Collaboratively Querying Sensor Networks through Handheld Devices

TszWai Chiu Qiong Luo
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{cswai, luo}@cse.ust.hk

Abstract

We envision that in some wireless sensor network applications, such as environmental monitoring, assisted living, and industrial control, handheld devices will be used from time to time to query the sensor networks. However, there is no full-fledged query processor for this purpose. Therefore, we propose WinyDB, a relational query processing system on Windows-CE based PDAs (Personal Digital Assistants) for sensor networks. One of the main features of WinyDB is that multiple PDAs running WinyDB can answer queries collaboratively. This feature is useful in that it improves both the power efficiency and the data quality. Our WinyDB prototype package is available online at http://www.cse.ust.hk/winydb and our simulation experiments have shown promising results on collaborative query processing.

1. Introduction

Wireless sensor networks (WSNs) have been widely deployed in applications such as habitat monitoring, medical care, intelligent building, and surveillance [2][6][11][13][23]. A common way to acquire sensory data from these WSNs is to issue a query from a PC-grade base station to the WSNs and have the sensors return their query results to the base station [19][35]. There has also been some work [5][23][27] on using handheld devices to collect WSN data, but the query processing capabilities of these data collection tools are limited. In this paper, we propose WinyDB, an embedded query processing system on Windows-CE based PDAs (Personal Digital Assistants) to query WSNs.

We envision that WinyDB will be useful for many WSN applications. For example, in a country park with WSNs deployed in the natural environment, Wi-Fi covered in the air, and WinyDB-installed PDAs provided to the park rangers and the visitors, people can issue queries about the environment through their PDAs as they move around. Furthermore, if multiple PDAs are roaming around in the park, they can also help one another answering queries about their local WSNs. These queries can be answered conveniently and promptly without contacting a central server. Most importantly, this collaborative query answering will save the power consumption of WSNs as well as improve the data quality, because the data acquisition is done locally and is shared among multiple queries.

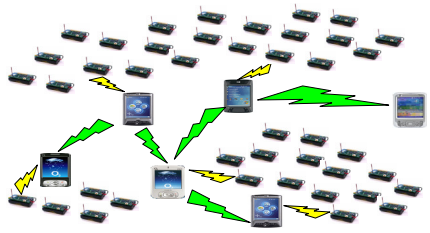

Figure 1. Mobile devices collaboratively querying WSNs.

Figure 1 illustrates mobile devices collaboratively querying WSNs. Now that mobiles devices become base stations for their local WSNs, we need to consider the resource limitations of the devices as well as the WSNs in collaborative query processing. Specifically, we address the following questions:

1. What query processing capabilities do we support on PDAs? Current leading WSN query processing systems for PC-grade base stations, such as TinyDB [31], support selection, projection, and aggregation. Given multiple WSNs deployed in a wide area, how do we answer a query involving multiple streams of sensory data?
2. Given queries of different kinds running on multiple PDAs for multiple WSNs, what collaborative query processing schemes do we design to answer these queries? Traditional distributed database systems have data shipping and query shipping techniques [29]. Recently there has been work on stream processing [10][17][37]. How are these techniques applicable to WinyDB?
3. Finally, given the resource constraints of WSNs and PDAs, what cost model do we use to capture these constraints and to direct WinyDB to process queries efficiently?

The reminder of this paper is organized as follows. Section 2 gives an overview of WinyDB and Section 3

presents our collaborative query processing framework. We discuss experimental results in Section 4, review the related work in Section 5, and conclude in Section 6.

2. System Overview

We consider multiple WSNs deployed in a wide area, e.g., in a park. Each WSN has a unique network ID and a sink node. A PDA may roam into any of the WSNs and can communicate with the sink node of the WSN if the sink node is within the PDA's communication range. Both PDA-WSN and PDA-PDA communications are via IEEE 802.11b.

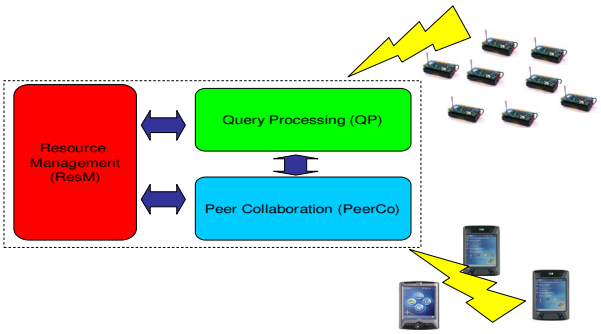

Figure 2. Architecture of WinyDB

Figure 2 shows the three software components of WinyDB - *query processing (QP)*, *peer collaboration (PeerCo)*, and *resource management (ResM)*. The QP module is in charge of query parsing, query optimization, and query result processing. Because PDAs move around, and queries are issued and removed dynamically, the PeerCo module on each PDA keeps track of the current status of its peer PDAs and coordinates its collaboration with these devices. Finally, the ResM module monitors the consumption of the resources, including WSN power and PDA memory.

When a PDA issues a query, the query is parsed and optimized by the QP module at this *query issuer*. During the query optimization, QP consults PeerCo to check whether any other PDAs (we call them *data providers*) can collaborate to answer part of or the whole query. Then it generates candidate query plans, works with ResM to estimate the goodness of each plan, and selects the best plan. This best plan may be sending the query to the WSNs directly or sharing the query results from some other PDAs. Finally, QP works with PeerCo to execute the query and to return query results to the user. Since most of the queries are continuous and the environment is dynamic, QP may change a query plan to adapt to the current setting.

3. Collaborative Query Processing

Our database model is similar to those in Cougar [35] and TinyDB [19], where a sensor network is represented as a virtual table and every sensor node generates a data tuple for each sample period. The difference is that in our work, this virtual table contains multiple physical WSNs, each of which is identified by a network ID. In addition to selections, projections, and aggregations supported by the previous work, we handle window join queries. All queries can either be a snapshot query that returns results only once or a continuous query that returns results every sample period. We focus on continuous queries, because they create more opportunities for as well as benefit more from collaborative query processing.

The goals of our collaborative query processing are to improve data quality and to reduce WSN power consumption. These two goals may be conflicting; when this happens, we take a user-centric stand and put data quality before resource efficiency. We measure the data quality of a given query in the query result output rate, because it reflects both the time efficiency in sensor stream processing and the success rate of data delivery. We estimate WSN power consumption by the amount of data communication. Additionally, because a PDA is resource constrained, we take its available memory into consideration.

In the following, we present our collaborative query processing schemes for different types of queries.

3.1. Selection, projection queries

We consider selection and projection queries with numeric comparison predicates. Note that, even though a query appears as a single table query, it may involve multiple physical WSNs unless the network ID of a WSN is specified in the selection condition. Query 1 is such an example.

Query 1:
SELECT nodeid, temperature
FROM sensors
WHERE NetworkID=1 AND temperature>30
SAMPLE PERIOD 60s

If the query issuer is out of the range of the sink of WSN 1, it must ask for the help of the PDAs that are in the range of WSN 1, which we call the *candidate data providers* for WSN 1; otherwise, the query issuer itself is also a candidate data provider. A candidate data provider for a WSN can answer a query either by (1) directly issuing the query to the WSN or by (2) evaluating the query on the cached results of another query. We call option (1) *direct evaluation* and (2) *cache evaluation*. We always choose cache evaluation over direct evaluation whenever applicable, because it can reduce the WSN workload and therefore improve the data quality and save the WSN power consumption.

Cache evaluation of query A over the result of query B is applicable when query B *contains* query A [32]. Given selection and projection queries A and B in their conjunctive normal forms (CNFs), we determine that B contains A when (1) B's list of attributes is a superset of A's, (2) each of B's selection predicate conjunct is less restrictive than one of A's, and (3) if A is a

continuous query, its sample period is a multiple of B's. For example, *temperature > 30* is less restrictive than *temperature > 40* or *temperature = 50*.

When multiple candidate data providers can answer a query by cache evaluation, we choose the one with the highest estimated result output rate. A higher output rate is preferred by the user and usually indicates the data provider has a lighter workload and/or a better capability.

In WinyDB, each candidate data provider provides its estimated result output rate for a given query. This estimation is done by examining the *input rate*, the *processing rate*, the *transmission rate*, the *buffer size*, and the *selectivity*. The input rate is the number of input tuples from a WSN or from another data provider per unit of time. The processing rate is the number of result tuples processed per unit of time. The transmission rate is the number of result tuples that can be transmitted to the query issuer per unit of time. The buffer size is the maximum number of result tuples to be cached. The selectivity is the number of final result tuples divided by the number of input tuples.

Based on the input rate (in_t), selectivity (sel_t), processing rate (p_t), buffer size (buf_t), and the transmission rate (x_t) at time t, the output rate (out_t) during a sample period [t_1, t_2] can be estimated in one of the following four cases:

Table 1. Output Rate Estimation

Cases	out_t
$in_t \le p_t$ $sel_t * in_t < x_t$	$\min(sel_t * in_t, x_t)$
$in_t \le p_t$ $sel_t * in_t \ge x_t$ $\int_{t_1}^{t_2} (sel_t * in_t - x_t)dt \le buf_{t_1}$	$\min(sel_t * in_t, x_t)$
$in_t > p_t$ $sel_t * p_t \le x_t$ $\int_{t_1}^{t_2} (in_t - p_t)dt \le buf_{t_1}$	$sel_t * p_t$
$in_t > p_t$ $sel_t * p_t > x_t$ $\int_{t_1}^{t_2} (in_t - p_t + sel_t * p_t - x_t)dt \le buf_{t_1}$	x_t

Only for the four cases in Table 1 will we consider cache evaluation on the data provider. Otherwise, we will exclude this data provider from performing cache evaluation because its buffer will overflow and it will lose query results. As the output rate can be changing over time, the query issuer keeps track of the output rates for candidate data providers periodically. If the output rate of the current data provider becomes slow compared with other candidates for a period of time, the query issuer will switch to collaborate with the data provider having the highest output rate.

If a data provider has no cache evaluation alternatives for a given query, it will estimate its output rate for direct evaluation of the query over the WSN. In this case, we estimate its output rate as follows:

$$out_t = \min(p_t, x_t)$$

Finally, we handle the case of using multiple data providers to answer a given query. In this case, each of the data providers can answer part of the query through cache evaluation or direct evaluation. Therefore, the query issuer generates alternative plans, estimates the output rates of each alternative, and selects the final query plan.

3.2. Aggregation Queries

We support seven common aggregate functions, including MAX, MIN, AVG, SUM, COUNT, COUNT DISTINCT, and MEDIAN. We classify these aggregate functions into two categories, *divisible* and *indivisible*. Aggregate functions MAX, MIN, COUNT, and SUM are divisible, because they can be evaluated on partitions of the result set and the final aggregation value can be obtained by combining the partial aggregates of the partitions. In contrast, aggregate functions COUNT, DISTINCT, and MEDIAN are indivisible, because partial aggregation is inapplicable to them. Finally, AVG itself is indivisible, but we treat it as divisible by representing it in the form of (SUM, COUNT) so that we can apply partial aggregation.

We handle an aggregation query in a way similar to selection and projection queries. Specifically, it can be done either through direct evaluation or cache evaluation. The cache evaluation can use the results of existing selection and projection queries. Furthermore, a query with a divisible aggregate function can be evaluated at a candidate data provider and the final aggregation is performed at the query issuer by combining the partial aggregation values.

Query 2:
SELECT MAX(temperature)
FROM sensors
WHERE NetworkID=1 OR NetworkID=2
SAMPLE PERIOD 60s

For instance, if the query issuer issues Query 2, it has to query the temperature readings of both the WSNs 1 and 2. Suppose data providers A and B are connected to WSNs 1 and 2 correspondingly, the query issuer can split Query 2 into two sub-queries by NetworkID and forward each sub-query to the corresponding data provider. The final maximum temperature reading of the WSNs 1 and 2 is then evaluated at the query issuer based on the local

maximum readings collected from data providers A and B.

However, for a query with an indivisible aggregate function, we can apply the aggregate function only at the query issuer, not at any other data provider.

Query 3:
SELECT Median (temperature)
FROM sensors
WHERE NetworkID=1 OR NetworkID=2
SAMPLE PERIOD 60s

Suppose data providers A and B are connected to WSNs 1 and 2 respectively, the query issuer of Query 3 can still split the query into two sub-queries by NetworkID for A and B correspondingly. However, the aggregate function, Median, must be removed from the sub-queries and is applied only at the query issuer after getting the query results from A and B.

3.3. Window Join Queries

A window join can be between different WSNs or within one WSN. The window is time-based. We consider only binary joins and the join predicate can use any of the numeric comparison operators <, <=, =, ≠, >=, or >.

For simplicity, we apply window join predicates only at PDAs, not at individual sensor nodes. This design decision is advantageous in that join is a complex and expensive operation and therefore is more suitable to perform at PDAs than at individual sensor nodes. Moreover, if window joins were pushed into WSNs, data flows would become extremely complex, which would further burden the resource-constrained and lossy WSNs.

However, the downside of the PDA-side join approach is that many input tuples may be useless when the join is highly selective. Furthermore, even though PDAs are more resource abundant than sensor nodes, their memory size is still quite limited. Therefore, we have developed a synopsis based prediction method for two input sources of an equi-join to decide on whether to discard an input tuple before performing the join. This synopsis works for one or multiple join attributes, with one synopsis per join attribute.

3.3.1. Join Synopsis

We build a join synopsis at the two data providers of a join to predict if an input tuple may be discarded or not for the join. These two data providers can be the same PDA or different ones. The query issuer may be one of them or a third PDA. This join synopsis works in two phases:

1) Start-up Phase

Suppose the two data providers are A and B. Each of the data providers determines the range of its join attribute values from its local WSN. Then, they send the ranges to the query issuer. The query issuer thereby determines a common range and a number of equal-width partitions, *k*. This common range covers both A and B's ranges, and the choice of k is dependent on the free memory space of the query issuer. A larger k incurs higher memory usage but leads to further reduction of communication and improvement of output rate.

Next, both A and B start to construct their join synopses. A join synopsis is a bit vector summarizing the distribution of the join attribute value. Each bit in a k-bit vector corresponds to one of the k partitions of the join attribute value over the common range. When the join attribute value falls into a certain partition, the corresponding bit is set to 1; otherwise, the bit remains to be 0. This bit vector applies to the tuples within the current join window.

For each window join, A and B exchange their bit vectors through the query issuer and only send the tuples that fall into the common partitions to the query issuer. Additionally, these bit vectors are used to determine what tuples to discard when the memory size is limited.

2) Maintenance Phase

As the sensor data values may change over time, the bit vectors and the common range of data values may change too. Therefore, we update the bit vector for each sample period. If it differs from the last sample period, it is sent to the query issuer. To prevent the common range and the number of partitions, k, from changing frequently, we expand the range to a certain degree so that it usually holds for the full length of a join window. In our experiments with the Intel Berkeley dataset [16], it is sufficient to expand this common range by 10%.

However, many data values may fall into one partition when the number of partitions is small or the data distribution is skewed. When this happens, few tuples may be discarded according to the bit vectors. Therefore, we maintain join statistics at the query issuer. Specifically, for each partition, the query issuer records two counters - the number of tuples received from the data provider and the number of tuples that satisfy the join condition. With these two counters, we know which partitions contain many tuples but only a few of them satisfy the join condition. The data provider then further divides up such a partition by constructing a child bit vector on the partition. Similarly, if the two counter values associated with a partition are similar, the data provider may decide to remove its child partitions, if any.

3.3.2. Collaborative Join Processing

Query 4 is an example window join query.

Query 4:
SELECT A.nodeid, A.light, B.nodeid
FROM sensors as A, sensors as B
WHERE A.NetworkID=1 AND B.NetworkID=2
AND A.light=B.light
SAMPLE PERIOD 60s

WINDOW SIZE 5min

Similar to selection, projection and aggregation processing, we prefer cache evaluation over direct evaluation for join processing in order to save WSN power efficiency and to improve data quality. Cache evaluation can be done on existing selection, projection, or join queries.

Given a new window join query Q on two WSNs, its collaborative processing is either (1) cache evaluation on another PDA or (2) cache evaluation on two other PDAs.

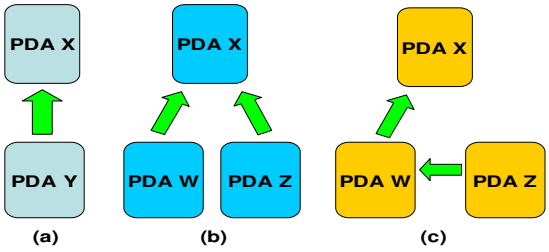

Figure 3. Cache Evaluation on other PDAs

Figure 3(a) illustrates the cache evaluation on another PDA. PDA X is the query issuer and PDA Y is the data provider. If there is an existing join query on Y that contains the new query, all processing can be done on PDA Y and only the final results are forwarded to PDA X. If there are only selection or projection queries on Y to be used for cache evaluation, the join processing can be done either on Y or on X. The choice of plan depends on the result output rate.

If the join query involves two other PDAs each of which connecting to a WSN, there are two possible communication schemes – either merging (Figure 3(b)) or pipelining (Figure 3(c)).

When a join query involves more than one data providers, the coordination of data providers is essential. The simplest way is to use merging as shown in Figure 3(b). All data are forwarded to and joined at the query issuer. However, this approach may not be cost-effective even with our join synopsis method applied, because data communication may be reduced if they are joined between the data providers before forwarded to the query issuer, as shown in the pipelining in Figure 3(c). In addition, the memory and processing requirement of the query issuer can be shared by the data providers in pipelining.

To use the pipelining approach, we need to address three main questions. First, which data providers do we choose for the join? Second, what is the ordering of the two data providers in joining their query results? The third question is how to adaptively change the communication scheme.

The first question is equivalent to finding out a pair of data providers that can reduce the data communication by joining their results. Specifically, these data providers must satisfy two conditions:

1. They return the join attributes.
2. The join between the two reduces the amount of data sent to the query issuer, compared with the merging approach.

Condition 1 can be determined by examining the join query. Condition 2 is checked in Formula (1).

$$N(R_i)+ N(R_j)> N(R_i)+N(R_i \rhd\lhd_c R_j) \qquad (1)$$

R_i is data provided by data provider i. $N(R_i)$ is the size of R_i.

The second question – the place of join between the pair of data providers – is determined by Formula (2). If Formula (2) holds, the join will be at data provider j; otherwise, it will be at data provider i.

$$N(R_i)>N(R_j) \qquad (2)$$

Finally, the communication scheme may switch between the two dynamically. In WinyDB, merging is the default communication scheme for using two other PDAs for join processing. The pipelining approach is adopted only when at least a pair of data providers satisfies Conditions 1 and 2 with a confidence level. To avoid oscillating between the two schemes, we set a threshold 80% for this confidence level. We use the percentage of the sample periods that Condition 2 was satisfied over all past periods as the confidence level. Only when this confidence level is above the threshold value, will the query issuer switch from merging to pipelining.

4. Evaluation

We have implemented WinyDB in C# using Visual Studio .NET 2003 and .NET Compact Framework 1.0. We have tested WinyDB on several models of Windows CE based PDAs to query simulated sensor networks. The sensor networks we simulated consist of tens of MICA2 motes [9]. We call this version the optimized WinyDB.

For comparison, we have ported TinyDB [31] to run on PDAs and have extended it to process window joins. We call it the original WinyDB. The routing protocol, time synchronization, and task management in this version for sensor nodes are the same as that in TinyDB. The query issuer, however, has no query sharing with the other PDAs. As a result, all queries are executed separately in sensor networks and the join operation is done at the query issuer.

Since we focus on continuous queries for sensor data streams, we select the result output rate as the end-to-end performance metric. This metric reflects the network traffic in the WSNs and the packet loss rate, or more generally, network efficiency and data quality. Also, as power consumption is the critical constraint for WSNs, we select the average sensor node power consumption as the other performance metric.

4.1. Experimental Setup

To evaluate WinyDB in a variety of settings, we ran WinyDB on three PDAs to query simulated WSNs. Two of the PDAs each has a 624 MHz processor and 64 MB memory, running Windows mobile 2003. The third PDA has a 400 MHz processor and 64 MB memory, running Windows mobile 2002.

We used TOSSIM [18] to simulate WSNs of different configurations. Each simulated WSN consists of 10, 30, or 50 nodes. These nodes are randomly deployed into a grid. The transmission range of each node is 50 feet. As TOSSIM simulates a lossy environment, most of the packets transmitted from a node more than 20 feet away will be lost. Therefore, we set the size of each grid cell to be 10 feet by 10 feet. These network configurations are comparable to those in the previous studies [21][36]. The sensor readings at each node are simulated using the real data from the Intel Lab data set [16].

In every experiment, each of the three PDAs connects to a WSN so that any queries about any of the three WSNs can be answered, with or without query sharing. All queries are continuous queries with sample periods varies from 8 to 64 seconds. The number of attributes returned in each query is between 1 and 4, both inclusive. Each attribute that appears in the query condition involves at most one selection predicate. The presence of selection condition, aggregation function, and join condition is all randomly generated. The window size of join queries varies from 40 to 120 seconds. The number of concurrent queries on each PDA is varied from 2 to 4. This small number is due to the limitations in TOSSIM; running a large number of queries concurrently overwhelms TOSSIM.

In addition to the experiments on all query types, we conducted another set of experiments to evaluate the effect of memory size on join queries. In these join query experiments, one PDA issues a window join query with a fixed sample period of 8 seconds, whereas the other two PDAs issue queries of all types by random. The available memory is varied from 25% to 100% of the join window size.

To estimate the power consumption of multiple queries running in the WSNs, we logged in each sensor node the time spent in different operation modes, including sensing, processing, listening, receiving, transmitting, and sleeping. We then use the electric current level of each mode from the specification of mica2 hardware platform [9] to calculate the power consumption.

Table 2 summarizes the simulation parameters of our experiments.

Table 2. Simulation parameters

Parameters	Values (Default*)
# nodes	10, 30*, 50
sample period (seconds)	8*, 16, 32, 64
window size (seconds)	40*, 80, 120
# queries issued per PDA	2*, 3, 4
available memory (% of window size)	25%, 50%, 75%, 100%*

4.2. Simulation Results

4.2.1. Effect of the number of queries

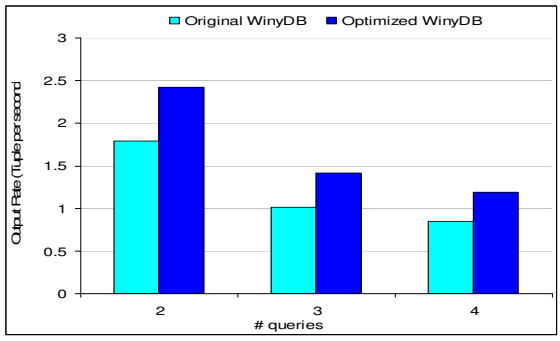

Figure 4. Output Rate

Figure 4 shows the effect of number of concurrent queries per PDA on the output rate. In the figure, the output rate is negatively affected by the number of queries. This phenomenon is expected because the query workload determines the network traffic and the response time, with all other factors fixed. Nevertheless, the optimized WinyDB outperformed the original WinyDB for all numbers of concurrent queries per node. The main reason is that, with query sharing, when a query can be answered by the results of some other queries, this query is not sent to the WSNs. Consequently, the workload of the WSNs is reduced. Additionally, the workload for a PDA to process the raw results from the WSNs directly is heavier than to process the intermediate results from other PDAs.

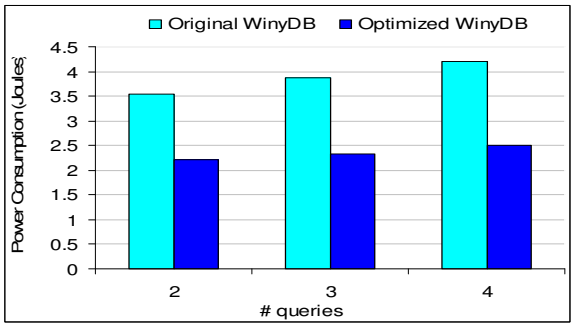

Figure 5. Power Consumption

With the same configuration as for Figure 4, Figure 5 shows the average node power consumption for one minute. The optimized WinyDB consumes much less power than the original one, and its power consumption increases much slower with the increase of the workload. Similarly, the experimental results for other sizes of WSNs show that the optimized WinyDB can

always improve the output rate by 30% to 40%, and the percentage of power saving is about 40% on average.

4.2.2. Effect of the memory size

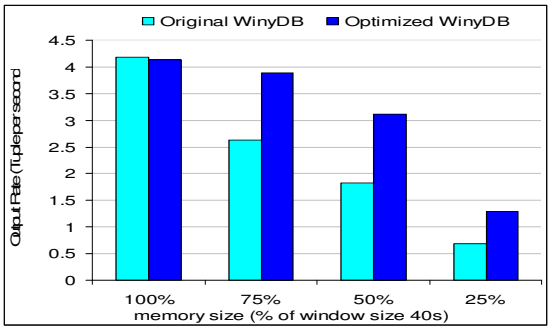

Figure 6. Output Rate

For handheld devices, the memory, especially the RAM, is very limited compared with a PC. This memory constraint directly affects how much query results within the sliding window of a join query can be cached. Therefore, we vary the memory size with respect to the sliding window size and evaluate the join query performance.

In Figure 6, the output rates of both schemes drop when the available memory decreases. The output rate of the original WinyDB drops sharply when the memory is not enough. This drop is because the original WinyDB discards some results when the memory is insufficient. In comparison, the optimized WinyDB tolerates memory insufficiency better by utilizing other PDAs to store some of the query results. As a result, the optimized WinyDB always has a higher output rate than the original, and this improvement becomes more significant when the memory size is much smaller than the sliding window size.

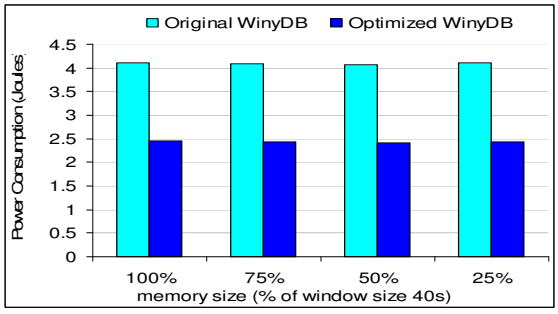

Figure 7. Power Consumption

With the same configuration as for Figure 6, Figure 7 shows the average node power consumption for one minute. The power consumption is nearly the same for all memory sizes. The reason is that the number of queries running in the WSNs is independent of the memory size. The improvement of optimized WinyDB is similar to Figure 5.

Experimental results for other join window sizes are similar to those in Figures 6 and 7.

5. Related Work

There are embedded database systems for handheld devices, such as IBM DB2 Everyplace [14], Microsoft SQL Server 2005 Mobile Edition [24], Oracle Berkeley DB [28], and Sybase SQL Anywhere [30]. These systems deal with disk-resident relational tables and usually have a backend database server to connect with. There have been also a few systems for sensor data collection from handheld devices or microserver-grade sensor nodes, for example, CodeBlue [23], MSR Sense [27], and TASK [5]. However, they are either designed for special purpose (e.g., medical care applications in CodeBlue) or for digital signal processing and visualization. In comparison, WinyDB is designed as a general-purpose query processing system on PDAs for querying sensor networks.

Existing query processing systems for sensor networks or data streams, such as Aurora [1], Cougar [35], NiagaraCQ [8], TelegraphCQ [7], and TinyDB [19] assume queries are issued from a PC-grade central base station. Query parsing, optimization, and query result processing are done at this base station. In contrast, WinyDB runs on PDAs to query WSNs and multiple PDAs running WinyDB can answer queries collaboratively. As a result, the query optimization in WinyDB considers the resource constraints of both the WSNs and the PDAs and responds to the current status of PDAs dynamically.

Many techniques [3][12][22] are proposed to allow resource sharing between similar queries. Arasu et al. [3] proposed a suite of algorithms to exploit the resource sharing opportunities in sliding-window aggregations. Goldstein et al. [12] presented a scalable algorithm to determine whether part or all of a complex query can be computed from materialized views. Fjords [22] is a data structure for query plans such that related queries can be combined into a single Fjord. NiagaraCQ [8] evaluates a large number of continuous queries efficiently by grouping queries with selection predicates on different constants. In comparison, WinyDB focuses on sharing sensor data streams as well as query results among multiple PDAs. The query sharing supported by WinyDB includes selections, projections, aggregations, and window joins. The queries can be either snapshot or continuous.

6. Conclusions and Future Work

In this paper, we have presented our embedded query processing system, WinyDB, for handheld devices to query sensor networks. In particular, we have focused on the collaborative processing techniques for selection, projection, aggregation, and window join queries. Our experiments with WinyDB running on real PDAs querying simulated WSNs have

shown that, with collaborative query processing, the output rate of the queries and the power consumption of the sensor nodes can be improved significantly. Furthermore, this collaboration is especially beneficial to continuous sliding window join queries when the memory size of a handheld device is limited

Developing a distributed, full-fledged query processor for PDAs to query sensor networks involves numerous interesting research issues. Our work on WinyDB has only been initial steps on the collaborative query processing techniques. Future work along this line includes extensions to WSNs with multiple sinks, count-based sliding window join queries, and other collaboration schemes.

7. References

[1] D. Abadi, D. Carney, et al. "Aurora: A New Model and Architecture for Data Stream Management," VLDB Journal, 2003

[2] Ian F. Akyildiz, Weilian Su, et al. "A Survey on Sensor Networks," IEEE Communications Magazine, vol. 40, no. 8, pp. 102-114, August 2002

[3] Arvind Arasu, Jennifer Widom, "Resource Sharing in Continuous Sliding-Window Aggregates," VLDB, 2004

[4] Philippe Bonnet, J.E. Gehrke, and Praveen Seshadri, "Towards Sensor Database Systems," Proceedings of the Second International Conference on Mobile Data Management in Hong Kong, 2001

[5] Phil Buonadonna, David Gay, et al. "TASK: Sensor Network in a Box," European Workshop on Sensor Networks (EWSN), 2005

[6] Alberto Cerpa, Jeremy Elson, et al. "Habitat Monitoring: Application Driver For Wireless Communications Technology," Proceedings of the 2001 ACM SIGCOMM Workshop on Data Communications, 2001.

[7] Sirish Chandrasekaran, Owen Cooper, et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," CIDR, 2003

[8] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," SIGMOD, 2000

[9] Crossbow Corp. http://www.xbow.com/

[10] Abhinandan Das, Johannes Gehrke, Mirek Riedewald, "Approximate join processing over data streams," SIGMOD, 2003

[11] GMP Wireless Medicine, Inc., http://www.wirelessecg.com/

[12] Jonathan Goldstein, Per-Åke Larson, "Optimizing queries using materialized views: a practical, scalable solution," SIGMOD, 2001

[13] Habit Monitoring on Great Duck Island. http://www.greatduckisland.net/

[14] IBM DB2 Everyplace, http://www-306.ibm.com/software/data/db2/everyplace/

[15] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," MobiCom, 2000.

[16] Intel Lab Data. http://berkeley.intel-research.net/labdata/

[17] Jaewoo Kang, Jeffrey F. Naughton, Stratis D. Viglas, "Evaluating Window Joins over Unbounded Streams," ICDE, 2003

[18] Philip Levis, Nelson Lee, et al. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," SenSys 2003

[19] Samuel R. Madden, Michael J. Franklin, et al. "TinyDB: An Acqusitional Query Processing System for Sensor Networks". ACM TODS, 2005

[20] Samuel R. Madden, Michael J. Franklin, et al. "The Design of an Acquisitional Query Processor for Sensor Networks". SIGMOD, June 2003

[21] Samuel R. Madden, Michael J. Franklin, et al. "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," OSDI, 2002

[22] Samuel Madden, Michael J. Franklin, "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data," ICDE, 2002

[23] David Malan, Thaddeus Fulford-Jones, et al. "CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care," ACM Workshop on Applications of Mobile Embedded Systems. Boston, Massachusetts, 2004

[24] Microsoft SQL Server 2005 Mobile Edition, http://www.microsoft.com/sql/editions/sqlmobile/default.mspx

[25] The Mobihealth Project, http://www.mobihealth.org/

[26] R. Motwani, J. Widom, et al. "Query Processing, Resource Management, and Approximation in a Data Stream Management System," CIDR, 2003

[27] MSR Networked Embedded Sensing Toolkit (MSR Sense), http://research.microsoft.com/nec/msrsense/

[28] Oracle Berkeley DB Product Family, http://www.oracle.com/technology/products/berkeley-db/index.html

[29] Tamer Ozsu, P. Valduriez, "Principles of Distributed Database Systems," 2nd edition, Prentice-Hall, Inc., 1999,

[30] Sybase SQL Anywhere, http://www.sybase.com/products/mobilesolutions/sqlanywhere

[31] TinyDB, http://telegraph.cs.berkeley.edu/tinydb/

[32] Jeffrey D. Ullman, "Principles of Database and Knowledge-Base Systems", Volume I. Computer Science Press 1988

[33] Windows CE, http://msdn.microsoft.com/embedded/windowsce/

[34] WinyDB, http://www.cse.ust.hk/winydb/

[35] Y. Yao, J. E. Gehrke, "Query Processing for sensor networks, " CIDR, 2003

[36] Wei Ye, John Heidemann, and Deborah Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks," INFOCOM, 2002

[37] Rui Zhang, Nick Koudas, et al. "Multiple aggregations over data streams," SIGMOD, 2005