Optimizing Multiple Join-Queries over Sensor Data Streams

Yao-Chung Fan
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, R.O.C.
dr938318@cs.nthu.edu.tw

Arbee L.P. Chen*
Department of Computer Science
National Chengchi University
Taipei, Taiwan, R.O.C.
alpchen@cs.nccu.edu.tw

Abstract

Sensor networks have received considerable attentions in recent years and played an important role in data collection applications. Sensor nodes have limited supply of energy. Therefore, one of the major design considerations for sensor applications is to reduce the power consumption. In this paper, we study an application that combines RFID and sensor network technologies to provide an environment for path tracking, which needs efficient join-query processing. In addition to individual query optimization, we consider multi-query optimizations to reduce query evaluation cost, and therefore power consumption. We present algorithms for efficiently processing multiple join-queries with common evaluations. Moreover, extensive experiments are made to demonstrate the performance of the proposed optimization techniques.

1. Introduction

Sensor networks have received considerable attention in recent years. A sensor network consists of a large number of sensor nodes. In general, sensor nodes are equipped with the abilities of sensing, computing, and communicating.

One of the features for wireless sensor networks is resource limitations. Sensor nodes typically are limited in computing power, network bandwidth, storage capability, and energy supply. Resource conservation therefore becomes a major consideration when devising sensor applications.

Sensor networks provide a new way of data collection and create new needs for information processing in a variety of scenarios. In this paper, we introduce a new application that combines RFID (Radio Frequency Identification) and sensor network technologies for path tracking. In the application, efficient join processing is critical to the performance. We consider multi-query optimizations to reduce the cost of join processing, and therefore power consumption.

*To whom all correspondence should be sent.

Application Consider a set of sensor nodes deployed in a shopping mall, as shown in Figure 1. In addition to basic sensing modules, the sensor nodes are further equipped with an RFID reader. Moreover, the customers who enter the monitoring area are equipped with a RFID tag. When a customer passes through a sensor node, the sensor node detects the event and generates a tuple of data. The tuple contains the customer identification (CID), the timestamp when the tuple was generated (TID), and the sensor identification (SID). The sensor nodes inherently generate data streams if customers keep entering the area.

Such RFID and sensing platform provides an environment to infer customer activities from the sensor readings. For example, the shopping mall manager may want to know the customers who walk through nodes A, B, C, and D in the mall. Having such information, the manager may be able to infer that the customers are looking for something, and then make suitable assistances or recommendations to the customers. The manager may issue the following query:

Q_1: Select CID
From A, B, C, D
Where A.CID = B.CID = C.CID = D.CID
AND A.TID < B.TID < C.TID < D.TID
Window = 10 minutes
Action = *Service* (UID)

In this query, "Select" clause specifies the customer identifications, "From" clause indicates the sensor data streams, "Where" clause joins the readings among the indicated streams with suitable predicates, "Window" clause specifies the valid time constraint for the customers along the path, and "Action" clause indicates the proper action of providing some service to the qualified customers. We refer to such a query as a *path tracking* query.

One possible approach for processing this query is to ask sensor nodes A, B, C, and D to send the associated tuples within the window to a base station

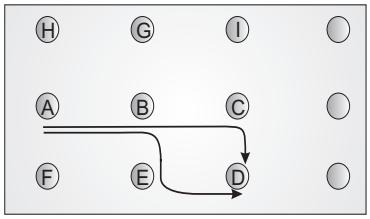


Figure 1 Motivating scenario

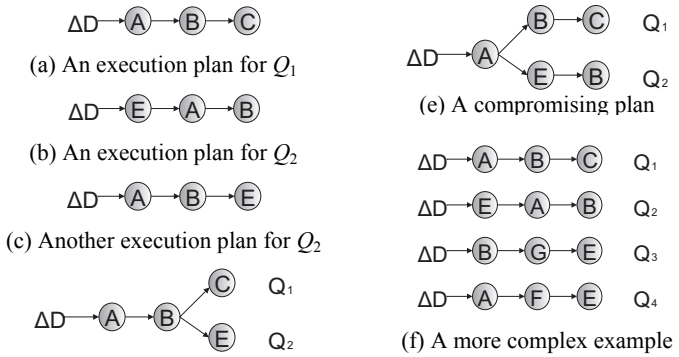


Figure 2 Execution Plans

to perform the evaluations. However, this approach suffers from high communication cost as each node transmits all the data to the base station regardless of whether the data contribute to the query results. This approach would be inefficient when the selectivity is low. In addition, the traffic bottlenecks at nodes near the base station can quickly deplete the nodes' energy and crash the network.

In this paper, we propose to in-network process Q_1 as follows. First, each of the participating nodes maintains a hash table that stores the tuples generated in the past ten minutes. A query execution plan is installed at node D to describe the order a new tuple generated at node D is probed [1] with the other hash tables. Figure 2(a) shows an execution plan for Q_1. In Figure 2(a), ΔD stands for a new tuple generated at node D, and ΔD→A→B→C indicates that Q_1 processes ΔD by first routing ΔD to node A to probe the associated hash table. If ΔD fails to match, the processing terminates. Otherwise, ΔD⋈A is routed to node B to probe the hash table of node B. Likewise, if ΔD⋈A is not dropped by node B, ΔD⋈A⋈B is routed to node C to probe the hash table of node C. A tuple that matches all other hash tables produces a join result, and then is returned to the base station. In this paper, we refer to routing a tuple or an intermediate result to some node to probe the associated hash table as a *task*.

We notice that another alternative for in-network processing Q_1 is to broadcast ΔD and the intermediate results for the query evaluation. However, this approach suffers from two limitations. First, the approach involves many communication collisions as multiple copies of ΔD and the intermediate results are propagated in the network simultaneously. Second, the approach produces redundant intermediate results as the evaluation is proceed in a parallel fashion. Both the limitations make the approach to be not energy efficient for practical use.

Now assume that in addition to Q_1, the manager poses another path tracking query as follows.

Q_2: Select CID
From A, B, D, E
Where A.CID = B.CID = E.CID = D.CID
AND A.TID < B.TID < E.TID < D.TID
Window = 10 minutes
Action = *Service2* (CID)

To process this query, we install another execution plan at node D, as shown in Figure 2(b). In such an environment with multiple queries, we study the problem of exploiting the common tasks among the queries to improve the query processing efficiency.

Challenges We identify two opportunities for optimizing query processing. First, we can *optimize the queries by choosing join orders*. The idea for *join-order optimizations* is as follows. Since a newly generated tuple that is dropped by one of the nodes cannot produce a join result, a worst case of the processing will be that a tuple matches all the nodes except the last one. In this case, no result is produced and all prior processing efforts are wasted. However, if we swap the last node with the first one for the probing, the tuple will get dropped in the first probe rather than in the last probe. Therefore, choosing an efficient join order is critical to efficient query processing.

The other optimization is to *share the common tasks among queries* that we will call *common-task-sharing optimizations*. It is usually beneficial to identify the common task among queries such that the redundant executions can be avoided. For example, we can plan Q_1 to be ΔD→A→B→C and Q_2 to be ΔD→A→B→E such that the tuple routing and the hash table probing for node A and node B can be shared. Figure 2(d) shows the resulting plan.

Both the optimization techniques reduce the cost for processing queries. However, the combination of these two techniques brings an interesting tradeoff to study.

In some cases, the two optimization techniques *complement* each other. For example, assume the plans in Figure 2(a) and Figure 2(c) are suggested by the join-order optimization strategy, we can further share

the common tasks {A, B} between the two plans to have a combined plan, as shown in Figure 2(d).

Nevertheless, in most cases, the two optimization techniques are *conflicting* to each other. For example, assume the join-order optimization strategy suggests the plan in Figure 2(b) instead of the one in Figure 2(c). In this case, when planning Q_1 and Q_2, if we choose to optimize the queries by choosing join orders, we lose the opportunities of sharing common tasks. On the other hand, if we choose to share the common tasks of node A and node B, we need to adopt the plan in Figure 2(c). If the cost saved by sharing is less than the cost incurred from the violation of the join-order optimality, sharing common tasks will be a bad decision.

Between the two extremes, there are other options for optimizing the query processing. We can loosen the optimality of the results provided by the two optimization techniques, and use the suboptimal results to plan query executions. For example, we can plan Q_2 to be $\Delta D \rightarrow A \rightarrow E \rightarrow B$ to share the routing and probing to node A with Q_1. Figure 2(e) shows the resulting plan. By sacrificing some optimality, this compromise may gain more benefit to the overall cost. As a result, one challenge for optimizing the queries lies in balancing the join-order optimizations and the common-task-sharing optimizations, that is, *combining both techniques for maximal benefits.*

In addition, the problem of optimizing multiple queries becomes more difficult as the number of queries involved increases. Consider now the example in Figure 2(f). We must consider the problem of which queries to be grouped to share common tasks. In Figure 2(f), no common task can be shared among all the queries. However, we still have opportunities to share tasks, e.g. the task {E} among Q_2, Q_3, and Q_4. As it can be seen, there are many ways to group the queries to consider task sharing. However, the decision of query grouping should be carefully made because the grouping can be mutually exclusive, prohibiting a query that shares some tasks with another query from sharing other tasks with other queries. For example, if Q_1 shares the task {A} with Q_4, then the sharing of Q_1 with Q_3 on the task {B} becomes infeasible. Consequently, another challenge lies in how to group queries such that the overall cost for query processing can be minimized.

In this paper, we study the above-mentioned challenges, and propose a new optimization method, which considers optimizing queries by combining join-order optimizations and common-task-sharing optimizations for maximal benefits.

Contribution In this paper, we make the following contributions.

- We introduce a novel application that combines RFID and sensor network technologies to provide an environment for path tracking.
- We investigate the problem of efficiently evaluating join-queries in the proposed application. In addition to individual query optimization, we consider exploiting multi-query optimizations.
- For a given optimization problem, we formulate the problem of determining an optimal solution and show the complexity of finding the solution. Furthermore, we propose a novel ε-approximation algorithm, which provide solutions with sub-optimal guarantees.
- Finally, we demonstrate the performance of the proposed techniques through extensive experiments. Our experimental results clearly demonstrate that our approach provides significant communication savings.

The rest of the paper is organized as follows. Section 2 discusses the cost model for the query processing. Section 3 presents our multi-query optimization techniques. Section 4 provides some practical considerations for real sensor devices. The experiment results are provided in Section 5. Related works are given in Section 6. Finally, we conclude the paper in Section 7 and give some directions for future research.

2. Cost Model

Terminology and Assumptions We first provide the terminology and the assumptions we use through this paper.

We consider a sensor network as one which consists of a set of sensor nodes $\{N_1, N_2, ..., N_n\}$ and a base station which has no energy and memory limitations. The sensor nodes are well-synchronized. The base station keeps the network topology and there are no communication delays in the sensor network. The sensor nodes generate a set of data streams $\Phi = \{S_1, S_2, ..., S_n\}$.

A path-tracking query Q installed at node N_j joins stream S_j with some other streams in Φ. We denote this subset of Φ as θ, and use the notation $Q(O)$ to refer to the execution of Q following a given join order O. A join order O is a permutation $O = <S_{O1}, S_{O2}, ..., S_{O|\theta|}>$ of the elements in θ. We use $Cost(Q(O))$ to denote the cost for processing ΔS_j using $Q(O)$.

All path-tracking queries are posed at the base station, from which the queries are planned and disseminated into the sensor network. All join results are collected at the base station. In this paper, we focus on the in-network join processing of the queries. The further processing of join results, such as making recommendations, take places at the base station, and

are beyond the scope of this paper. In this study, we assume that all path-tracking queries have the same window specification. Further Relaxation of this assumption can be a future direction to proceed.

In the following, we elaborate on the mechanism of processing a path-tracking query Q in a sensor network, and present a cost model for the processing of the path-tacking query. The model helps us to make a decision on choosing a good join order.

In processing ΔS_j for Q using a given join order $O = <S_{O1}, S_{O2}, ..., S_{O|\theta|}>$, three phases are required. In the first phase, ΔS_j is inserted into the hash table maintained in node N_j. In the second phase, ΔS_j is routed to node N_{O1}, and the associated hash table is probed by ΔS_j for matching. If ΔS_j fails to match, it is not processed further. Otherwise, $\Delta S_j \bowtie S_{O1}$ is routed to the next sensor node to continue query processing. A join result will be produced when $\Delta S_j \bowtie ... \bowtie S_{O|\theta|-1}$ matches with the table of the stream $S_{|\theta|}$.

There are two types of costs in the processing:

- ***Routing***: the cost for routing ΔS_j and the associated intermediate results during the query execution.
- ***Probe***: the cost for probing ΔS_j for matching with the associated hash tables during the query execution.

Probe includes the cost of probing ΔS_j for matching with the associated hash tables. This cost depends on the join order. Note that not every ΔS_j probes all hash tables. ΔS_j goes to a next node only if it matches with the hash table in the previous node. Therefore, ***Probe*** can be specified as follows.

$$\textbf{\textit{Probe}} = \sum_{i=1}^{|\theta|} i \cdot \beta \cdot \sigma_i + |\theta| \cdot \beta \cdot (1 - \sum_{i=1}^{|\theta|} \sigma_i)$$

Where σ_i stands for the probability of $\Delta S_j \bowtie ... \bowtie S_{Oi-1}$ being dropped by S_{Oi}, where $1 < i < |\theta|$, and β is the cost for performing a probe operation at a node.

Routing includes all transmission cost for routing ΔS_j and the associated intermediate results during the query execution. As compared to ***Probe***, ***Routing*** further depends on the hop distance between nodes. ***Routing*** can be specified as follows.

$$\textbf{\textit{Routing}} = \sum_{i=1}^{|\theta|} (\gamma \cdot \sigma_i \cdot \sum_{j=1}^{i} d_j) + \sum_{i=1}^{|\theta|} d_i \cdot \gamma \cdot (1 - \sum_{i=1}^{|\theta|} \sigma_i)$$

Where d_i denotes the minimum number of hop distance between N_{Oi} and N_{Oi-1}, where $1<i<|\theta|$, and γ is the cost for transmitting a tuple at a node. Note that $N_{Ot}=N_j$.

In total, $Cost(Q(O))$ for processing ΔS_j using the join order O can be given by

$Cost(Q(O))$ = ***Probe*** + ***Routing***

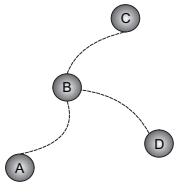


Figure 3 A Routing Topology

Example 1 We use Q_1 to illustrate the cost model. Assume Q_1 uses $O = <$ B, C, A>, and the routing topology is as shown in Figure 3. Let $\beta =$ 20μJ and $\gamma =$ 100μJ and let $\sigma_1 = 0.3, \sigma_2 = 0.1, \sigma_3 = 0.5$.

In processing ΔD using $O = <$ B, C, A>, there are four cases to consider:

Case 1: ΔD is dropped by node B

In this case, the cost for processing ΔD is 0.3*(1*20μJ+1*100μJ)=36μJ since if ΔD dropped by node B, one probe operation and one hop transmission are performed.

Case 2: ΔD⋈B is dropped by node C

In this case, the cost for the processing is 0.1*((1+1)*20μJ+(1+1)*100μJ)=24μJ since two probe operations and two hop transmissions are performed.

Case 3: ΔD⋈B⋈C is dropped by node A

In this case, the cost for the processing is 0.5*((1+1+1)*20μJ+(1+1+2)*100μJ)=230μJ.

Case 4: ΔD⋈B⋈A⋈C is produced

If ΔD⋈B⋈C is not dropped by node A, the join result ΔD⋈B⋈A⋈C is produced. The processing cost is 0.1*(3*20μJ+4*100μJ)=46μJ since three probe operations and four hop transmissions are needed.

Thus, the total cost is 36μJ+24μJ+230μJ+46μJ= 336μJ in average.

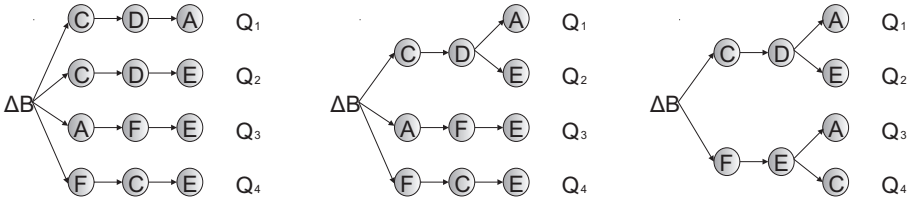
3. Multiple Query Optimizations

This section presents optimization techniques for the query executions. In the subsequent discussion, we refer to the execution plan for processing a query individually as a *local execution plan*, and use the term *global execution plan* to refer to a plan that provides a way to compute results for multiple queries.

3.1 Join-order Optimization

Given a path-tracking query Q to be installed at node N_j, the goal of the join-order optimizations is to choose a join order for Q such that $Cost(Q(O))$ can be minimized.

In the processing, not every ΔS_j probes all hash tables. The intermediate result of ΔS_j goes to a next



(a) A simple plan **(b)** A naïve sharing plan **(c)** Yet another sharing plan

Figure 4

node only if it matches with the hash table in the previous node. Once an intermediate result gets dropped, all prior processing efforts are wasted.

One naive method for finding optimal join order is to enumerate all possible orders. Nevertheless, this approach results in the complexity of $O(|\theta|!)$. Therefore, the AGreedy proposed in [2] is often used to determine join orders for stream join-queries. The AGreedy works by greedily choosing the stream that drops the maximum number of the tuples at each step to determine a join order. However, in AGreedy, the streams are assumed to be collected in a central station. As mentioned earlier, such framework is not energy efficient for sensor networks. In this study, we extend the AGreedy by further considering the communication cost.

The original idea behind the AGreedy method is that if a tuple ΔS_j eventually gets dropped, it is better that it gets dropped as early as possible such that the processing cost can be minimized. Therefore, when choosing S_{Oi}, $1 < i < |\theta|$, the stream which most likely drops $\Delta S_j \bowtie ... \bowtie S_{Oi-1}$ should be chosen. However, if the chosen stream is far away from N_{Oi-1}, the execution will instead incur a great deal of message traffics. Therefore, we choose join orders by balancing the two factors, i.e., we choose the stream that most likely drops $\Delta S_j \bowtie ... \bowtie S_{Oi-1}$and that needs as few hop transmissions as possible to reach.

We use Q_1 as an example to illustrate how our join-order optimization approach works.

Example 2 Let the probability of ΔD being dropped by A is 0.5, the probability of ΔD being dropped by B is 0.3, and the probability of ΔD being dropped by C is 0.1. The routing topology is as shown in Figure 3.

For ease of illustration, we assume the probabilities are independent.

The join-order optimization proceeds as follows. First, node B is chosen to be S_{O1} because it provides an expectation of 0.3/1=0.3 (the expected number of ΔD to be dropped per number of hop transmissions) while node A only provides an expectation of 0.5/2=0.25 and node C only provides an expectation of 0.1/2=0.05. Then, node A is chosen to be S_{O2} since it provides an expectation of 0.5 to drop ΔD⋈B while node C only provides an expectation of 0.1. Finally, node C is chosen to be S_{O3} and the order O = <B, A, C> is outputted.

3.2 Naïve Sharing Strategy

Having described the join-order optimizations, we now turn to the problem of optimizing multiple queries. Let us first consider an example for optimizing multiple queries.

Example 3 Given a set of queries $W = \{Q_1, Q_2, Q_3, Q_4\}$ to be installed at node B, with θ_1 = {A, C, D}, θ_2 = {C, D, E}, θ_3 = {A, E, F}, and θ_4 = {F, C, E}.

The first option for optimizing the queries is to optimize the queries individually by the join-order optimizations. Figure 4(a) shows the global execution plan for processing ΔB, which is constituted by four locally optimized execution plans. We can see no common tasks between the plans are shared in Figure 4(a).

One straightforward method to further improve the global execution plan in Figure 4(a) is to combine the locally optimized plans which have common prefixes. For example, we can combine the plans for Q_1 and Q_2 in Figure 4(a) to produce a global execution plan as shown in Figure 4(b). This execution plan is more efficient than the plan in Figure 4(a), because the redundant tasks are avoided and the plan does not violate the optimized join-orders. We refer to this strategy as *naïve sharing strategy*.

However, the naïve sharing strategy restricts the scope of optimizations. The main problem with naïve sharing strategy is that more than one possible local execution plan can be used to produce query results. The naïve sharing strategy, however, only considers the locally optimized plans. For example, although there is no common prefix between the locally optimized plans of Q_3 and Q_4, we still can plan Q_3 to be <F, E, A> and Q_4 to be <F, E, C> such that the common tasks {F, E} can be shared. Figure 4(c) shows the resulting plan, which can be more efficient than the one shown in Figure 4(b). In fact, in addition to sharing {F, E}, the queries have other tasks in common,

and the naïve sharing strategy does not consider them to be shared, potentially missing further optimization opportunities.

3.3 Hybrid Optimization

In this section, we study the problem of combining the join-order optimizations and the common-task-sharing optimizations for maximal benefits. We first discuss the construction of the search space for the optimization (Section 3.3.1), and then propose a randomized algorithm for finding the optimal solution (Section 3.3.2).

3.3.1 Search Space

We first state some definitions for the subsequent discussions. The definitions are helpful to describe elements in the search space.

Given a set of queries $W = \{Q_1, Q_2, ..., Q_m\}$ to be installed at node N_j. One prerequisite for queries to share a task(s) is that the queries have the task(s) in common. In the following discussion, we refer a set of tasks as a *task-set*. We say a query Q *contains* a task-set if the task-set is contained in θ. The *count* of a task-set is the number of queries containing the task-set. In Example 3, {C, D} is a task-set of Q_1, and the count for the task-set {C, D} is 2.

Definition 1: We say a task-set is a *common task-set*, *CTS*, among $W = \{Q_1, Q_2, ..., Q_m\}$ if and only if the count of the task-set is equal to or larger than two, i.e., at least two queries in W contain the task-set.

Definition 2: For a given *CTS*, we call a query set w, $w \subseteq W$, whose elements all contain the *CTS* as the *sharing set* of the *CTS* and denote it as w_{CTS}.

In Example 3, {C, D} is a *CTS* and $w_{\{C, D\}} = \{Q_1, Q_2\}$ is the sharing set of CST{C, D}.

Definition 3: A *sharing cost*, c_{CTS}, indicates the minimal cost of processing the queries in w_{CTS} by sharing the associated *CTS*. This cost can be formulated as

$$c_{CTS} = Cost^*(CTS) + \sum_{Q_i \in w_{CTS}}^{m} Cost^*(\theta_i - CTS),$$

where $Cost^*(CTS)$ is the cost for processing the *CTS* by using the plan optimized by the join-order optimizations and $Cost^*(\theta_i - CTS)$ is the cost of the executions that cannot be shared (also optimized by join-order optimizations).

A sharing set and the associated sharing cost implicitly describe an execution that shares the corresponding *CTS*. In Example 3, $w_{\{C,D\}}=\{Q_1, Q_2\}$ and $c_{\{C,D\}}$ describe the combined execution that shares the common task {C, D} for Q_1 and Q_2, as the plan shown in Figure 4(b).

The construction of the search space proceeds as follows. The first step is to derive all *CTS* from W. The process of finding *CTS* amounts to the process of deriving the frequent item-set [4] with count $\geqq 2$ from $\{\theta_1, \theta_2, ..., \theta_m\}$as only the task-set with count $\geqq 2$ are the *CTS* for queries in W.

Procedure *Search_Space_Constructer*
Input: A set of queries $W=\{Q_1, Q_2, ..., Q_m\}$ with $\theta_1, \ldots, \theta_m$.
Output: A matrix **A** and a cost vector **c**
0. Initialize n = 1 *// recording number of sharing vectors*
1. Find all *CTSs* from W
2. **For** each *CTS*,
3. Find the corresponding sharing set, say *SS*.
4. **For** $s \in 2^{SS}$-{ e| $e \in 2^{SS}$ and $|e| \leq 2$},
where 2^{SS} is a power set of *SS*.
5. Create a vector $\mathbf{a}_n \in \{0, 1\}^m$,
where a_k=1 if $Q_k \in s$; otherwise a_k=0, for k = 1, ..., m.
6. $c_n = Cost^*(CTS) + \sum_{i=1}^{m} Cost^*(\theta_i - CTS) \times \mathbf{a}_n^i$
7. n = n + 1
8. **Endfor**
9. **Endfor**
10. **For** each $Q_i \in W$
11. Create a vector $\mathbf{a}_n \in \{0, 1\}^m$,
where a_k=1, if k = i; otherwise a_j=0, for k = 1, ..., m.
12. $c_n = Cost^*(\theta_i)$
13. n = n + 1
14. **Endfor**
15.**Return** an m × n matrix **A** and a vector **c**,
where $\mathbf{A}_i = \mathbf{a}_i$ and $c_i = c_i$, for i = 1, ..., n.

Figure 5 The Search Space Constructor

In the second step, for each *CTS* derived from W, we enumerate its associated sharing sets. Note that there is more than one sharing set for an *CTS*. By definition, any subset S of a sharing set with $|S| \geq 2$ is also a sharing set. In Example 3, the subsets $\{Q_1, Q_2\}$, $\{Q_1, Q_4\}$ and $\{Q_2, Q_4\}$ of $w_{\{C\}} = \{ Q_1, Q_2, Q_4 \}$ are also sharing sets for {C}, because if the queries can share {C}, then any combination of the queries definitely can share the common task {C}.

Then, for each sharing set w_{CTS}, we compute its sharing cost c_{CTS} and represent the w_{CTS} by a vector $\mathbf{a}_{CTS} \in \{0, 1\}^m$ whose component a_i equals to one if $Q_i \in w_{CTS}$, and zero otherwise. In Example 3, $\mathbf{a}_{\{C\}}$ = (1, 1, 0, 1) represents that Q_1, Q_2, and Q_4 share the *CTS*{C}.

Fourth, the above steps find all possible executions that share common tasks for queries. However, there are also possibilities that queries share nothing, i.e., the queries use their locally optimized plans. Therefore, we also consider the options of the individual execution for the queries.

To establish the search space, we combine the vectors, which represent the sharing sets, into a matrix **A** (column wise) and their associated sharing costs into

0	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	0	0
1	0	1	0	1	1	0	1	1	1	1	1	0	0	0	1	0	0
0	1	0	1	0	0	0	0	0	1	1	0	1	1	0	0	1	0
1	1	0	0	1	0	1	1	0	1	0	1	1	1	0	0	0	1
{C,E}	{E,F}	{C,D}	{A}	{C}	{C}	{C}	{C}	{D}	{E}	{E}	{E}	{E}	{F}	{}	{}	{}	{}
c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}

Figure 6 The search space for *W*

a cost vector **c**.

The search space for the problem can be then formulated as

Minimize $\mathbf{c} \cdot \mathbf{x}$

Subject to $\mathbf{A} \cdot \mathbf{x} = \mathbf{e}$,

$$x_i \in \{1,0\},\ i = 1, ..., n$$

The vector **x** contains components x_i, i = 1, ..., n, which equals to 1 if $\mathbf{A}_i$ is chosen, and 0 otherwise. **e** denotes the vector whose components all equal to one. A vector **x** satisfying $\mathbf{A} \cdot \mathbf{x} = \mathbf{e}$ is called a *feasible solution*. Each feasible solution corresponds to a global execution plan for processing *W*. The algorithm for the search space construction is given in Figure 5.

Our goal is to find a vector **x*** with subject to **A**, which corresponds to a global execution plan that minimizes the total cost for processing *W*. Theorem 1 shows that this problem is NP-Complete.

Theorem 1*: Given a set of queries* $W = \{Q_1, Q_2, ..., Q_m\}$*, the problem of selecting one join order for each query with the goal of minimizing* $\sum_{i=1}^{m} Cost(Q_i(O))$ *is NP-Complete.*

*Proof Idea***:** By showing the equivalence between a set partition problem [3], a well-known NP problem, and our problem, we say that no polynomial time algorithm is expected to exist. See Appendix for the proof. ■

Example 4 The search space for *W* in Example 3 is constructed as follows.

Step1: Find *CTS*s from *W*. We have {C, E}, {E, F}, {C, D}, {A}, {C}, {D}, {E} and {F}.

Step2: For each *CTS*, we enumerate the associated sharing sets and then generate a vector and a sharing cost to represent each enumerated sharing set. For {C, E}, we have the sharing vector $\mathbf{a}_1$=(0, 1, 0, 1) and the sharing cost c_1 with respect to the sharing set $w_{\{C, E\}}$={ Q_2, Q_4 }. Likewise, we have $\mathbf{a}_2$=(0, 0, 1, 1) and c_2 for {E, F}, $\mathbf{a}_3$=(1, 1, 0, 0) and c_3 for {C, D}, and $\mathbf{a}_4$=(1, 0, 1, 0), and c_4 for {A}. Note that, for {C}, we will have $\mathbf{a}_5$=(1, 1, 0, 1), $\mathbf{a}_6$=(1, 1, 0, 0), $\mathbf{a}_7$=(1, 0, 0, 1), and $\mathbf{a}_8$=(0, 1, 0, 1) with respect to the combinations of the sharing set $w_{\{C\}}$, i.e. {Q_1, Q_2, Q_4} {Q_1, Q_2}, {Q_1, Q_4} and {Q_2, Q_4}. Likewise, for {D}, we have $\mathbf{a}_9$=(1, 1, 0, 0). For {E}, we have $\mathbf{a}_{10}$=(0, 1, 1, 1), $\mathbf{a}_{11}$=(0, 1, 1, 0), $\mathbf{a}_{12}$=(0, 1, 0, 1), and $\mathbf{a}_{13}$=(0, 0, 1, 1) with respect to the combinations of the sharing set $w_{\{E\}}$. For {F}, we have $\mathbf{a}_{14}$=(0, 0, 1, 1)

Step3: For the options of the individual executions for the queries, we have $\mathbf{a}_{15}$=(1, 0, 0, 0), $\mathbf{a}_{16}$=(0, 1, 0, 0), $\mathbf{a}_{17}$=(0, 0, 1, 0), $\mathbf{a}_{18}$=(0, 0, 0, 1), and the associated sharing costs.

Step4: Finally, we have a 4×18 matrix **A** and the corresponding cost vector **c**, where $\mathbf{A}_i = \mathbf{a}_i$ and $c_i = c_{i,}$ i = 1, ..., 18.

Figure 6 shows the search space for *W*, where columns describe all possible executions and the associated *CTSs*. For example, the first column describes the execution that shares {C, E} between Q_2 and Q_4 with cost c_1. ■

3.3.2 ε -approximation Solution

Guaranteeing an optimal solution for the hybrid optimization is impractical because it is an NP-Complete problem. Moreover, optimal solutions may fluctuate over time in the face of data stream environments. Therefore, in the following, we consider approximate solutions for the hybrid optimization. We extend the linear programming rounding known as LP-rounding [5] and propose a novel randomized algorithm that we call *Randomized Rounding*.

Definition 4**:** We say an algorithm constitutes an ε -approximation algorithm for a minimization problem with optimal cost *opt*, if the algorithm runs in polynomial time and returns a feasible solution with cost opt^+, such that $opt^+ \le (1+\varepsilon)*opt$, for any $\varepsilon > 0$. We refer the solution as an ε -approximate solution.

For the problem: minimizing $\mathbf{c} \cdot \mathbf{x}$ subject to $\mathbf{A} \cdot \mathbf{x} = \mathbf{e}$ and $\mathbf{x} \in \{0, 1\}^n$ formulated in Section 3.3.1, the Algorithm *Randomized Rounding* proceeds as follows. We first solve the relaxed version of the given problem in which the integrality constraint ($x_i \in \{0, 1\}$, i = 1, ..., n) is replaced by (x_i = [0, 1], i = 1, ..., n). That is, we allow x_i to assume real values between 0 and 1. The relaxed problem, a linear programming problem, can be solved in polynomial time. Let $\mathbf{x}^+ = \{x^+_1, ..., x^+_n\}$ be the solution of the relaxed problem. Note that the components of $\mathbf{x}^+$ might not be integral but fractional, implying that $\mathbf{x}^+$ is not a feasible solution for our problem.

To obtain a feasible solution, we randomly round these fractional values. More precisely, we treat $\mathbf{x}^+$ as a probability vector, and for i =1, …, n, set x_i to one with probability x^+_i. After this rounding process, we obtain a solution whose components take values from {0, 1}, which probably is a feasible solution for the problem. We then verify whether the solution is feasible and ε-approximate. The condition whether a solution is feasible and whether a solution is ε-approximate can be verified in polynomial time. In case the solution is not fulfilled, we repeat the rounding process to obtain another solution. After a provable expected-number of repeating times, we can get a feasible solution together with a guarantee on its degree of the sub-optimality. The detailed algorithm is given in Figure 7.

The probability to obtain a feasible solution can be small. However, if we keep repeating the rounding process, the possibility to obtain a feasible solution grows. The algorithm we propose is based on such principle. Theorem 2 shows the probability for Procedure *Rounding* in Figure 7 to output a feasible solution satisfying approximate guarantees. Theorem 3 shows the expected number of repeating times to get a feasible solution.

For ease of presentation, we denote $[e^{\delta}/(1+\delta)^{(1+\delta)}]^{\mu}$ by $F(\mu,\delta)$ in the following discussion.

Lemma 1: *With probability at least*

$$F(m*(\frac{e-1}{e}),\frac{1}{e-1}) - F(m*(\frac{e-1}{e}),\frac{m-1}{m}*\frac{e}{e-1}-1),$$

Procedure Rounding returns a feasible solution.

Proof: In a rounding process, the probability that a query Q_i is not planned is: $\mathbf{Pr}[Q_i \text{ is not planned}] = \prod_{k:A_k^i=1}(1-x_k^+)$. Let n_i be the number of ways to execute Q_i, that is, the number of **A**'s column whose the i-th component equals to 1. Since $\sum_{k:A_k^i=1} x_k^+ = 1$, it is clear that $\mathbf{Pr}[Q_i \text{ is not planned}]$ is maximized when $x_k^+ = 1/n_i$. Therefore, we get

$\mathbf{Pr}[Q_i \text{ is not planned}] = \prod_{k:A_k^i=1}(1-x_k^+) \le (1-1/n_i)^{n_i} \cong 1/e$.

Conversely, $\mathbf{Pr}[Q_i \text{ is planned}] = 1-1/e$. Then, let X_i be the indicator random variable, i.e., $X_i = 1$ if Q_i is planned, otherwise $X_i = 0$. Formally, we have $\mathbf{Pr}[X_i=1] = 1-1/e$, for i = 1, ..., m.

Let $\mathbf{X} = \sum_{i=1}^{m} X_i$, which states in one rounding process the number of queries to be planned. We have $E[\mathbf{X}] = \sum_{i=1}^{m} 1*\Pr[X_i=1] = m*(1-1/e)$.

Algorithm *Randomized_Rounding*
Input: Minimizing $\mathbf{c}\cdot\mathbf{x}$ subject to $\mathbf{A}\cdot\mathbf{x}=\mathbf{e}$, $\mathbf{x}\in\{0,1\}^n$
Output: a solution for our problem **x**
Begin
1. Solve $\mathbf{A}\cdot\mathbf{x}=\mathbf{e}$, where $x_j=(0,1), \forall j\in n$
2. Treat the solution $\mathbf{x}^+=\{x^+_1, \ldots, x^+_n\}$ as a probability vector
3. **While ($\mathbf{A}\cdot\mathbf{x}$!= $\mathbf{e}$)**
4. Set $\mathbf{x}=\mathbf{0}$
5. *Rounding* (**x**, $\mathbf{x}^+$)
6. **End**
7. **Return x**
End

Procedure *Rounding*
Input: a decision vector **x**, a probability vector $\mathbf{x}^+$
Output: a decision vector **x** whose components are randomly set.
Begin
1. **For** i = 1 **to** n
2. **Set** x_i **to** 1 with probability x^+_i
3. **Endfor**
4. **Return x**
End

Figure 7 ***Randomized Rounding***

The event that Procedure *Rounding* outputs a feasible solution corresponds to [**X** = m], meaning that all queries are planned. We now derive $\mathbf{Pr}[\mathbf{X}=m]$.

Chernoff Bound Let Y_1, ..., Y_m be independent random variables such that for i=1, ..., m, $\mathbf{Pr}[Y_i=1] = p_i$, where $0< p_i <1$.

Then, for $\mathbf{Y}=\sum_{i=1}^{m} Y_i$, $\mu = E[\mathbf{Y}] = \sum_{i=1}^{m} p_i$, and $\delta>0$,

$\mathbf{Pr}[\mathbf{Y} \le (1+\delta)\mu] \ge 1-[e^{\delta}/(1+\delta)^{(1+\delta)}]^{\mu}$.

Apply the Chernoff Bound to **X**, we can find that

$$\mathbf{Pr}[\mathbf{X}\le m] \ge 1 - F(m*(\frac{e-1}{e}),\frac{1}{e-1}), \text{ and}$$

$$\mathbf{Pr}[\mathbf{X}\le m-1] \ge 1 - F(m*(\frac{e-1}{e}),\frac{m-1}{m}*\frac{e}{e-1}-1).$$

By subtracting the above two inequalities, we have

$$\mathbf{Pr}[\mathbf{X}=m] \ge F(m*(\frac{e-1}{e}),\frac{1}{e-1}) - F(m*(\frac{e-1}{e}),\frac{m-1}{m}*\frac{e}{e-1}-1)$$

and the lemma follows. ■

Lemma 2: *For every $\varepsilon>1$, with probability at least $1-1/\varepsilon$, Procedure Rounding returns a solution with a cost lower than $\varepsilon*opt$, where opt denotes the optimal cost for our optimization problem.*

Proof: Let **x** be the solution returned by Procedure *Rounding*. For the expected cost of $\mathbf{c}\cdot\mathbf{x}$, we have $E[\mathbf{c}\cdot\mathbf{x}] = \sum_{i=1}^{n} c_i * x_i^+$, which is the same as the optimal cost to the relaxed problem. We call the optimal cost for the relaxed problem as opt_f. Note that $opt \ge opt_f$.

Markov's Inequality Let Y be a random variable assuming only non-negative values. Then, for all t >0, $\mathbf{Pr}[Y \geq t] \leq E[Y]/t$.

Applying the Markov's Inequality to $\mathbf{c} \cdot \mathbf{x}$, we have $\mathbf{Pr}[\ \mathbf{c} \cdot \mathbf{x} \geq \varepsilon * opt_f\ ] \leq (opt_f / \varepsilon * opt_f) \leq 1/\varepsilon$. Conversely, $\mathbf{Pr}[\ \mathbf{c} \cdot \mathbf{x} < \varepsilon * opt_f\ ] > 1 - 1/\varepsilon$. As we note that $opt \geq opt_f$, we have $\mathbf{Pr}[\ \mathbf{c} \cdot \mathbf{x} < \varepsilon * opt_f \leq \varepsilon * opt\ ] > 1 - 1/\varepsilon$. ■

For ease of presentation, we denote $(F(m*(\frac{e-1}{e}),\frac{1}{e-1}) - F(m*(\frac{e-1}{e}),\frac{m-1}{m}*\frac{e}{e-1}-1))*\frac{\varepsilon-1}{\varepsilon}$ by $\Phi(m,\varepsilon)$ in the following discussion.

Theorem 2: *With probability at least* $\Phi(m,\varepsilon)$ *, Procedure Rounding constitutes an* ε *-approximation algorithm for our problem.*

Proof: By Lemma 1, Procedure *Rounding* returns a feasible solution with probability at least $F(m*(\frac{e-1}{e}),\frac{1}{e-1}) - F(m*(\frac{e-1}{e}),\frac{m-1}{m}*\frac{e}{e-1}-1)$. By Lemma 2, Procedure *Rounding* returns a solution with a low cost with probability at least $1-1/\varepsilon$. Therefore, the probability that Procedure *Rounding* output a feasible solution with a cost lower than $\varepsilon * opt$ can be given as $\Phi(m,\varepsilon)$. ■

We can run Procedure *Rounding* multiple times until it is successful. We expect to have a complexity of $O(1/\Phi(m,\varepsilon))$ to obtain an ε -approximation solution.

Theorem 3: Algorithm *Randomized_Rounding is expected to invoke Procedure Rounding* $\lceil (1/\Phi(m,\varepsilon)) \rceil$ *times.*

***Proof*:** Algorithm *Randomized_Rounding* terminates only when Procedure *Rounding* successfully output an ε -approximation solution. By Theorem 2, Procedure *Rounding* outputs an ε -approximation solution with probability $\Phi(m,\varepsilon)$. Therefore, the expected calling times of Procedure *Rounding* can be given by $\sum_{i=1}^{\infty} i * \Phi(m,\varepsilon) * (1-\Phi(m,\varepsilon))^{i-1} = 1/\Phi(m,\varepsilon)$, and our theorem follows. ■

4. Practical Considerations

4.1 Memory Constraints

Since our plans are executed inside a sensor network, it is important to ensure that the workload does not exceed the capabilities of an individual sensor node. One of the resource constraints for a sensor node is the limited memory. Existing sensor nodes only have a few kilobyte of memory, e.g, the memory for Tmotesky nodes is only 10 KB. This limitation raises the concern of how to maintain arbitrary large hash tables for query processing.

One straightforward solution for the problem is to move a portion of data that is out of a node's memory capability back to BS. In this solution, if the N_i's memory is out of use, stream S_i requires to maintain its hash table at two places, one at the N_i's memory and the other at BS. However, the solution will incur a great deal of energy for communications. When a new tuple generated from the streams S_j, $i \neq j$, and it requires S_i to probe for matching, the tuple needs to travel to N_i and BS to make sure whether S_i will drop the tuple. This mechanism inevitably invokes at least two message traffics and therefore consumes a great deal of energy.

The reason for transmitting ΔS_j to the BS is that the tuples maintained at the BS can contribute to the join result of ΔS_j. However, if we can be sure that the tuples never contribute to the join result, we can suppress the unnecessary communications, i.e. transmitting ΔS_j to the BS, to reduce the energy consumption. Therefore, based on this idea, we propose a strategy to cope with the memory limitation as well as the unnecessary communications.

We make use of the Bloom filter (BF) to detect the unnecessary communications. A BF contains a bit array of size *m* and a set of hash function $\{h_1(x), ..., h_K(x)\}$. All bits in the array are initially set to zero and the hash functions are assumed to be uniform and independent to each other. The BF is shown to be useful for representing the presence of a set of elements.

Our problem can be stated as follows: given a sensor data stream, S_i, and a certain amount of space, *M*, we want to maintain a sliding window with size *N*, $N > M$, over S_i. Since $N > M$, some tuples have to be stored at the BS. Many replacement polices can be used to select the tuples to be stored at the BS. We take the simplest scheme; we move the oldest tuple to the BS and add a new one to the local memory. In addition to moving tuples to the BS, we use BF to construct a synopsis of the tuples stored at the BS. We spare a small portion of *M*, *m*, to maintain the BF, and, for each tuple *x* to be moved to the BS, we set the bits $\{h_1(x.a), ..., h_K(x.a)\}$ of the BF to one, where *a* is the join attribute of *x*.

As a result, when we need to ascertain whether ΔS_j can join with the tuples maintained in the BS, we check the bits $\{h_1(\Delta S_j.a), ..., h_K(\Delta S_j.a)\}$ of the BF in N_i. If one of these bits is zero, with 100% confidence we know that the tuples in the BS will never join with ΔS_j. Otherwise, the tuples may contribute to the join result, and we have to transmit ΔS_j to the BS.

There are three things to note about our strategy. First, there is a possibility that ΔS_j is transmitted but not joined with the tuples in the BS. This error occurs because the bits $\{h_1(\Delta S_j.a), ..., h_K(\Delta S_j.a)\}$are set by elements other than ΔS_j.a. The probability is often known as *false positive rate* and can be minimized to $(1/2)^{ln(2)(m/n)}$ under the optimal selection of $k=ln(2)(m/n)$, where n is the number of distinct elements seen so far.

Second, when more and more tuples are generated, the fraction of zeros in the BF will gradually decrease. This leads to a result that the false positive rate eventually reaches to 1. In this case, the BF becomes useless since all ΔS_j will then be required to be transmitted to the BS. To deal with this problem, our solution is to periodically emend the BF with the data in the BS. The issues of how to smartly avoid a full BF can be a future direction.

Third, some existing nodes are equipped with an external non-volatile flash storage, e.g., Tmotesky nodes have 1 MB flash storage. We can maintain a portion of data in the flash storage instead of the BS. Although the flash I/Os are slow and consume energy as much as the power of invoking a radio communication, if a node is distant from the BS, keeping data in the local storage is still energy-efficient.

4.2 Handling Failures

Reliability is another concern for sensor applications because of high failure rates for both nodes and communications. To improve the reliability, we can leverage the broadcast nature of wireless communications of the sensor nodes by having each node broadcast its packets to multiple neighbors, known as *multi-path routing* scheme [18]. The multi-path routing scheme is very robust to the communication failures because each packet has multiple copies sent inside the network and only when all the packets fail the packet will be lost.

However, the multi-path scheme comes along with the problem of generating duplicate query results. If more than one duplicate packet reaches a destination node, the duplicate query results will be generated. To cope with the problem, we can exploit the temporal attributes of a routed tuple to filter out the duplicate query results. Toward this goal, we propose a caching scheme: each node keeps each newly received tuple for a constant time related to the *network diameter*, which is the maximum delay time between the duplicates of a tuple. When a destination node receives a new tuple, it first ascertains whether the tuple is a duplicate. If not, process the tuple as it is planned. Otherwise, ignore the tuple.

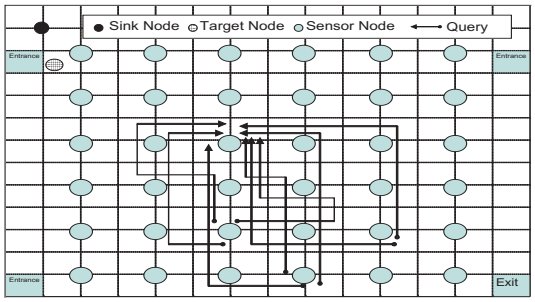


Figure 8 Experiment setup

5. Performance Evaluation

5.1 Prototyping Experiences

We have completed an initial implementation for the path tracking application by adopting Tmotesky nodes[6] and SkyeRead-M1-Mini RFID reader[7]. All programs in the base station are coded using Java, and the sensor network is programmed by Maté[8]. The execution plans of the given queries are optimized at the base station (BS), and then disseminates into the network by Maté. The dissemination begins with the broadcast of Maté capsules from the root of the network. As each node hears the capsules, it decides if the Maté capsules should be installed locally or need to be flooded to its children or both. Our queries successfully run on this environment. We have night Tmotesky nodes equipped with M1-Mini Reader in our implementation. However, this implementation is too small in its scale to measure the performances of our algorithms. Consequently, the experiment results in this section are based on *J-sim simulator* [9], which allows us to measure the performance of our techniques in a scale much larger than our current system.

5.2 Experiment Setup

In J-sim, a sensor network consists of three types of nodes: *sensor nodes* (detect events), *target nodes* (generate events), and *sink nodes* (utilize and consume the senor information). In our setting, we create a monitoring area in which a regular 13*14 grid with 36 sensor nodes is simulated. The sensor nodes are assumed to have a 2*2 grid of monitor radius. The *sink node* is on the left-upper corner of the grid. Figure 8 shows this setting. In addition, we generate 6000 target nodes that simulate customers with RFID tag. The target nodes are assumed to enter the area from the entrance and leave from the exits of the area. We use *random walk* to simulate the customers' moving behavior: the target nodes on the area move to every possible direction with the equal probability, and stop

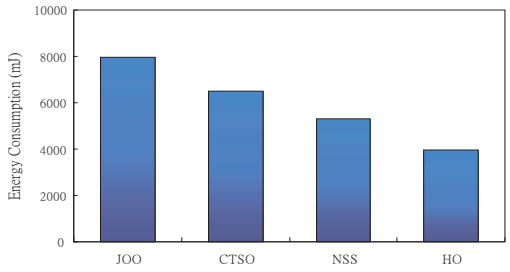


Figure 9 Performance comparison

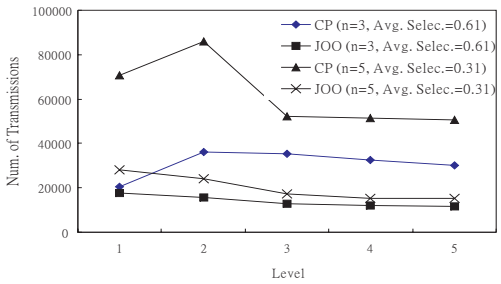


Figure 10 Effect of in-network processing

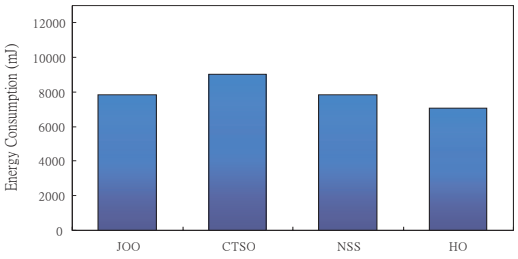


Figure 11 Effect of low sharing opportunity

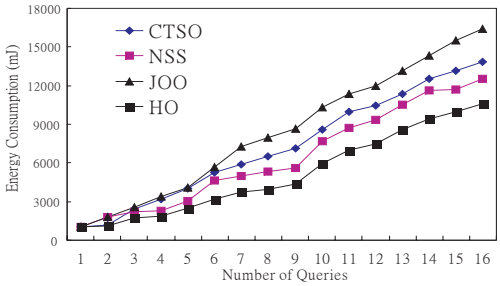


Figure 12 Effect of number of queries

its move process when it reaches the exits. When a target node enters the monitor region of a sensor node, the sensor node generates a tuple (5bytes) containing the sensor ID (1byte), the target node ID (3bytes), and the timestamp (1byte). As mentioned earlier, if target nodes keep entering the area, the sensor nodes generate data streams. We process the streams by the proposed techniques, and compare the performance of the proposed techniques in this environment. In all experiments we show the average value of 100 runs.

5.3 Evaluations

5.3.1 Basic Comparison

In the first set of experiments, we evaluate the performance of the presented optimization techniques.

- Join-order optimizations (JOO): all queries are individually optimized by join-order optimizations.
- Common-task-sharing optimizations (CTSO): the queries are optimized by only considering share common tasks among queries. The implementation is given in Appendix A.2.
- Naïve sharing strategy (NSS): all the queries are firstly optimized by JOO and then share the common prefix among queries.
- Hybrid Optimization (HO): the queries are optimized by the HO optimization with parameter $\varepsilon = 2$.

We pose eight queries on the simulated environment, as shown in Figure 8. The average selectivities for the queries are 0.2104. Figure 9 shows the experiment results, where y-axis shows the energy consumption for processing the given queries. We can see that HO significantly outperforms the other approaches, while JOO performs poorly as JOO does not consider exploiting sharing common tasks.

5.3.2 Effect of In-network Processing

This experiment demonstrates the benefits of in-network processing against the centralized processing (all data are collected and evaluated in a BS). We mainly compare the centralized processing (CP) with JOO (as a representative of in-network processing). Two query sets are used to evaluate the performance, one with high selectivities (average selectivity = 0.6115) of joining three streams (n=3) and the other with lower selectivities of join five streams (0.3116, n=5). We vary the level of nodes at which the execution plans are installed to observe the performance of the strategies. Figure 10 shows the results, where x-axis is the level of nodes (the sink node at level 0) and y-axis is the number of transmissions involved in the query processing.

We can see that CP performs poorly. There are two reasons for the poor performance. First, while JOO only requires transmitting the tuples of the node at which the execution plans are installed, CP requires all streams involved in the queries to transmit all their data to the BS. This negative effect becomes huge as the number of involved streams increases. Second, CP transmits all data to BS regardless of whether the data contribute to the query results, while JOO invokes

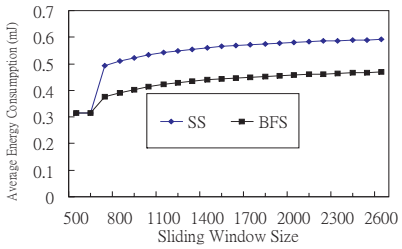


Figure 14 Effect of sliding window size

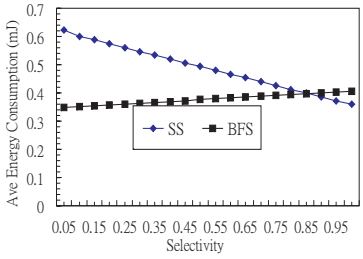


Figure 15 Effect of join selectivity

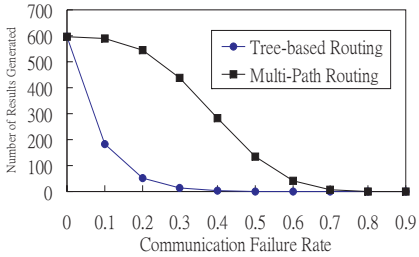


Figure 16 Effect of routing scheme

communications only when a tuple possibly contribute to the results. This effect can be observed from the selectivity factor. When the selectivity is 0.3116, JOO obviously outperforms CP, because most of tuples get dropped during execution. This experiment suggests that CP is only good for the query processing that is close to the sink, with very high selectivity, and few streams involved in queries.

5.3.3 Effect of Low Sharing Opportunity

This experiment considers the effect of low sharing opportunity. We consider a query set where each query only has one common task with others. Even under this restriction, our approach still has positive results, as shown in Figure 11. This is because if no common task exists, HO will degenerate to JOO, i.e., all queries will be optimized individually. One thing to note in this experiment is that JOO outperforms CTSO. As we mentioned, blindly sharing the common tasks is not always beneficial to the query processing. In this experiment with low sharing opportunity, the poor performance of CTSO comes from the fact that the tasks saved by sharing are less than the cost incurred from the violation of the individual optimizations.

5.3.4 Effect of Number of Queries

This experiment considers the effect of the number of queries involved in the processing. Figure 12 shows our results, where x-axis is the number of the queries involved in the processing and y-axis is the energy consumption. We vary the number of queries to observe the performance of the techniques. We can see that HO always outperforms the other approaches and shows obvious advantages as the number of queries increases, since when the number of queries increases, the sharing opportunities also increase.

5.3.5 Effect of Memory Constraints

This experiment evaluates the efficiency of the strategies proposed in Section 4. The strategies deal with the memory limitation problem for query processing. We compare the BF strategy (BFS) with the straightforward solution (SS). A query that joins the streams formed by the left-lower node and right-lower node in Figure 9 is issued. Assume that, in each node only three kilobytes memory is available for the query processing. In BFS, each node utilizes its memory to maintain a hash table with 564 tuples and a BF with 1440 bits, while in SS, the node simply keeps a hash table with 600 tuples. We vary the sliding window size of the query to observe the performance of the strategies. Figure 14 shows the results, where the x-axis is the sliding window size and the y-axis is the average energy consumption for processing a tuple. As expected, BFS outperforms SS when the memory is out of use (when window size >600), since BFS only transmits the tuples that may contribute to the join result to the BS.

We also vary the join selectivity (the probability that a tuple joins the other stream) to observe the performance of the strategies. Figure 15 shows the performance of BFS and SS (with a window of 700 tuples) for different selectivities. We can see that BFS is efficient for low selectivities, since BFS only transmits the tuples that may generate a join result at the BS. When the selectivity is low, only a small potion of tuples contributes to the join result. As a result, BFS suppresses most of the communications with the BS to reduce the energy consumption. However, when the selectivity grows to the range [0.8,

1.0], SS outperforms BFS. The reason is that most of the tuples contribute to the join result and therefore have to be transmitted to the BS. In this case, the BF becomes useless. Even worse, the ineffectual BF occupies a portion of the limited memory and decreases the probability that a tuple gets matched in local memory.

5.3.6 Effect of Routing Scheme

This experiment verifies the effectiveness of exploiting the multi-path routing scheme to handle the communication failures. We vary the communication failure rate to observe the effectiveness of the multi-path routing scheme. Figure 16 shows the results, where x-axis is the failure rate and y-axis the number of query results produced. We can see that the multi-path routing scheme shows high failure resilience when compared with a tree-based routing scheme.

6. Related Work

The problem of processing multiple queries in a data stream environment has been studied in several fronts [10][12][13][14][15][16].

Hammad, et al.[10] study the problem of sharing executions of queries that that have different join window specification, where the proposed techniques are only capable of handling two-way join queries. The approach proposed in [12] address the problem of optimizing multiple group-by queries in GigaScope [11], where the queries can only differ in the grouping attributes.

In addition to sharing query plans, another alternative for processing multiple queries is to use the query predicate index. The main idea for *query predicate index* [13][14][15] is to decompose queries into operators and use an index structure to simultaneously evaluate multiple queries. One concern for these approaches is the intuition that the sharing among queries always benefits the overall query processing. However, this intuition does not always hold. In order to induce sharing, the some executions, such as selection and projection, may need to be postponed, which can result in a significant increase in the size of intermediate results. Moreover, if the operators are with high selectivity, executing the queries individually can be more beneficial to the overall cost. [13][14] proposes Psoup operator that treats queries and data as duals, and apply a symmetric hash join between query table and data table.

Krishnamurthy, et al.[15] address sharing executions of the multiple aggregates with different window specification and different selection predicates over single data stream. The idea is to fragment input tuples into disjoint sets with respect to the query predicates and then on-the-fly answer queries with the associated disjoint sets.

All the above-mentioned works [10][12][13][14] [15] use the same framework where massive data streams are collected and sent to a central processing engine where the data are processed. As reported in our experiments, such frameworks are not efficient for processing the streams formed by a sensor network due to power considerations.

Huebsch, et al. [16] extends the idea of fragmenting tuples[15] to processing multiple aggregation-queries over distributed data streams, where the queries are assumed to differ in their selection predicates. Both [15][16] focus on the aggregate queries, but none of them permit join processing.

The in-network query processing systems for sensor networks, including Tinydb [19], TAG [18], and Cougar [17], support simultaneous multiple query executions. However, none of them addresses the problem of optimizing multiple join-queries.

The recent works [20][21][22] propose multiple query optimization techniques for sensor networks. However, the proposed techniques all focus on optimizing multiple aggregation or selection queries, which are orthogonal to the problem we solve. Müller et al. [21] considers the problem of optimizing multiple aggregation queries for sensor networks, while the work [20][22] addresses the problem of merging multiple selection queries which request different data at different acquisition rates.

7. Conclusion and Future Work

In this paper, we investigate the problem of efficiently evaluating join-queries in a sensor network. In addition to individual query optimization, we consider exploiting multi-query optimizations. We formulate the problem of the multi-query optimization and show the complexity of finding the optimal solution. We propose an efficient algorithm, which provide solutions with sub-optimal guarantees. The experiment result shows that our techniques ensure scalability, minimize the message traffic, and therefore reduce energy consumption. In this study, we consider scalability issues of processing path-tracking queries. However, in stream environments, the adaptivity issues are also critical when dealing with the dynamic nature of data streams. If the data distribution of streams fluctuates over time, an adaptive approach to execute the queries is therefore essential for a good performance. We are currently implementing the path tracking application in a large sensor network. This raises some practical problems at the system level,

such as communication scheduling, and etc. We plan to explore these issues as a part of our future work.

8. References

[1] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 285-296, 2003.

[2] S. Babu, and et al. Adaptive ordering of pipeline stream filters. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 407-418, 2004.

[3] E. Balas and M. Padberg. Set partition: a survey. In *SIAM review* (18), pages 710-760, 1976.

[4] J. Han and M. Kamber. Data Mining Concepts and Techniques. Morgan Kaufmann Publishers, 2001.

[5] R. Motwani and P. Raghavan. Randomized Algorithms. Cambridge Press, 1995

[6] Moteiv Inc. *Tmote Sky Brochure and Datasheet.*

[7] SkyeTek Inc. *SkyeRead M1 Mini User Manual.*

[8] P. Levis and D. Gay. Maté: a tiny virtual machine for sensor networks. In *Proc. of Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85-95, 2002.

[9] A. Sobeih and et al. J-sim: a simulation and emulation environment for wireless sensor networks. *IEEE Wireless Communication*, vol. 13, issue 4, pages 104-119, 2006.

[10] M. A. Hammad, and et al. Scheduling for shared window joins over data streams. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 297-308, 2003.

[11] C. D. Cranor, and et al. Gigascope: a stream database for network application. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 647-651, 2003.

[12] D. Srivastava, N. Koudas, R. Zhang, and B. C. Ooi. Multiple aggregations over data streams. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 299-310, 2005.

[13] S. Chandrasekaran and and M.J. Franklin. Streaming Queries over Streaming Data. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 203-214, 2002.

[14] S. Madden, and et al. Continuously Adaptive Continuous Queries over Streams. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 49-60, 2002.

[15] S. Krishnamurthy, and et al. On-the-fly sharing for streamed aggregation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 623-634, 2006.

[16] R. Huebsch, and et al. Sharing aggregate computation for distributed queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, 2007.

[17] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proc. of Intl.Conf. on Innovative Data Systems Research*, 2003.

[18] S. Madden and et al. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. of Annual Symps. on Operating System Design and Implementation*, pages 131-146, 2002.

[19] S. Madden and et al. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. on Database Systems*, vol. 30, issue1, pages 122-173, 2005.

[20] N. Trigoni, and et al. Multi-query optimization for sensor networks. In *Proc. of Intl. Conf. on Distributed Computing in Sensor Systems*, pages 301-321, 2005.

[21] R. Müller and G. Alonso. Efficient sharing of sensor networks. In *Proc. of Intl. Conf. on Mobile Ad-hoc and Sensor Systems,* 2005.

[22] S. Xian, and et al. Two-Tier Multiple query optimization for sensor networks. In *Proc. of the IEEE Intl. Conf. Distributed Computing System.* Pages 39, 2007.

[23] Y.C. Fan and A. L.P. Chen. Optimizing multiple join queries over sensor data streams. Technical Report. http://make.cs.nthu.edu.tw. 2006.

Appendix

A.1 Proof of Theorem 1

Theorem 1*: Given a set of sub-queries* $W = \{Q_1, Q_2, ..., Q_m\}$*, the problem of selecting one join order for each query in W with the goal of minimizing total execution cost is NP-Complete.*

Idea**:** We refer to the problem as MJQO. By showing the equivalence between the classical NP-Complete problem SET PARTITION and the MJQO problem, we conclude that there is no polynomial time algorithm expected to exist.

Set Partition: Given a universe $U = \{e_1,..., e_m\}$ of m elements, a collection of subsets of U, $S = \{S_1,..., S_r\}$, and a cost function $f : S \rightarrow R^+$, find a minimum cost collection of S that partition all elements of U, R is the set of real number.

Proof: The MJQO is NP-Complete if it satisfies two conditions: (1) the MJQO is in NP, and (2) every problem in NP is polynomial time reducible to MJQO. It is easy to see that the MJQO have a polynomial time verifiers. Namely, the MJQO is in NP. Therefore, we only need to prove that all problems in NP reduce to the MJQO in polynomial time. We do so is by showing the set partition problem polynomial time reduce to the MJQO problem. An instance of the set partition can

be transformed into an instance of the MJQO problem as follows.

Transformation: MJQO→ Set Partition
1. Create as many elements as the queries in W: $U = \{e_1,..., e_m\}$ 2. For each possible execution t_k, If the plan of Q_i is part of t_k, Create a set S_k containing e_i 3. For each possible execution t_k, Create a cost $f(t_k) = \sum cost(Q_i)$ if Q_i is a part of t_k.

It is easy to see that any feasible solution of the created instance of the MJQO problem corresponds to a feasible solution for the original instance of the set partition problem. On the other hand, we can reduce MJQO to set partition by the following transformation.

Transformation: Set Partition → MJQO
1. Create as many queries as elements in U: $W = \{Q_1, Q_2, ..., Q_m\}$ 2. For each set $S_k \in S$, create an execution t_k with cost $c_k = f(S_j)$. 3. If e_i is in S_k Then create for Q_i a plan which is a part of t_k.

As can be seen, the transformations can be achieved in polynomial time. Namely, the set partition problem polynomial time reduce to the MJQO problem. As a result, the MJQO can be concluded as NP-Complete.

A.2 Common-Task-Sharing Optimization

Given a set of queries $\{Q_1, Q_2, ..., Q_m\}$ to be installed at node N_j. The first step for common-task-sharing optimizations is to identify the task-set with $count \geqq 2$ from $\{\theta_1, ..., \theta_m\}$.

The common-task-sharing optimizations are based on *maximum-potential-benefit* (MPB) value. The MPB value for a common task-set is defined by the size of the common task-set multiplied by the count of the task-set minus one. The MPB value for a common task-set represents the maximum potential benefit we can gain from sharing the common task-set. The maximum benefit comes up when the selectivity of joining ΔS_j with the common task-set equals to one.

Once the common task-sets are identified, our algorithm proceeds as follows. First, compute the MPB value for each common task-set. Second, choose the common task-set with the maximum value. Third, plan the associated queries such that the common task-set will be executed first, and merge the common prefixes between the query plans. Finally, delete all other common task-sets involved with all the queries in third step. If no common task-set remains, terminate the planning process. Otherwise, repeat the above steps.