Change-aware Skyline Query and Update in Wireless Sensor Networks

Cheng Wang[1, a], Yongli Wang[1, b], Yuncheng Wu[1], Shujie Sun[2,]

[1]Department of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, 210094, China

[2]Department of Oil seven factory in Daqing City, Daqing, Heilongjiang 163517, China

[a]email:wangcheng46@hotmail.com, [b]email:yongliwang@njust.edu.cn

Keywords: Wireless Sensor Network; Compressed Skycube; Extended Skyline; Skyline Query; Skyline Update

Abstract. In energy-limited wireless sensor networks, multi-subspace skyline query is widely used for all kinds of applications. In this paper, we propose a change-aware skyline query model based on Compressed Skycube (CSC) and Extended Skyline (ES), which reduces the traffic to save the energy of sensor in wireless sensor network on the premise of meeting user needs. We analyze the updating possibilities of different sensors according to the variously updating frequency in monitoring crop growth environment. We introduce an UpdateCSC algorithm based on CSC structure and an UpdateES algorithm based on Extended Skyline. Experiments on simulated environment show that the proposed model and algorithms can deal with mufti-subspace skyline query, which improve the computational efficiency and effectively reduce the updating energy.

Introduction

Wireless Sensor Network (WSN) consists of many stationary or mobile sensor nodes in a specific area, and each node has a certain computing ability and storage capacity which is to perceive, collect, and process the monitoring information about an object, and report the result to users[1-3]. Because there are many limits in sensor nodes, such as power energy, communication capabilities, network transmission bandwidth and so on, the existing database query processing technology could not widely used in WSN. Currently, data processing algorithms plays an important role in WSN research.

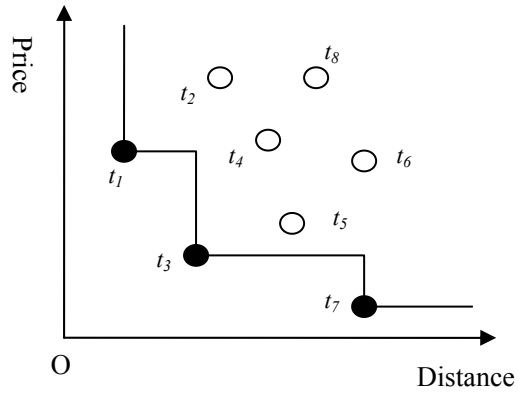

Fig.1. Skyline query of finding hotel

Skyline query [4], also known as Pareto (reach own optimal situation without damaging others' benefits), is very useful for making decisions about multi-objective selection. Specifically speaking, skyline query processing is to select a subset *A* from a *D*-dimensional set *S*, and for any element in subset *A*, it would not be dominated by any other elements in *S*. Given two *D*-dimensional objects *p* and *q*, *p* dominates *q* means that, *p* is better than *q* at least on one dimension, and on other dimensions *p* is not worse than *q*. A typical example of skyline query is, when you go to a beach for holiday, you may want to find a hotel that is cheap and close to the beach. As shown in figure 1, suppose that each *2*-dimensional object represents a hotel, and the value of each dimension represents a property of the hotel (e.g. distance), then the skyline include t_1, t_3, t_7. With skyline

query, you just need to choose in skyline result to find your favorite hotel [5].

However, in wireless network applications, different users may care about different subspace skyline query, and the data may change frequently. For example, as shown in figure 2, there are 8 greenhouses $t_1 \sim t_8$ in greenhouse system, each greenhouse has 4 types of sensors, including temperature, humidity, soil moisture, soil PH, to monitor environment in the house. Suppose the most appropriate conditions for vegetable are as following, temperature is 20℃, humidity is 30%rh, soil moisture is 15%, soil pH is 8. Sometimes, the user may want to know skyline query result on all dimensions (4 dimensions here), sometimes the user may be only interested on one dimension or multi-subspace skyline query (e.g. the temperature of outside is very high, the user need to pay significant attention on temperature and humidity subspace skyline query).

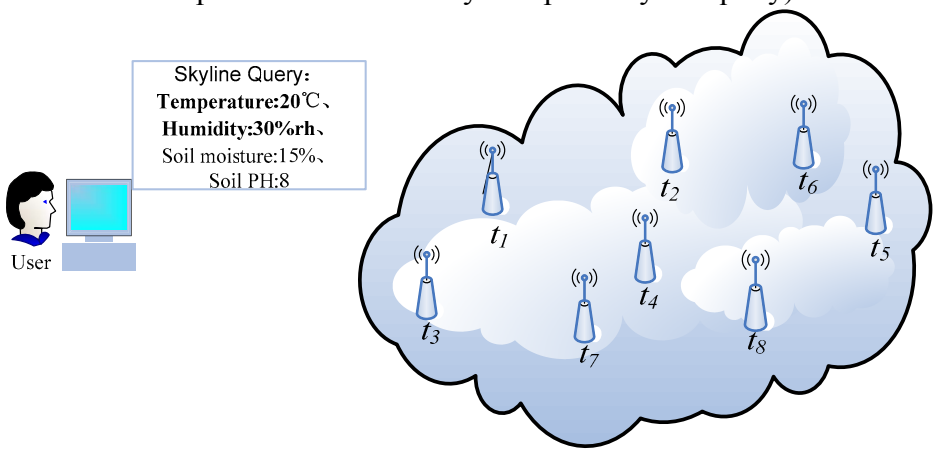

Fig.2. Greenhouse monitoring system deployed with wireless sensor network

In addition, the sensor data in the greenhouse rather changes in accordance with updating frequency respectively. For example, the temperature sensor's updating frequency is 15 minutes, the humidity sensor's updating frequency is 30 minutes, the soil moisture sensor's updating frequency is 1 hour, and the soil PH sensor's updating frequency is 4 hours. If we use traditional skyline query processing algorithm, when one type of sensor's data is changed, we need to acquire all sensors' data to re-compute the skyline. This will increase the amount of data transmission and energy consumption greatly in WSN.

To sum up, the key point of multi-subspace skyline queries in pervasive computing environment is to reduce energy consumption. In this paper, we analyze the Extended Skyline Query and Compressed Skycube Structure, and propose a new method to reduce energy consumption of sensor network and improve the efficiency of skyline computing. The contribution of this paper is as follows: (1) According to the relationship between extended skyline query and compressed skycube structure, we propose ES-CSC (Extended Skyline-Compressed Skycube) model, which could reduce energy consumption of sensor network effectively; (2) We put forward an update algorithm based on ES-CSC model when sensors' data are changed.

The rest of this paper is organized as follows. Section 2 describes related work on skyline query. Section 3 introduces relevant definitions and presents ES-CSC model. Section 4 presents an update algorithm based on ES-CSC. Finally, Section 5 shows experimental result and Section 6 concludes this paper.

Related Work

The skyline query is a typical multi-objective optimization problem, and its research can be traced back to 1960. The skyline study in the database field began in 2001, which mainly concerned how to process skyline query in the case of large amount of data and could not be placed in memory. Current research on skyline query mainly focus on the whole space and multi-subspace query.

Borzsonyi *et al.* proposed BNL algorithm and D&C (divide and conquer) algorithm, BNL compares every object with others and returns a set of objects that do not dominated by any of others[4]; D&C divides objects into several parts and computes the skyline of each part respectively,

and merges them to get the final skyline. SFS[7] algorithm sorts the data set first, and puts a certain number of objects in a window as skyline objects, then every object compares with skyline objects, and it replaces one or many of the skyline objects if it dominate them. BBS[8] algorithm and NN[9] algorithm are based on the nearest neighbor strategy, their difference is, NN performs multiple nearest neighbor queries, but BBS only executes index traversal once, the result showed the BBS algorithm achieves the best I/O.

The algorithms proposed above assumed that all objects are in a certain D-dimensional space, and the skyline query's dimension does not change. But in real life, different users may have different interest. Recent studies prefer researching variable query dimensions scenes, that is, given a set of D-dimensional objects, skyline queries can be executed on any of the D-dimensional subspace. Pei *et al.*[11] introduced notions of *skyline group* and *decisive subspace* by combining the semantics of skyline, and provided information for subspace skyline query using skyline group and decisive subspace. According to extended skyline theory, Xin Junchang[5] proposed a multi-subspace skyline query algorithm based on efficient energy.

Recently, researchers in the domain of *data warehouse* and *online analytical processing* are interested in multi-subspace skyline query. Yuan et al.[10] proposed *skycube* structure which is the results of all non-empty multi-subspace skyline queries in a given data set, and presented two algorithms that could process several skyline queries simultaneously and effectively. BUS (bottom-up skycube) algorithm computes each *Cuboid* in the skycube from bottom to up; and TDS (top-down skycube) algorithm computes *Cuboid* by using share division, merge, and share parent results. But due to the number of skyline is 2^D-1(D-dimensional objects), it is inefficient to compute and store all the *Cuboid* results. Tian Xia[6] presented *compressed skycube (CSC)* structure based on skycube, which could improve the efficiency of computing and storing *Cuboid* results; and they put forward a solution for updating one tuple based on CSC structure.

However, in greenhouse system, different sensor has different updating frequency, it means that, the data does not update by tuple, but update by attribute. This situation is similar with column storage mechanism. As we know, most of existing work is based on row storage. When we want to modify value of a specific attribute, we need to get the whole tuple. Then as mentioned in section 1, if there are 4 types of sensor (4 attributes, or 4 dimensional) with different updating frequency, we need to get tuples four times than the situation that their updating frequencies are the same. And we know, one feature of WSN is limited energy, so it will have great energy consumption when their updating frequencies are different. So, in order to solve these problems, we combine the extended skyline query and compressed skycube structure, propose a query model: ES-CSC, and we also put forward an updating algorithm based on this model.

Model and Definitions

At present the main means is low click type and the lever type, low click type is on the bottom of the ball through attack the ball flew over obstacles, this method is able to pick the ball's advantages and makes the energy loss in institutions least, the shortcoming is the ball high requirement of the shape of the electromagnetic valve [11]. Therefore, the development of a high-performance control system of soccer robot has become an urgent desire for soccer robot fans.

First, we assume that the values of sensor data in greenhouse system environment are shown in table 1 (where t_x represents the number of greenhouse, each tuple represents the set of each attributes in the same greenhouse). Suppose there is a query to find the most appropriate greenhouses, the query conditions are: temperature equals 20℃, humidity equals 30%rh, soil moisture equals 15%, soil PH equals 8. Then the variances between every greenhouse's sensor value and the values that query expect are shown in table 2.

In this paper, the query model and update algorithm are based on compressed skycube structure (CSC)[6] and extended skyline query[12]. The CSC structure is to compute the minimum subspace (mss) of the whole skycube structure[10]. It only store data which is relate to skyline query results, which reduces storage cost and transmission cost greatly. Extended skyline query is to merge the

same subspace of a number of skyline queries. Computing several extended skyline queries is more efficient than computing each skyline query (specific proof, see [13]), so extended skyline query also can improve query efficiency and reduce transmission cost. The following we will give a brief introduction to compressed skycube structure and extended skyline query.

Table 1. Environment conditions in eight greenhouses

	Temperature (℃)	Humidity (%rh)	Moisture (%)	Soil PH
t_1	22	33	17	7
t_2	24	29	16	5
t_3	18	38	19	7
t_4	19	28	12	6
t_5	26	31	13	9
t_6	25	27	18	10
t_7	15	28	14	7
t_8	16	25	18	10

Table 2. variances value to query conditions

	Temperature (℃)	Humidity (%rh)	Moisture (%)	Soil PH
t_1	2	3	2	1
t_2	4	1	1	3
t_3	2	8	4	1
t_4	1	2	3	2
t_5	6	1	2	1
t_6	5	3	3	2
t_7	5	2	1	1
t_8	4	5	3	2

Table 3. Summary of notations

Notation	Definitions
T	tuples in wireless sensor networks
D	full-space
S	set of skyline query or extended skyline query
T	one tuple in T
t[i]	value of tuple t on subspace i
U,V	subspace of D
skyline(U)	skyline on subspace U
mss(t)	minimum subspaces of tuple t
XSkyline(U)	extended skyline on subspace U
S	initial skyline query
X	extended skyline query

Table 4. skycube structure

Cuboid	*skyline*
Temperature	t_4
Humidity	t_2,t_5
Moisture	t_2,t_7
PH	t_1,t_3,t_5,t_7
Temperature, Humidity	t_2,t_4
Temperature, Moisture	t_1,t_2,t_4
Temperature, PH	t_1,t_3,t_4
Humidity, Moisture	t_2
Humidity, PH	t_5
Moisture, PH	t_7
Temperature, Humidity, Moisture	t_1,t_2,t_4
Temperature, Humidity, PH	t_1,t_2,t_4,t_5,t_7
Temperature, Moisture, PH	t_1,t_2,t_4,t_7
Humidity, Moisture, PH	t_2,t_5,t_7
Temperature, Humidity, Moisture, PH	t_1,t_2,t_4,t_5,t_7

On any subspace, the domination relationship of tuples is defined as follows.

Definition 1: dominate. Tuple *t* dominates tuple t' on *U* (denoted as $t_U \geq t'_U$) if and only if two conditions meets: 1) for any dimension *i* on *U*, *t* is not worse than t', that is $t[i] \leq t'[i](\forall i \in U)$; 2) there exists at least on dimension *j*, *t* is better than t', that is $t[j] < t'[j](\exists j \in U)$.

Definition 2: strictly dominate. Tuple *t* strictly dominates tuple t' on U (denoted as $t_U \rightarrow t'_U$) if and only if any dimension *k* on *U*, *t* is better than t', that is $t[k] < t'[k](\forall k \in U)$.

Compressed Skycube Structure

Skycube structure, based on the conventional multi-dimensional hierarchy structure - *Data Cube*, is the whole result set which is computed from skyline query on every non-empty subspace of a given data set. Skyline result set of each subspace is called a *Cuboid*[10]. The skycube structure of above-mentioned greenhouse system is shown in table 4 (e.g. the first row means, t_4 is in the cuboid <Temperature>; the last row means t_1, t_2, t_4, t_5, t_7 is in the cuboid <Temperature, Humidity, Moisture, PH>, which is skyline result of full-space).

Obviously, there are many duplicate tuples in table 4, which is not only a waste of storage space, but also a waste of energy in WSN when it transmit duplicate tuples. In order to reduce duplicate tuples in skycube, we quote the notion of *minimum subspace (mss)*[6],which is the basis of *compressed skycube (CSC).*

Definition 3: minimum subspace. For a given tuple *t*, the *minimum subspace* of *t*, denoted as *mss(t)*, is a subset of all subspaces, such that $\forall U \in mss(t), t \in skyline(U), and\ \forall V \subset U, t \notin skyline(V)$.

The minimum subspace of greenhouse system is shown in table 5 , where t_1, t_2, t_4, t_5, t_7 are in skyline(D); although t_3 is not in skyline(D), its minimum subspace is not empty, this means t_3 is in the skyline of one or some subspace; mss(t_6) and mss(t_8) is empty, which means t_6 and t_8 is not only in skyline(D), but also not in skyline of any subspace. So they are not in table . Base on definition 3, the compressed skycube structure is defined as follows.

Table 5. minimum subspaces of tuple *t*

	mss(t)
t_1	<PH>, <Temperature, Moisture>
t_2	<Humidity>, <Moisture>
t_3	<PH>
t_4	<Temperature>
t_5	<Humidity>,<PH>
t_7	<Moisture>,<PH>

Table 6. CSC structure

Cuboid	Skyline
Temperature	t_4
Humidity	t_2 , t_5
Moisture	t_2 , t_7
PH	t_1 , t_5 , t_7 , t_3
Temperature, Moisture	t_1

Definition 4: compressed skycube. The *compressed skycube* (CSC) consists of non-empty cuboids *U*, and for any tuple $t \in U$ if and only if $U \in mss(t)$.

According to definition 4, we can get CSC structure of greenhouse system, as shown in table 6. Take tuple t_1 as an example, in the skycube shown in table 4, t_1 is exists in seven cuboids, including <PH>, <Temperature, Moisture>, <Temperature, PH>, <Temperature, Humidity, Moisture>, <Temperature, Humidity, PH>, <Temperature, Moisture, PH> and <Temperature, Humidity, Moisture, PH>. In this way, we need to store t_1 for seven times. But based on table 6, we only need to store t_1 twice in CSC, <PH> and <Temperature, Moisture>.

We can get the result from table 6 that, five subspaces (<Temperature>, <Humidity>, <Moisture>, <PH>, <Temperature, Moisture>) and ten tuples (t_1, t_2, t_5, t_7 twice, and t_3, t_4 once) could represent 15 subspaces and 40 tuples stored in table 4. In a word, CSC structure can improve storage efficiency greatly.

Extended Skyline Query

Definition 5: extended skyline. Given a *D*-dimensional tuple set *T*, *U* is subspace of *D* ($U \subseteq D \wedge U \neq \emptyset$), then *D* is parent space of *U*. The extended skyline on *U* (denoted as *XSkyline(U)*) contains every tuple that is not strictly dominated by others on all subspace of *U*.

According to table 4, on cuboid <Temperature>, the skyline is t_4; on cuboid <Humidity>, the skyline is t_2, t_5; on cuboid <Temperature, Humidity>, the skyline is t_2, t_4. According to table 2, t_5. has the same value with skyline(Temperature,Humidity) on subspace <Humidity>. So, the extended skyline on <Temperature, Humidity>, called XSkyline(Temperature, Humidity), is t_2, t_4, t_5.

To discuss the property of extended skyline query, we quote a quality of extended skyline[5], denoted as lemma 1.

Lemma 1 Given a set of *D*-dimensional tuples *T*. *U* and *V* are subspaces of *D* ($U \subseteq V$). Suppose $t \in skyline(U)$ *and* $t' \in skyline(V)$, then $skyline(U) \subseteq skyline(V)$ *or* $t[U] = t'[U]$.

Proof: Contradiction. Suppose tuple *t* is not in *skyline(V)*, and it does not have the same value with any tuple in *skyline(V)* on space *U*.

Analyze: 1) if $t \notin skyline(V)$, then there exists tuple t' which dominates *t* on space *V*; because $U \subseteq V$, so t' dominates *t* on space *U* too, that means $t'[U] \leq t[U]$. 2) for *t* does not have the same value with t' on space *U*, then $t'[U] \neq t[U]$. In summary, $t'[U] < t[U]$, which means there exists t' dominates *t* on space *U*, so $t \notin skyline(U)$. This contradict hypothesis.

According to lemma 1, we come to the conclusion as theorem 1.

Theorem 1 If *U, V* are subspaces of *D*, and $U \subseteq V$, then $skyline(U) \subseteq XSkyline(V)$.

Proof: Contradiction. Suppose there exists tuple $t \in skyline(U)$ *and* $t \notin XSkyline(V)$.

Analyze: 1) because $t \in skyline(U)$ and according to *definition 1*, we know that, *t* is not dominated by other tuples on space U. 2) $t \notin XSkyline(V)$, so $\exists t' \in T, t'$dominates t on V. according to lemma 1, when $U \subseteq V$, t'*strictly dominates* t *on* U *or* $t'[U] = t[U]$. From 1) we know, *t* is not dominated by others, so $t'[U] = t[U]$. However, according to definition 5,

$\forall t'' \in \mathrm{T}, \text{if } t''[\mathrm{U}] = t[\mathrm{U}], \text{then } t'' \in \mathrm{XSkyline}(V)$. Contradiction to hypothesis.

From what we have discussed above, the skyline of subspace is contained by the extended skyline of parent space. In this case, if we know the extended skyline of parent space in a certain time, we will know the skyline of subspace when there are skyline queries on such subspace, then the back server do not need to retrieve sensor data from WSN. As we see, computing several extended skyline queries on parent space could get the skyline of subspace, so can we just compute a full-space extended skyline query to solve all subspace skyline queries? Theoretically, it is possible, but it will cost too much energy. The mathematical expectation on the number of tuples in skyline is $E[skyline] = O(ln^{k-1}n/(k-1)!)$ (n is number of tuples, k is dimensionality)[14]. Extended skyline only add tuples with same value on subspace, so the mathematical expectation on extended skyline $E[XSkyline] = E[skyline]$. Then we can find, the number of tuples in extended skyline will be greatly increasing when dimensionality increasing. It would be non-efficiency for using a full-space extended skyline to replace all subspace skyline queries. So we can join a skyline query into an existed extended skyline query, but not generate a new extended skyline query.

To sum up, the factor that really decide whether an extended skyline query *X* exist is whether there is a initial skyline query *S*, *S'*s query space is equal to *X*'s. Here we introduce determined query and coverage query.

Definition 6: determined query and coverage query. Given an extended skyline query *X*, *determined query of X* means its query space is equal to *X*'s query space; *coverage query of X* means its query space is a subspace of *X*'s query space.

Take an example, suppose there are 6 initial skyline queries in greenhouse system shown in table 7a, s_1, s_2, s_3 s_4, s_5, s_6, extended skyline queries are shown in table 7b.

Table 7a. An Example of Extended Skyline Query(Initial)

Initial Skyline Query	Query Space	Extended Skyline Query
s_1	<Temperature>	x_1
s_2	<Temperature, Moisture>	x_1
s_3	<Temperature, Humidity, Moisture>	x_1
s_4	<Humidity, Moisture>	x_2
s_5	<Moisture, PH>	x_2
s_6	<Humidity, Moisture, PH>	x_2

Table 7b: An Example of Extended Skyline Query

Extended Skyline Query	Extended SkylineQuery Space	Determined Query	Coverage Query
x_1	<Temperature, Humidity, Moisture>	s_3	s_1, s_2
x_2	<Humidity, Moisture, PH>	s_6	s_4,s_5

According to table 2 and table 7, we can know that, by using extended skyline query, the number of skyline queries can be reduced from 6 to 2, the number of tuples transmitted in WSN can also be reduced. So extended skyline query could also improve query efficiency (Algorithms of extended skyline query see [5]).

ES-CSC Query Model

Now that both CSC structure and ES query can improve skyline query efficiency, could we carry out ES query on CSC structure? Actually, according to lemma 1, when we compute extended skyline on parent space *V*, we need to compare every tuple's value on *U* with tuples in *skyline(V)*, to judge whether they are equal. But in CSC structure, this extra work is unnecessary.

Theorem 2 In CSC, *T* is a set of *D-dimensional* tuples, *V* is subspace of *D*. Suppose that set *A* is *XSkyline(V)*. Then $\forall U \subset V, if\ \exists U \in CSC, skyline(U)$ in CSC structure is in set *A*.

Proof: Contradiction. Suppose that $\exists t' \notin CSC\ and\ t'[U] = t[U](t \in XSkyline(V))$.

According to the definition of skyline, $t' \in skyline(U)$. And according to *definition 4*, we can deduce $U \notin mss(t')$, then there are two situations based on *definition 3*.

1) $t' \notin skyline(U)$. This contradicts with hypothesis.
2) $t' \in skyline(U)$, *and also* $t' \in skyline(W)$ (W is a subspace of U). According to definition of minimum subspace, $W \in mss(t')$. Then because *W* is a subspace of *U*, skyline of *W* would be added in *A* when computing $XSkyline(V)$, so $t' \in XSkyline(V)$. This

contradicts with hypothesis too.

From theorem 2, we can draw the conclusion that CSC structure has saved tuples who have the same value with $skyline(V)$ on space *U*. When computing extended skyline of parent space, we just need to join the subspace in CSC together. By this means, we can save query time and improve query efficiency.

In this paper, we present a skyline query processing model --- *ES-CSC*, which based on ES query and CSC structure. As figure 3 shows.

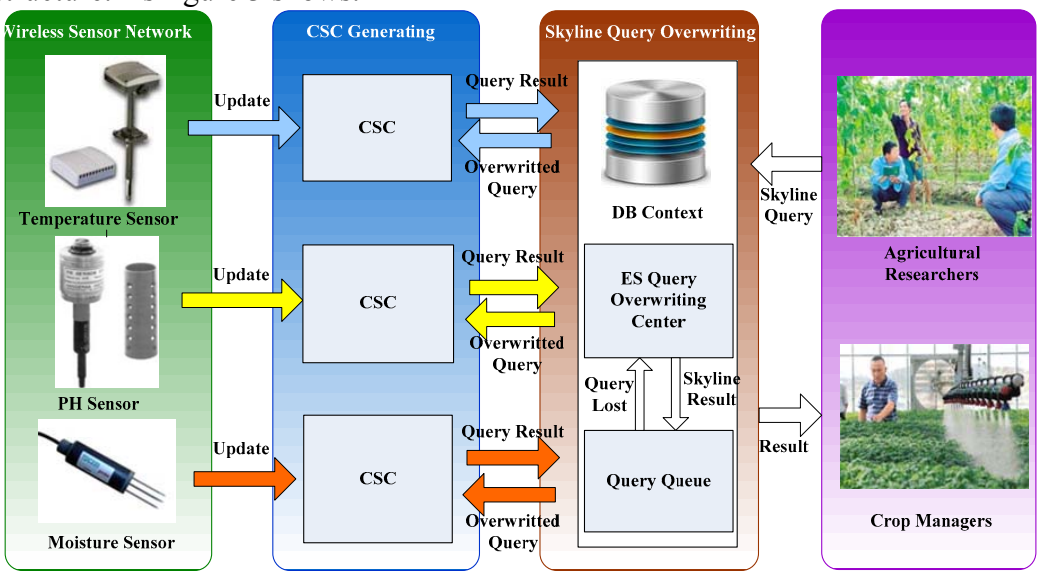

Fig.3. ES-CSC Query Model

The kernel of ES-CSC query model is *CSC Generating Module* and *Skyline Query Overwriting Module*. In figure 3, *query queue* stores skyline queries that user proposed and the result of frequent skyline queries; *ES query overwriting center* loads queries in the queue, and overwrites skyline query, then send the extended skyline queries to *CSC module*. *CSC module* receives the overwrite queries, and send the results to overwriting center and query queue. ES-CSC query model could reduce storage space, improve query efficiency, and lessen retrieval data from WSN.

Updating on ES-CSC Model

Because energy in WSN is limited, we always want to lessen retrieval data as much as possible on the premise of meeting user's requirements. Traditional updating methods need to re-compute all the skyline queries, and send all the sensor data back. But in greenhouse system, not all attributes of a tuple is changed in a moment. According to their own specialities, different sensor has different updating frequency, we have assumed that temperature sensor update every 15 minutes; humidity sensor: 30 minutes; moisture sensor: 1 hour; PH sensor: 4 hours. In this situation, if we need to re-compute all the cuboids when a type of sensor updating, it will cost too much energy. But it could be better when updating on ES-CSC model.

Updating on CSC Structure

In CSC structure, one attribute's value of some tuples changes would not change *skyline(D)*, if we have to re-compute *mss(t)* and CSC structure when every attribute's value of every tuple changes one time, no doubt, it waste lot of energy. So we need to judge whether new tuples would be added in *skyline(D)* when a attribute's value changes.

Theorem 3 Given a tuple *t* and $\forall t_{skyline} \in skyline(D)$, if $t_{skyline}$ strictly dominates *t*, then $mss(t) = \emptyset$.

Proof: according to *definition 2*, if $t_{skyline}$ strictly dominates *t*, for any attribute *i*, t[*i*] is worse than $t_{skyline}[i]$; then $\forall U \subseteq D$, *t* can not be in *skyline(U)*. Thus we know $mss(t) = \emptyset$ based on *definition 3*.

Lemma 2 Given a tuple *t* and $\forall t_{skyline} \in skyline(D)$, if $t_{skyline}$ dominates *t* and

$t_{skyline}[U] = t[U](U \subseteq D)$, then $mss(t) \subseteq mss(t_{skyline})$.

Proof: According to [6], $\forall V \subseteq U, if\ V \in mss(t_{skyline}), V \in mss(t)$. Since $t_{skyline}$ strictly dominates t on *D-U*, *t* can not be in the skyline of any subspace $V'(V' \cap (D-U) \neq \emptyset)$. Then $ss(t) \subseteq mss(t_{skyline})$.

Analyze of Updating on CSC

Tuples can be divided into three kinds based on CSC structure.

1) $t \in skyline(D)$: skyline of full-space (such as t_1, t_2, t_4, t_5, t_7);

2) $t \notin skyline(D) \cap mss(t) \neq \emptyset$: not in *skyline(D)*, but have the same value on subspace *U* with *skyline(D)* (such as t_3);

3) $t \notin skyline(D) \cap mss(t) = \emptyset$: not in skyline of any subspace (such as t_6, t_8). Among them, minimum subspace of first two is not empty, let's analyze these respectively.

1) $t \in skyline(D)$. Suppose the value of *t* is change to t_{new} on subspace *Y* (here $|Y| = 1$) after updating. Because $t \in skyline(D)$, CSC structure must be changed. There are two situations under this condition.

a) t_{new} dominates *t*. Cause $t \in skyline(D)$, there is not a tuple who dominates *t*. Now t_{new} dominates *t*, so there is not a tuple dominates t_{new} too. Thus $t_{new} \in skyline(D)$.

Suppose the value of other tuples are not changed, then according to *theorem 3*, for tuples $t_{other} \notin skyline(D) \cap mss(t) = \emptyset$ still would not be in CSC structure; Considering whether t_{new} dominates tuples in CSC structure, $\forall\ t_{CSC} \in CSC$ except t, compare to t_{new}: if t_{new} does not dominates any of them, then the minimum subspace and CSC structure would not be changed; else if t_{new} dominates a tuple, representing as t_{CSC}, based on *lemma 2*, we compare subspace *U* in $mss(t_{CSC})$ which includes *Y (Y ⊆ U)* to subspaces in $mss(t_{new})$, if there exists *U* in $mss(t_{new})$, we reserve $mss(t_{CSC})$; else add *U* in $mss(t_{new})$ and delete *U* in $mss(t_{CSC})$.

b) t_{new} does not dominates *t*. According to *definition 1* and *theorem 3*, for tuples $t_{other} \notin skyline(D) \cap mss(t) = \emptyset$ still would not be in CSC structure; $\forall\ t_{CSC} \in CSC$ except t , compare to t_{new}, if there does not exist dominate relationship, the minimum subspace and CSC structure would not be changed; else if t_{CSC} dominates t_{new} on subspace U, then based on *lemma 2*, we compare subspace *U* in $mss(t_{new})$ which includes *Y (Y ⊆ U)* to subspaces in $mss(t_{CSC})$, if they exists *U* in $mss(t_{CSC})$, reserve; else add *U* in $mss(t_{CSC})$ and delete *U* in $mss(t_{new})$.

2) $t \notin skyline(D) \cap mss(t) \neq \emptyset$. Suppose the value of *t* is change to t_{new} on subspace *Y* (here $|Y| = 1$) after update. There are two situations under this condition too.

a) t_{new} is dominated by *skyline(Y)* on subspace *Y*. If $t_{new}[Y]$ is equal to the value of *skyline(Y)*, then $\forall\ t_{skyline} \in skyline(D)$, find all tuples $t_{skyline}$ that dominates t_{new}. According to *lemma 2*, we clear $mss(t_{new})$ first, and compare the value on subspace *U* which in $mss(t_{skyline})$ to t_{new}, if $t_{new}[\mathrm{U}] = t_{skyline}[U]$, add *U* in $mss(t_{new})$; else do nothing. If $mss(t_{new}) = \emptyset$, remove t_{new} from CSC structure. If $t_{new}[Y]$ is not equal to the value of *skyline(Y)*, the update of t_{new} would not affect minimum subspace and CSC structure.

b) t_{new} strictly dominates *skyline(Y)* on subspace *Y*. Obviously, in this case , $t_{new} \in skyline(D)$. According to *lemma 2*, we compare every tuple t_{mss} whose $mss(t_{mss}) \neq \emptyset$ to t_{new}: if t_{new} strictly dominates $\mathrm{t_{mss}}$ on full-space *D*, then put all subspaces in $mss(t_{mss})$ into $mss(t_{new})$, clear $mss(t_{mss})$ and remove t_{mss} from CSC structure; if t_{new} strictly dominates t_{mss} not on full-space *D*, but on subspace *U* which in $mss(t_{mss})$, then add *U* into $mss(t_{new})$ and delete *U* from $mss(t_{mss})$; else if on subspace *U*, $t_{new}[U] = t_{mss}[U]$, then add *U* into $mss(t_{new})$ but reserve *U* in $mss(t_{mss})$.

3) $t \notin skyline(D) \cap mss(t) = \emptyset$. Suppose the value of *t* is change to t_{new} on subspace *Y* (here $|Y| = 1$) after update. This situation is similar with the second except one: when t_{new} is dominated by *skyline(Y)* on subspace *Y*, if $t_{new}[Y]$ has the same value with *skyline(Y)*, we need to add t_{new} into CSC structure; if $t_{new}[Y]$ does not have the same value with *skyline(Y)*, the minimum subspace and CSC structure would not be changed.

From what we analyzed above, we come to the conclusion that CSC structure would be changed only under three conditions. First, $t \in skyline(D)$ and t_{new} has domination relationship (dominate or dominated) with tuples in CSC structure. Second, $t \notin skyline(D) \wedge t_{new} \notin skyline(D)$ and t_{new} has the same value with tuple in *skyline(D)* on subspace *U*. Third, $t \notin skyline(D) \wedge t_{new} \in skyline(D)$. According to these conditions, we proposed *UpdateCSC* algorithm, the *UpdateCSC* algorithm could reduce the times to re-compute CSC structure as many as possible, thereby improve skyline query efficiency when updating. Table 8 shows some functions used in *UpdateCSC* algorithm. Figure 4 shows *UpdateCSC* algorithm.

Table 8: Functions and Their Explanation Used in UpdateCSC Algorithm

Function Name	Explanation
mss(t).add(U)	add *U* into *mss(t)*
mss(t).remove(U)	remove *U* from *mss(t)*
findDominateTuple(SD,t)	find tuples which dominate *t* in skyline set *SD*
overlapSubspace(mss(t_1), mss(t_2))	find overlap subspaces between *mss(t_1)* and *mss(t_2)*

Algorithm 1. *UpdateCSC*
Input: *skyline(D)*, *minimum subspaces* of all tuples, tuple *t* and $t_{new}[Y]$ (*Y* is update column and $|Y| = 1$)
Output: *minimum subspace*s after update

```
if  t ∈ SD                                  // the update tuple t ∈ skyline(D)
    if  t_new dominates t on U              // Declaration U (U is temporary subspace)
        ∀t_CSC (t_new strictly dominates t_CSC)  mss(t_new).add(U); mss(t_CSC).remove(U);
        ∀t_CSC (t_new[U] = t_CSC[U])   mss(t_new).add(U);
    else  t_new does not dominates t on U
        ∀t_CSC ( t_CSC strictly dominates t_new)  mss(t_CSC).add(U); mss(t_new).remove(U);
        ∀t_CSC (t_new[U] = t_CSC[U])  mss(t_CSC).add(U);
else  t ∉ SD                                // the update tuple t ∉ skyline(D)
    value = skyline(Y)
    if  value == t_new[Y]     // value of t_new has the same value with value of skyline(Y)
        t_skyline = findDominateTuple(SD, t_new); mss(t_new) = ∅;
      U = overlapSubspace( mss(t_new), mss(t_skyline)); mss(t_new).add(U)
    endif
    if  t_new strictly dominates skyline(Y)
        ∀t_mss(t_new strictly dominates t_mss on D)
                mss(t_new).add(mss(t_mss)); mss(t_mss) = ∅;
        ∀t_mss(t_new strictly dominates t_mss on U(U ≠ D))
                mss(t_new).add(U); mss(t_mss).remove(U)
        ∀t_mss(t_new[U] == t_mss[U])
                mss(t_new).add(U);
    endif
    endif
return mss
```

Fig.4. *UpdateCSC* Algorithm

Complexity Analysis of UpdateCSC

Assume that there are n_{mss} tuples in minimum subspaces, and dimensionality of full-space id *d*, the update space is *Y* ($|Y| = 1$).

Best Case: we need to compare t_{new} to all tuples in minimum subspaces to see whether they have domination relationship. According to our analysis, if all of them do not have any relationship with t_{new}, CSC structure would not change, then complexity is the comparing times, that is the number of tuples in minimum subspaces − n_{mss}. So under this situation, the algorithm complexity is $O(n_{mss})$.

Worst Case: every time we compare tuple *t* in minimum subspaces to t_{new}, we need to update minimum subspaces; and the max dimensionality of *mss(t)* is *d*. According to our analysis, there are 2^{d-1} subspaces contain space *Y*, and each of them need to compare to t_{new}. So the algorithm complexity is $O(2^{d-1} * n_{mss})$.

Average Case: in this case, we need to differentiate whether tuple *t* in $skyline(D)$.

1) $t \in skyline(D)$

Assume that after update, the possibility of t_{new} dominates *t* (denote as $P[t_{new} - t]$) and t dominates t_{new}(denoted as $P[t - t_{new}]$) are same, equal to 1/2; assume the possibility of t_{new} and $t_{mss} \in mss$ have domination relationship (denoted as $P[dom]$) and not (denoted as $P[none\_dom]$) are same, equal to 1/2 too; assume that if t_{new} and t_{mss} have domination relationship, the comparing times between t_{new} and tuples in $mss(t_{mss})$ which contain space *Y* is denoted as $revise\_mss(t_{mss})$; assume the total comparing times is *NUM*1 under $t \in skyline(D)$ condition.

According to our analysis, we can deduce to:

$$NUM1 = P[t_{new} - t] * P[dom] * revise\_mss(t_{mss}) * n_{mss} + P[t_{new} - t] * P[none_{dom}] * 1 * n_{mss} + P[t - t_{new}] * P[dom] * revise\_mss(t_{mss}) * n_{mss} + P[t - t_{new}] * P[none\_dom] * 1 * n_{mss}$$

Substitute by the assumption, we can get:

$$NUM1 = n_{mss} * (1/2) * (revise\_mss(t_{mss}) + 1) \quad (1)$$

2) $t \notin skyline(D)$

According to the analysis in 4.1.1, in situation 2) and 3), they all need to find tuples in minimum subspaces that dominates $\mathrm{t_{new}}$ or dominated by $\mathrm{t_{new}}$(assume the possibility of tuple dominates others or dominated by others are same, equal to 1/2), and revise minimum subspaces. Assume the total comparing times is *NUM2* under $t \notin skyline(D)$ condition, we can get:

$$NUM2 = n_{mss} * (1/2) * revise\_mss(t_{mss}) \quad (2)$$

Now we deduce the comparing times of mathematical expectation on updating minimum subspaces once, denoted as $E[revise\_mss(t_{mss})]$.

Because the dimensionality of full-space is *d*, there are $(2^d - 1)$ subspaces, assume that the possibility of each subspace exists in minimum subspaces are the same, equal to $1/(2^d - 1)$. When update minimum subspaces, we need to find all subspaces that contain space *Y*, and then compare. This problem could be transformed to select several dimensions (assume *Z*) from $(d - |Y|)$ dimensions, and compare value to t_{new} on subspace of *Z*+*Y*. Take an example, suppose we select $Z = \{A, B, C\}(|Z| = 3)$, then we need to enumerate all subspaces of *Z*, along with *Y*. Here the enumerations are <*Y*>, <*Y,A*>, <*Y,B*>, <*Y,C*>, <*Y,A,B*>, <*Y,A,C*>, <*Y,B,C*>, <*Y,A,B,C*>, are eight times, that is $2^{|Z|}$ (we can prove it is general using mathematical induction method). Besides, there are $C_{d-|Y|}^{|Z|}$ choices for selecting $|Z|$ dimensions from $(d - |Y|)$ dimensions. So the comparing times here is $C_{d-1}^{3} * 2^3$. We take abstract representation as follows:

$$E[revise\_mss(t_{mss})] = [1/(2^d - 1)] * \sum_{i=0}^{d-1} C_{d-i}^{i} * 2^i \quad (3)$$

According to (1) (2), we can get the mathematical expectation of total comparing times

$$E[NUM] = E[NUM1] + E[NUM2] \approx E[n_{mss} * revise\_mss(t_{mss})] = n_{mss} * E[revise\_mss(t_{mss})]$$

Substitute $E[revise\_mss(t_{mss})]$ by (3), we can get complexity of UpdateCSC algorithm in average case is $\Theta([n_{mss}/(2^d - 1)] * \sum_{i=0}^{d-1} C_{d-i}^{i} * 2^i)$.

If we need to consider the relationship between complexity with total number of tuples in minimum subspaces, we can use the mathematical expectation of extended skyline query proposed in section 3.2, $E[XSkyline] = E[skyline] = O(ln^{k-1}n/(k-1)!)$. Since expectation of minimum subspaces is approximately equal to expectation of extended skyline query (adding tuples with same value), we can substitute this equation to get the result.

Updating on Extended Skyline Query

In this paper, when data is updated on subspace *U*, extended skyline query need to re-compute too. Given a extended skyline query *X*, if it contains subspace *U*, then *X* would be update in accordance with data is updated, we need to clear coverage query set and re-generate extended skyline query[5]; if it does not contain subspace *U*, extended skyline query would not be changed.

Steps of *UpdateES* algorithm are as follows.

1) For all extended skyline queries X, to its determined query V, if $U \subseteq V$, then put X's *determined queries* and *coverage queries* into initial query set, and set X null;
2) According to extended skyline generate algorithm [5], re-compute initial query set;
3) Compute the extended skyline and send back, return.

In extended skyline query example in table 7*a*, when data on <Temperature> update in a moment, according to *step 1*, we find x_1 cause it contains subspace <Temperature>; and based on *step 2*, we put s_1, s_2, s_3 into initial query set, and re-generate extended skyline queries, as table 9 shown; at last send back the result and return.

Table 9. Update on Extended Skyline Query

Extended skyline	Query Space	Determined Query	Coverage Query
x_3	< Temperature, Moisture >	s_2	s_1
x_2	< Temperature, Moisture, PH>	s_6	s_3,s_4,s_5

Performance Evaluation

In this section, we would evaluate the performance based on the proposed ES-CSC model and *UpdateES-CSC* algorithm. We compare communication cost between query using ES-CSC model and query using traditional method --- BNL[4]. We also compare *CPU time* between updating on ES-CSC and updating on *skycube*. The experiment environment is Intel(R) Core(TM)2 Duo CPU T7500 @2.20GHz, 3.00GB Memory, 250GB disk and Windows XP operating system, all algorithm is based on C++.

Data set in this paper is generated by using method presented in reference [4], which is widely used for standard test data set, data distribution is independent distribution. According to the greenhouse system situation proposed above, we mainly analyze the situation when the number of greenhouse (which is number of tuples m) and the number of sensor type in greenhouse (which is dimensionality n) are varied. In simulation environment, we generate m greenhouses in $\sqrt{m} \times \sqrt{m}$ square units randomly, thus every greenhouse own 1 square unit; meanwhile, we implement n types sensor on every greenhouse (each type has one sensor). Each greenhouse chooses sensor at bottom-left as cluster head, to collect data of all sensors in this greenhouse. Assume that the communication radius between sensors in a greenhouse is 1 units, and set the max size of data packet is 48 bytes; the coordinate of sink node is (0, 0), each cluster head fuses data in that greenhouse and sends its data to sink node directly, and the sink node process data for sending back.

For better illustrating, we define

$$\text{communication cost} = \text{communication radius} \times \text{packet size}.$$

Thus, there are m tuples in greenhouse system, each tuple has n columns representing sensors' value. We set the parameters that we want to evaluate in table 10, if one parameter is varied, others are default.

Table 10. Experiment Parameter

Parameter	Default Value	Range
number of greenhouse --- m	300	100,200,300,400,500
number of sensor type --- n	5	3,4,5,6,7,

Compare to BNL on Skyline Query

In this section we mainly evaluate the communication cost on computing multi-subspace skyline query based on ES-CSC model (*QueryES-CSC*) and computing multi-subspace skyline query based on BNL (*QueryBNL*). Suppose there are 20 multi-subspace skyline queries.

First, evaluate the number of greenhouse m impact on the performance of two algorithms. Figure 5 shows, as m increasing, the communication cost of two algorithms increases. The reason is that when m increases, the number of tuples involved in computing greatly increase, which increases number of tuples in skyline. Meanwhile, as figure 5 shows, the communication cost of *QueryES-CSC* is much lower than that of *QueryBNL* when the number of sensor type n is equal, this is because, by using extended skyline query, it reduce transmission of duplicate tuples in

network, which reduce communication cost greatly.

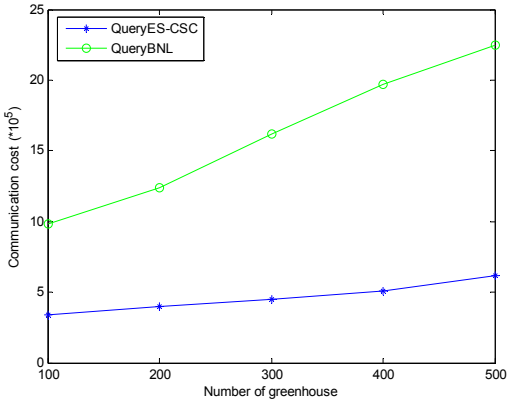

Fig.5. Number of greenhouse impact on skyline querying

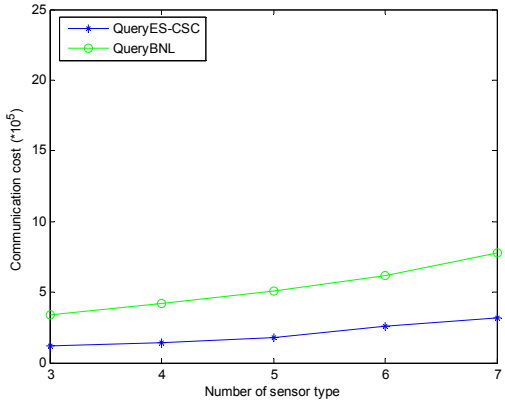

Fig.6. Number of sensor type impact on skyline querying

Then we evaluate the number of sensor type *n* impact on the performance of two algorithms. As figure 6 shows, the communication cost of two algorithms increase when *n* is growing. The reason is that the increasing dimensionality involved in computing expand the size of tuple. Besides, it can be seen that the communication cost of *QueryES-CSC* is lower than that of *QueryBNL* when the number of greenhouse *m* is equal, the reason is same as when *n* is equal.

Compare to Skycube on Update

This section we mainly evaluate CPU time for updating on ES-CSC model (*UpdateES-CSC*) and Skycube (*UpdateSkycube*)[10]. We would consider two situations, one is sensors' updating frequencies are same, which means data would update by tuple; one is their update frequencies are vary, which means data would update by attribute. Suppose there are 20 multi-subspace skyline queries.

Situation 1: Update frequencies are same

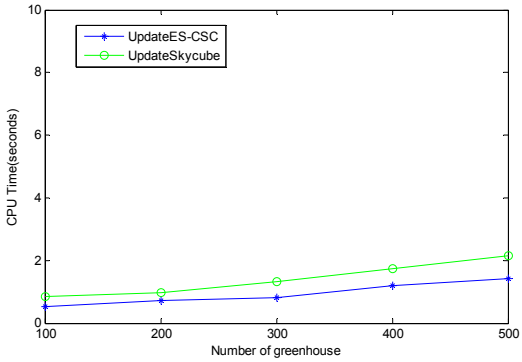

Fig.7. Number of greenhouse impact on updating of situation 1

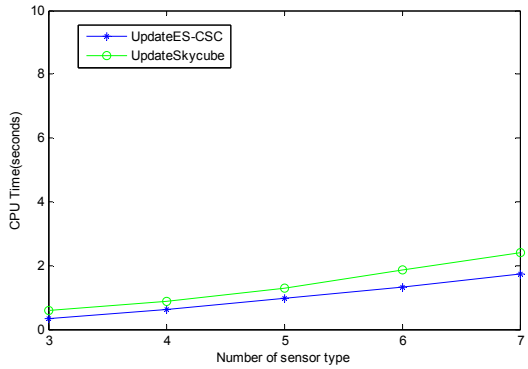

Fig.8. Number of sensor type impact on updating of situation

First, we evaluate the number of greenhouse *m* impact on the performance of two algorithms. As figure 7 shows, CPU time for updating increasing as *m* increasing. The reason is that, *m* increase would lead tuples for updating increase, which directly increase CPU time for iterate the whole data set. Meanwhile, we can get, CPU time cost by *UpdateES-CSC* is lesser than cost by *UpdateSkycube*. This is because, *UpdateES-CSC* algorithm does not re-compute tuples whose update do not affect CSC structure, thereby saving CPU time.

Then we evaluate the number of sensor type *n* impact on the performance of two algorithms. Figure 8 illustrates that when *n* increases, CPU time of two algorithms increases, and *UpdateES-CSC* algorithm uses less CPU time for updating than *UpdateSkycube* uses. Because *UpdateES-CSC* algorithm only considers space *V* who contains the updated subspace($U \subseteq V$), it does not compute other subspaces.

Situation 2: Update frequencies are vary

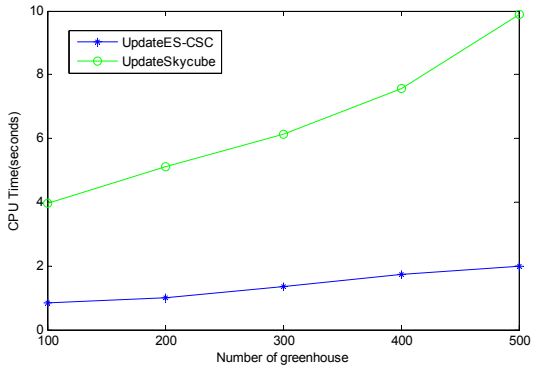

Fig.9. Number of greenhouse impact on updating of situation 2

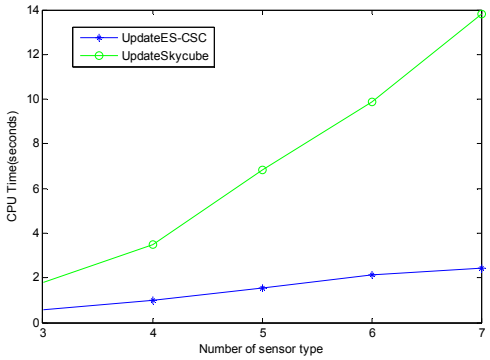

Fig.10. Number of sensor type impact on updating of situation 2

In this situation, we also evaluate the number of greenhouse *m* impact on the performance of two algorithms firstly. As shown in figure 9, CPU time for updating increasing as *m* increasing. This is because when *m* increases, the number of tuples involved in re-computing is increases. And we can notice that, CPU time cost by UpdateSkycube is significantly more than cost by UpdateES-CSC. This reason is same as situation 1. Meanwhile, comparing figure 9 to figure 7, we can get, CPU time for updating using by UpdateSkycube on the condition that their update frequencies are vary is approximately five times as large as their update frequencies are same; while CPU time for updating using by UpdateES-CSC only increases a little. The reason is that, when each type of sensor's data is change, UpdateSkycube need to re-compute the skyline, and the default value of the number of sensor type is close to five; but UpdateES-CSC only need to process a few computing to judge whether some tuple to add into or remove from the skyline while most of skyline do not need to change.

Then we evaluate the number of sensor type *n* impact on the performance of two algorithms in this situation. Figure 10 shows that when *n* increases, CPU time of two algorithms also increases. The reason is same as situation 1. Meanwhile, comparing figure 10 to figure 8, we can notice that when *n* is growing, CPU time of UpdateES-CSC on the condition that their update frequencies are vary is approximately *n* times as large as update frequencies are same. The reason is same as the above condition.

Experiment Summary

According to the above experiments, we know that, carrying out skyline query and updating on ES-CSC model are better than *QueryBNL* and *UpdateSkycube* respectively, especially when the number of sensor type *n* increasing. These experiments shows, using ES-CSC model in WSN could reduce energy consumption and improve updating efficiency greatly.

Conclusions

In this paper, we take wireless sensor network (WSN) as the background and research multi-subspace skyline query in greenhouse system. After analyzing compressed skycube (CSC) structure and extended skyline (ES) query, we propose a change-aware model --- ES-CSC and discuss three situations for updating on this model. Experiments show that the ES-CSC model has great performance on both multi-subspace skyline query and update.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by the National Natural Science Foundation of China under Grant 61170035, 61272420 and 61370207, and the Fundamental Research Funds for the Central Universities No. 30920130112006.

References

[1] Fung WF, Sun D, Gehrke J. Cougar: The network is the database. In: Franklin MJ, Moon B, Ailamaki A, eds. Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM Press, 2002. 621.

[2] Madden S, Franklin M, Hellerstein J, Hong W. The design of an acquisitional query processor for sensor networks. In: Halevy AY, Ives ZG, Doan AH, eds. Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM Press, 2003.491−502.

[3] Considine J, Li F, Kollios G, Byers JW. Approximate aggregation techniques for sensor databases. In: Gray J, Shenory PJ, eds. Proc. of the 20th Int'l Conf. on Data Engineering. Washington: IEEE Computer Society Press, 2004. 449−460.

[4] Borzsonyi S, Kossmann D, Stocker K. The skyline operator. In: Proc. of the 17th Int'l Conf. on Data Engineering. Heidelberg, IEEE Computer Society Press, 2001. 421-430. http://www.dbis.ethz.ch/research/publications/38.pdf

[5] Baichen Chen, Weifa Liang, Jeffrey Xu Yu. Energy-efficient skyline query optimization in wireless sensor networks. Wireless Networks. 2012, 18(8): 985-1004

[6] Tian Xia , Donghui Zhang . Refreshing the Sky: The Compressed Skycube with Efficient Support for Frequent Updates. SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.

[7] Chomicki J, Godfrey P, Gryz J, Liang D. Skyline with presorting. In: Proc. of the IEEE Int'l Conf. on Data Engineering. Los Alamitos: IEEE Computer Society Press, 2003. http:// www.cs.sfu.ca/CC/843/jpei.Skyline/Chomicki- Presorting-ICDE-3.pdf

[8] Papadias D, Tao Y, Fu G, Seeger B. Progressive skyline computation in database systems. ACM Trans. On Database Systems, 2005, 30(1):41-82, http://delab.csd.auth.gr/courses/c_mmdb/skyline.pdf

[9] Kossmann D, Ramsak F, Rost S. Shooting stars in the sky: An online algorithm for skyline queries. In: Proc. of the Int'l Conf. on Very Large Data Bases. 2002.

[10] Yuan Y, Lin X, Liu Q, Wang W, Yu JX, Zhang Q. Efficient computation of the skyline cube. In: Proc. of the 31st Int'l Conf. on Very Large Databases. ACM, 2005. 241-252.

[11] Pei J, Jin W, Ester M , Tao YF. Catching the best views of skyline: A semantic approach based on decisive subspaces. In Proc. of the 2005 Int'l Conf. on Very Large DataBases(VLDB 2005). 2005.

[12] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M.Vazirguannis. Skypeer: Efficient subspace skyline computation over distributed data[C]. In Proc. of the 23rd Int'l Conf. on Data Engineering(ICDE'07), Marmara Hotel, Istanbul, Turkey, April 2007, pp.415-425.

[13] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. An application-specific protocl architecture for wireless microsensor networks [J]. IEEE Transactions on Wireless Communications, October 2002, 1(4):660-670.

[14] S. Chaudhuri, N. N. Dalvi, R. Kaushik. Robust cardinality and cost estimation for skyline operator [C]. In Proceedings of the 22nd International Conference on Data Engineering(ICDE'06), Atlanta, GA, USA, April 2006, P.64.