_i) = trans(q_i) \cdot (C_{start} + C_{trans} \cdot len(q_i)) \quad (3)$$

**Benefit estimation.** If we integrate two queries $q_1$ and $q_2$ into one synthetic query $q_{12}$, to ensure correctness, all the data requested by $q_1$ and $q_2$ must be requested by $q_{12}$. In other words, the data requested by $q_{12}$ is a superset of

the data requested by $q_1$ and $q_2$. Semantic correctness constraints must be considered as well.

If $q_1$ and $q_2$ are aggregation queries (and hence $q_{12}$). In order to derive results for both $q_1$ and $q_2$ from the result of $q_{12}$, the two queries must have the same predicates. Hence, the integration of two aggregation queries in this way is guaranteed to be beneficial and hence we do not need to estimate their benefit.

For other integrations, we have to estimate their benefits. After integration, the requested attributes and predicates of $q_{12}$ will be the union of those of $q_1$ and $q_2$, while the epoch duration should be the Greatest Common Divisor of $epoch_1$ and $epoch_2$. We can estimate the cost of $q_{12}$ by using Eq. (3). The benefit of the integration is estimated as $benefit(q_1, q_2) = cost(q_1) + cost(q_2) - cost(q_{12})$.

**Statistics.** To compute our cost function, we have to maintain some statistics. We use the reciprocal of the data rate of the sensor nodes (given by the sensor specifications) as the value of $C_{trans}$, while we periodically measure the actual average transmission startup time and use it as $C_{start}$. Another value to be estimated is $sel(q_i, N_k)$. To do so, at each level of the routing tree, we can maintain the data distribution, which is an independent problem studied in other literatures, such as [3]. In practice, to save maintenance cost, we can maintain one data distribution for multiple levels and assume the data distributions among these levels are identical. Since our focus is on multiple query optimization, in our experiments, we only use one distribution for all the levels, which actually biases against our techniques.

### 3.1.3 Greedy query insertion algorithm

Given a new query $q_i$ that arrives at the base station and a list of currently running synthetic queries $Q_{syn}$, our greedy query insertion algorithm (shown in Algorithm 1) works as follows. If there is no synthetic query available, we directly add $q_i$ in the synthetic query list. Otherwise, it searches for the most beneficial synthetic query $q_{id}$ to rewrite with this $q_i$ to produce a new synthetic query (lines 5–10). If $q_{id}$ covers $q_i$ (line 11), the newly added user query $q_i$ will not have any effect on the workload in the sensor network. Otherwise, *Integrate($q_{id}$, $q_i$)* is called to update the most beneficial query $q_{id}$ into a new synthetic query. To identify that $q_i$ is covered by $q_{id}$, as shown in line 6, we design the $Beneficial(q_i, q_j)$ function to return the benefit rate instead of the original $benefit(q_i, q_j)$ defined in Section 3.1.2. More specifically, we divide the computed $benefit(q_i, q_j)$ by $cost(q_i)$. If there is no synthetic query that can be rewritten with the query $q_i$ so that there are benefits, $q_i$ is added into the synthetic query list. Upon the termination of the algorithm, if the synthetic query list is changed, corresponding query abortion and injection operations will be invoked to complete the whole process.

**Algorithm 1**: $Insert(q_i, Q_{syn})$

```
1  if Q_syn == NULL then
2      Q_syn.add(sqid); q_i.qid' ← sqid; UpdateCount(q_i, sqid, 1);
3  else
4      q_j ← Q_syn.next; max ← 0; id ← 0;
5      while q_j != NULL do
6          BenefitRate ← Beneficial(q_i, q_j);
7          if BenefitRate > max then
8              max ← BenefitRate; id ← j;
9          if max == 1 then break;
10         q_j ← q_j.next;
11     if max == 1 then
12         q_i.qid' ← q_id; UpdateCount(q_i, q_id, 1);
13     else if max > 0 then
14         Integrate(q_id, q_i); Insert(q_id, Q_syn);
15     else
16         Q_syn.add(sqid); q_i.qid' ← sqid; UpdateCount(q_i,
           sqid, 1);
```

It is possible that synthetical queries can further benefit from the newly integrated synthetic query. Below shows a simple example to illustrate the situation:

$q_1$:select light where 280<light<600 epoch duration 2

$q_2$:select light where 100<light<300 epoch duration 4

$q_3$:select light where 150<light<500 epoch duration 4

For simplicity we assume all the sensor readings are uniform distribution and the value of $(C_{start}+C_{trans}*len(q_i))$ for any $q_i$ is equal to 1. Then, $benefit(q_1, q_2) = d * (\frac{sel(p1)}{epoch_1} + \frac{sel(p2)}{epoch_2} - \frac{sel(p1 \cup p2)}{GCD(epoch_1, epoch_2)}) = \frac{d}{L} * (\frac{320}{2} + \frac{200}{4} - \frac{500}{2}) < 0$, where L is the value range of light attribute and $d$ is the average depth of a node in the routing tree (i.e. $d = \sum_k N_k \cdot k/|N|$). Under this situation, $q_1$ and $q_2$ will not be integrated, and both of them are directly added into the synthetic query list as $q_1'$ and $q_2'$.

When $q_3$ is admitted, $benefit(q_1', q_3) = \frac{d}{L} * (\frac{320}{2} + \frac{350}{4} - \frac{350}{2}) < 0$, no integration with $q_1$'. But $benefit(q_2', q_3) = \frac{d}{L} * (\frac{200}{4} + \frac{350}{4} - \frac{400}{4}) > 0$, so we integrate $q_3$ with $q_2'$:

$q_2''$:select light where 100<light<500 epoch duration 4

If we evaluate $q_2''$ against synthetic query $q_1'$, $benefit(q_1', q_2'') = \frac{d}{L} * (\frac{320}{2} + \frac{400}{4} - \frac{500}{2}) > 0$, so $q_1'$ can benefit from new $q_2''$. The resulting query is :

$q_1''$: select light where 100<light<600 epoch duration 2

Hence, we need a more aggressive solution to remove the redundant data requests among user queries. To achieve this, after $Integrate(q_{id}, q_i)$ in line 14 in Algorithm 1 has updated the synthetic query $q_{id}$ into a new one, we iteratively exploit further benefit by rewriting $q_{id}$ with the current running synthetic querylist by calling $Insert(q_{id}, Q_{syn})$.

The *Beneficial* function first identifies whether two queries are rewritable based on semantic correctness con-

straints, and then computes the benefit rate. The *Integrate* function modifies the synthetic query $q_{id}$ so that all the data requested by $q_i$ will be requested by the new $q_{id}$; it is also responsible for changing the values of the enhanced fields of the synthetic queries shown in section 3.1.1. The modification of the count fields upon insertion and termination is accomplished by an $UpdateCount$ procedure, with a flag to differentiate increment or decrement.

#### 3.1.4 Adaptive query termination algorithm

To handle dynamic workloads where user queries may join or leave dynamically, we introduce a parameter $\alpha$ to adjust our query termination algorithm according to the property of application workload.

**Algorithm 2**: $Terminate(q, Q_{syn})$

```
1 Find the synquery sq_old that q has been written into;
2 UpdateCount(q, sq_old, 0);
3 Remove q.qid from sq_old.fromlist;
4 if some count in sq_old has decreased to 0 then
5     if vol(q) > sq_old.benefit * α then
6         for All query q_i in sq_old.fromlist do
7             Insert(q_i, Q_syn);
```

As shown in Algorithm 2, when a query $q$ is terminated by a user, the synthetic query it was written into, denoted as $sq_{old}$, can be easily determined, based on the information kept at $q.qid'$. Query $q$ eliminates its contribution in the synthetic query $sq_{old}$ by *UpdateCount*. If the count of some field has been decreased to 0, it means that this query is the only query that requires $sq_{old}$ to request some specific data. The termination of this query may trigger the reconstruction of the synthetic queries.

We hide the effect of termination of query $q$ from the sensor network by keeping the old synthetic query $sq_{old}$ unchanged, if the following condition is satisfied:

$$\frac{|sq_{old}.benefit - sq_{old}.benefit'|}{sq_{old}.benefit} \leq \alpha$$

where $sq_{old}.benefit'$ is the new $benefit$ value of $sq_{old}$ after the removal of $q$. Since $cost(q)$ is equal to $sq_{old}.benefit - sq_{old}.benefit'$ according to Section 3.1.2, the condition can also be represented as: $cost(q) <= sq_{old}.benefit * \alpha$ (line 5). If such condition is not satisfied, we re-insert the remaining user queries contained in $sq_{old}$ in the same way as the newly arrival queries (lines 6-7). $\alpha$ is a system parameter to tune the aggressiveness of query rewriting upon query termination. A good $\alpha$ value can avoid frequent query abortion and injection to the sensor network, which are also costly operations.

Moreover, when there are considerable similarity between queries, it is very likely that the query insertion and termination can be handled at the base station, without affecting the sensor network.

### 3.2. In-network Optimization Algorithm

We note that the base station optimization is able to exploit the similarity among queries and eliminate the redundancy among queries through greedy query rewriting. However, the base station optimization does not support sharing of the commonality among queries at the finest granularity. Since every query from the base station has the same meaning for each sensor node all the time, the base station optimization is a "all-or-nothing" approach. Moreover, base station optimization cannot take advantage of the special properties of sensor nodes, such as the broadcast nature of sensor radio transmission. Hence, we have our second-tier optimization inside the wireless sensor network, called in-network optimization, where sensors make local decisions by themselves and behave adaptively to the query workload with time.

#### 3.2.1 Sharing Over Time

Consider two queries $q_1$ and $q_2$, whose only difference is their epoch durations. If the epoch duration of one query can be divided by that of the other (such as 2048ms and 4096ms), these two queries can be integrated into one according to the base station algorithm in Section 3.1 and thus the common result transmissions are shared. Otherwise (such as 4096ms and 6144ms), these two queries are sent into the network as two independent queries because we are not be able to construct a beneficial synthetic query. However, in this case, half of the data requested by $q_2$ are also requested by $q_1$, which can be saved if we can schedule these two queries properly.

Based on the above observation, we exploit more sharing by scheduling the data acquisition and transmission of all queries in a whole. After a new query is propagated to the network, we (re)set the node's clock to fire at the GCD (Greatest Common Divisor) of the epoch durations of all the queries. The epoch start time for the new query on a sensor node is set to be divisible by the epoch duration (the smallest allowed epoch duration is 2048ms, and we assume that every epoch duration is divisible by it). In this way, the latency of the first epoch may be longer; however, for a continuous query, this extra latency for the first epoch is acceptable. On the other hand, by introducing such a little delay, various queries that have the same epoch duration will start sampling at the same time in every epoch, and hence can share sample acquisition. More specifically, when the clock is fired at time $t$, if there exists any $q_i$ such that $t \bmod q_i.epoch = 0$, a shared data acquisition is conducted for all such $q_i$s.

### 3.2.2 Sharing Over Space

After the sample rate has been set at each node, data will be retrieved periodically and transmitted out of the network to the base station. During the query result collection, we use the following optimization heuristics to aggressively share data over space. Each sensor node dynamically selects a route (parent) that is aware of the query space; in the meanwhile, it tries to take advantage of the broadcast nature of the radio channel to satisfy multiple queries in one message.

In TinyDB, a parent node is associated with each node based on the link quality, and hence a fixed routing tree is constructed, which is ignorant of the query space. In our scheme, we focus on the data that are required by queries during specific time interval. We let the source sensor node multicast/unicast the data along a DAG with the base station as the sink point, and dynamically form the routing trees for various queries at the same time. The scheme works as follows:

**Query Propagation Phase.** Queries are flooded throughout the network from the base station. For a value-based query, flooding is necessary, because the accurate set of sensors that have data for the query are not known a prior to the base station and the set of sensor nodes can vary with time as well. If the query is a region-based query or a node-id based query, the set of answer nodes are known in advance, and more efficient techniques such as SRT [6] can be used. Here, we let every sensor decide where to propagate to based on its local information about neighbors.

When the query is propagated from node $x$ at level $i$ to level $i+1$, node $x$ checks whether it has the data the query retrieves, and piggybacks this information down. In the meanwhile, the DAG is formed by having an edge from every node to each of its upper level neighbors (If the network is too dense and not all neighbors can be maintained, preference is given to the neighbors that also have query result to transmit). If the data at node $x$ does not satisfy any query, $x$ switches into sleep mode and will wake up after a predefined time. When it wakes up, if it finds that its current data satisfies a query, it sends a one-hop broadcast message so that its lower level neighbors would consider the node as an option to relay its data.

**Result Collection Phase.** When the data of a sensor node satisfy the predicates of any query that is triggered at the current time, the node will pack the data and select routes to forward them. Data acquisition queries and aggregation queries are processed independently, and hence the way they can share their common data in the network is different. For data aggregation queries, in-network aggregation at internal nodes is applied and each aggregation operator (such as MAX) is processed with a result message. Thus, one data message can be packed to share among all of the queries whose partial aggregation value are the same. For data acquisition queries, the sensor node generates a result message that contains the requesting attributes of all the queries whose predicates are satisfied. In this way, the message transmission can be shared among multiple queries, and would be further forwarded all the way along until the base station. Note that the length of a shared message may be larger, but it is cheaper to transmit one shared message than multiple query result messages.

After the result messages are generated, each sensor node dynamically chooses a parent for each message based on local information. To intelligently select a route to transmit data, each node keeps a list of its neighbors as what is done in TinyDB, but we also maintains the information about whether its upper level neighbor has data for each query, which was achieved through piggyback mentioned above. When a node $x$ at level $k$ has result messages for one or more queries, it checks whether there is a neighbor at level $k-1$ that also has data for these queries. Neighbors with data for more queries have higher priority to be chosen. Ties are broken by favoring those nodes with more stable link with $x$. In this case, unicast message is sent to the chosen neighbor to further forward or aggregate. Otherwise, if multiple neighbors are chosen (each is responsible for forwarding message for a subset of queries), one multicast message is required to send out the message to all these neighbors.

When an upper level node $y$ receives a multicast message and it is one of the destinations of the multicast message, from the packet header, it identifies the set of queries that the message is for. It may perform necessary processing on the message (e.g. aggregate with its own data for aggregation query) and choose an upper level neighbor to forward the message. This procedure repeats until the message reaches the base station.

**Discussion.** In real applications, sensor readings are often spatially and temporally correlated, and hence the set of sensor nodes involved in a query are likely to be spatially connected and temporally stable. When a node has a result message for a set of queries, it is very likely that one of its neighbors would also have one for those queries. Under the dynamic route selection strategy together with multicast, such result transmission cost can be shared among queries. This is especially beneficial for aggregate queries, whose common partial aggregation will continue to be aggregated with other partial data at the upper level nodes to further reduce the radio transmission.

From the above, we can see that much data transmission and energy can be saved by enabling sensor nodes to make intelligent local decisions: a sensor node only needs to transmit its data once to answer all the data acquisition queries; in-network aggregation is conducted sooner for data aggregation queries; the nodes that have no data to transmit can operate in a sleep mode to save energy.

In Figure 2, we illustrate the algorithm by a simple ex-

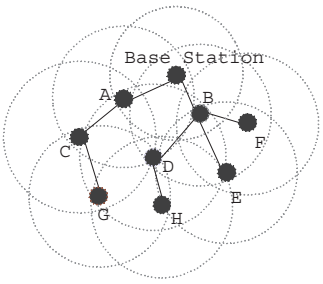



**Figure 2. Routing tree in TinyDB**

ample. The solid lines denote the routing tree in TinyDB, and dotted lines represent the radio range of a node. Suppose D, E, F, G, H are queried by data acquisition query $q_i$, and D, G, H are queried by data acquisition query $q_j$, and both queries need data at time $t$. Using TinyDB, to answer $q_i$, all nodes will be involved. To answer $q_j$, nodes D, G, H will conduct sample acquisition again and intermediate nodes will relay their data twice. Hence, in total, 8 sensor nodes are involved, and 12+8=20 radio messages are transmitted. Using our DAG, G will choose D instead of C to relay for both $q_i$ and $q_j$, and hence node C and A can be instructed to sleep. The data message from node D, G and H can be transmitted only once to answer both of queries. Thus, a total of 6 sensor nodes are involved and 4+8=12 radio messages are transmitted.

For data aggregation queries, even more messages and energy can be saved. By dynamically choosing node D as the parent of node G, the aggregation for data at node G that is supposed to be done at base station is done sooner at D. Moreover, the aggregation from nodes G, H and D can be shared at D among $q_i$ and $q_j$. Thus, even node B still needs to send one aggregated message representing $q_i$ and $q_j$ respectively due to the further aggregation of data at E and F for $q_i$ at B, only 7 out of 14 messages will be transmitted in total.

## 4. Experimental evaluation

In this section, we shall present representative experimental results to show the performance of our scheme. More details can be found in [14].

### 4.1. Methodology

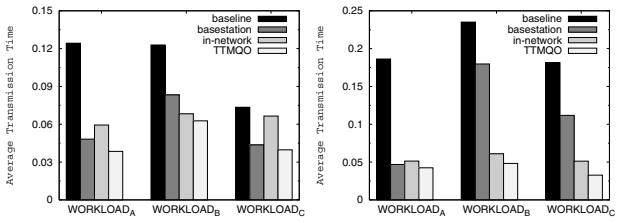
We have implemented our TTMQO scheme on top of TinyDB, the most popular query processing system for sensor networks. In our experiments, we used the packet-level TOSSIM [5], an emulator for TinyOS-based sensor networks.

We assume that the sensor nodes are deployed uniformly in a $n \times n$ two-dimensional grid, with the base station node 0 at the upper left corner. The radio transmission radius is set to be 50 feet, while the grid spacing is 20 feet. In this work, we assume a lossless communication environment in which each node could transmit data to sensor nodes that are within its radio range. As a reference, we use the following strategy as the baseline for comparison: each query is optimized by TinyDB, and multiple queries that have been sent to the base station are all injected into the network to run concurrently without multi-query optimization.

The cost of radio transmission is our performance metric to minimize energy and bandwidth in the sensor network. The cost function there actually tries to measure the transmission time of the result messages. To be realistic, we count in the transmission time of all radio messages, which comprise result transmission messages, query propagation and abortion messages, network maintenance messages and retransmission messages due to transmission failure. More specifically, we report the *average transmission time* in our figures, which measures the average percentage of transmission time spent on each node for all running queries over the simulation time.

### 4.2. Impact of optimization tiers

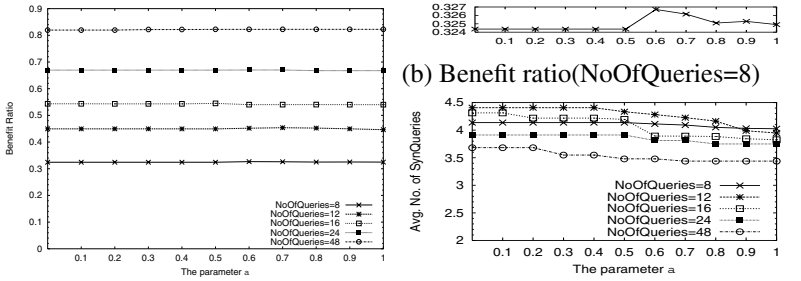
In this section, we study the performance gain we can achieve with each optimization tier. We construct three static workloads. The $WORKLOAD_A$ is designed to focus on the (common) savings that can be achieved by both the base station optimization and in-network optimization; the $WORKLOAD_B$ is used to show the complementary of in-network optimization to base station optimization; the $WORKLOAD_C$ is designed to test the mutual complementary of these two optimizations.




(a) Number of Nodes=16 (b) Number of Nodes=64

**Figure 3. Average Transmission Time**

From the results in Figure 3, we can see that our optimization algorithms behave as what we have expected. For $WORKLOAD_A$, the base station optimization and in-network optimization algorithm both eliminated the redundant data requests for similar queries, though in different ways. Compared with base station optimization, in-network optimization can more progressively share data requested over time and space, but it cannot enable aggregation queries to benefit from data acquisition queries in addition to its larger message size to support multiple queries.




(a) Benefit ratio

(b) Benefit ratio(NoOfQueries=8)

(c) The average number of synqueries

**Figure 4. The performance against various parameter $\alpha$**

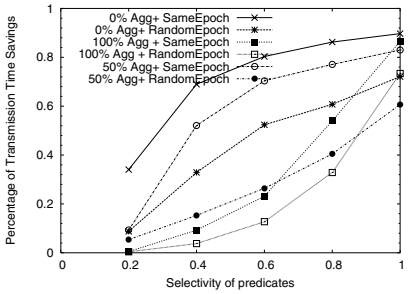



**Figure 5. Transmission time savings against various predicate selectivity**

The average transmission time by the two tiers shown are quite similar, and have both been significantly reduced by up to around 61% and 75% compared with that of the baseline when the number of nodes is 16 and 64 respectively.

For $WORKLOAD_B$, as designed, the average transmission time under in-network optimization is considerably smaller than that under base station optimization, as shown in Figure 3. Interestingly, the percentage of improvements by in-network optimization is much bigger in the network with 64 nodes than 16 nodes, compared with that of base station optimization. Since the number of radio messages at each node for aggregation queries will not increase with network size while that for data acquisition queries will be proportional to the network size, the number of radio messages under in-network optimization grows much slower than that under the base station optimization, and consequently the percentage of improvement on number of radio messages increases faster. As we analyzed in Section 3.1.2, the average transmission time increases with the number of radio messages, and thus the percentage of improvement on average transmission time increases faster.

The results under $WORKLOAD_C$ (see Figure 3) show that the TTMQO performs much better than applying in-network optimization or base station optimization separately. It shows that the two tiers are mutually complementary, and it is beneficial to apply in-network optimization after base station optimization. In-network optimization does not support the similarity sharing among aggregation queries and data acquisition queries, but base station optimization can support it in the finest granularity. By applying base station optimization first, the aggregation queries whose answers can be derived from data acquisition queries are suppressed from injecting into the sensor network, so the in-network optimization will not face the problem of doing extra work to answer these aggregation queries; moreover, with the common sharing that can be achieved by both tiers enabled at the base station, the in-network message size will not be unnecessarily enlarged. On the other hand, the in-network optimization can effectively handle the situation where the queries cannot be effectively rewritten by base station optimization due to epoch duration constraint. It is also interesting to note that: when the number of nodes is 16, base station optimization is more effective than in-network optimization; while the contrary is true when the number of nodes has increased to 64. This is due to the same reason that applies to the scenario where there is fast increase in the percentage of improvement by in-network optimization as network size grows which we have explained above. Our two-tier optimization scheme is shown to improve up to 82% in terms of the transmission time, which implies that it can save much bandwidth and energy.

### 4.3. Performance under adaptive workloads

We evaluate the TTMQO scheme against various adaptive workloads. First, we evaluate the scalability of our TTMQO scheme with the number of queries and study the effect of parameter $\alpha$ with a model of queries that randomly select attributes (nodeid,light,temp), aggregations (MAX, MIN), predicates and epoch durations (from shortest 8092ms to longest 24576ms, all divisible by 4096ms). We keep the average arrival frequency at 40s per query, but we vary the average duration so that the average number of concurrent queries is changing. A set of workload is complete after the termination of 500 queries. We divide the sum of benefit by the sum of the cost() of every query to get benefit ratio. Though we do not study skewed query workload, we expect the similarity to be greater among such workload, and the benefit can be even bigger.

Given random queries, as we can see in Figure 4(a), the benefit ratio increases significantly from around 32% to 82% as the number of current queries increases from 8 to 48. Comparing with the effect of number of concurrently running queries, the parameter $\alpha$ has less effect on the benefit ratio. As shown in Figure 4(b) , when there are 8 simultaneous queries, the most benefit is obtained when $\alpha$=0.6, which validates our analysis of Section 3.1.4. When $\alpha$ is too

small, the significantly overlapped remaining queries may be forced to rewrite with other synthetic queries which may incur less benefit than original old synthetic query; on the other hand, when $\alpha$ is too big, unnecessary data fetched for previously-existed queries may incur so much overhead that it is better to rewrite the remaining queries.

Figure 4(c) shows that our scheme can scale pretty well with the number of concurrent queries. The average number of synthetic queries is less than 4 even when the number of concurrent queries reaches 48. As the value of $\alpha$ increases, the average number of synthetic queries slightly decreases, because bigger value of $\alpha$ favors keeping the old single synthetic query instead of rewriting the remained queries into a new synthetic query set whose number is generally bigger than 1.

Next, we further evaluate our TTMQO scheme against workloads with various specific properties. More specifically, different composition of aggregation and data acquisition queries with predicates of different selectivity is utilized. In this experiment, the number of concurrent queries is 8; data acquisition queries retrieve all the attributes; aggregation queries request for MAX(light); selectivity of predicates = 0.6 means that one of the attributes (nodeid, light, temp) is randomly specified in the query predicate with a range coverage as 0.6. Figure 5 shows that the percentage of transmission time savings grows with selectivity of predicates for all workloads, because there is higher probability that queries request similar data, which also suggests that similarity among queries with same epoch duration or different epoch durations are both well exploited, and much savings are introduced by our TTMQO scheme. More carefully, we can see that when the selectivity of predicates is 1, 8 data acquisition queries with the same epoch duration achieves around 89.7% message savings, which is even more significant than the theoretical value $\frac{7}{8}$, because less result message transmission required by TTMQO incurs less transmission failure and radio message retransmission. And, it is interesting to note that with 100% aggregation queries, there is a sharp performance improvement when the selectivity of predicates reaches 1. This is because base station optimization cannot effectively optimize two data aggregation queries with different predicates due to semantic correctness constraints as discussed in section 3.1.2, and only in-network optimization scheme can take effect by selecting proper routes to enable aggregation as soon as possible and sharing data among queries when the value of their partial aggregation is the same.

## 5. Conclusion

In this paper, we have proposed a two-tier multiple query optimization scheme (TTMQO) to enable similar queries to share both communication and computational resources in the sensor network. Our experimental results showed that the TTMQO scheme can provide significant performance improvements, with lower cost of radio transmission (average transmission time), and can scale well with the number of concurrently running queries. Currently, our multi-query optimization algorithm has not taken into consideration of node failures and unreliable wireless transmissions that are inherent with wireless sensor networks. We plan to study quality-of-service driven multi-query optimization in the future. Furthermore, we would like to extend our multiple query optimization to support more complex queries such as self-join queries [15].

## References


[1] D. J. Abadi, S. Madden, and W. Lindner. REED: Robust, efficient filtering and event detection in sensor networks. In *VLDB*, 2005.

[2] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical in-network data aggregation with quality guarantees. In *EDBT*, 2004.

[3] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[4] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[5] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyos application. In *SenSys*, 2003.

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TINYDB: An acquisitional query processing system for sensor networks. *ACM TODS*, 30(1), November 2005.

[7] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[8] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[9] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *TKDE*, 2(262-266):23–54, June 1990.

[10] A. Silberstein, R. Braynard, and J. Yang. CONSTRAINT CHAINING: On energy-efficient continuous monitoring in sensor networks. In *SIGMOD*, 2006.

[11] A. Silberstein and J. Yang. Many-to-many aggregation for sensor networks. In *ICDE*, 2007.

[12] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.

[13] N. Trigoni, Y. Yao, J. Gehrke, R. Rajaraman, and A. Demers. Multi-query optimization for sensor networks. In *DCOSS*, 2005.

[14] S. Xiang, H. B. Lim, K. L. Tan, and Y. Zhou. Two-tier multiple query optimization for sensor networks. http://www.comp.nus.edu.sg/∼xiangshi/TTMQO.pdf.

[15] X. Yang, H. B. Lim, T. Ozsu, and K.-L. Tan. In-network execution of monitoring queries in sensor networks. In *SIDMOD*, 2007.

[16] Y. Yao and J. Gehrke. Query processing for sensor networks. In *CIDR*, 2003.