A Novel Incremental Maintenance Algorithm of SkyCube

Zhenhua Huang and Wei Wang

Fundan University, China
{weiwang1, 051021055}@fudan.edu.com

Abstract. Skyline query processing has recently received a lot of attention in database community. And reference [1] considers the problem of efficiently computing a SkyCube, Which consists of skylines of all possible non-empty subsets of a given set of dimensions. However, the SkyCube is can not use further as original data set is changed. In this paper, we propose a novel incremental maintenance algorithm of SkyCube, called *IMASCIR*. *IMASCIR* splits the maintenance work into two phases: identify and refresh. All the materialized SkyCube views share two tables which stores the net change to the view due to the change to the original data set. In the phase of identify, we identify and store the source changes into these shared tables. Then in the phase of refresh, each materialized view is refreshed individually by applying these two shared tables. Furthermore, our experiment demonstrated that *IMASCIR* is both efficient and effective.

1 Introduction

Recently, there has been a growing interest in so-called skyline queries[2,3]. The skyline of a set of points is defined as those points that are not dominated by any other point. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension.

Numerous algorithms have been proposed for skyline retrieval. Borzsonyi et al.[2] propose the methods based on divide-and-conquer(DC) and block nested loop(BNL). Specifically, DC divides the dataset into several partitions that can fit in memory. The skylines in all partitions are computed separately using a main-memory algorithm, and then merged to produce the final skyline. BNL essentially compares each tuple in the database with all the other records, and outputs the tuple only if it is memory. The skylines in all partitions are computed separately using a main-memory algorithm, and then merged to produce the final skyline. BNL essentially compares each tuple in the database with all the other records, and outputs the tuple only if it is not dominated in any case. The sort-first-skyline(SFS)[4] sorts the input data according to a preference function, after which the skyline can be found in another pass over the sorted list. Tan et al.[3] propose a solution that deploys the highly CPU-efficient bit-operations by computing the skyline from some bitmaps capturing the original dataset. The authors also provide another method based on some clever observations on the relationships between the skyline and the minimum coordinates of individual points. Kossmann et al.[5] present an algorithm that finds the skyline with numerous nearest neighbor searches. An

S. Bressan, J. Küng, and R. Wagner (Eds.): DEXA 2006, LNCS 4080, pp. 781 – 790, 2006.

improved approach following this idea appears in[6]. Balke et al.[7] study skyline computation in web information systems, applying the "threshold" algorithm of [8].

Furthermore, we are interested in a class of new applications of skyline queries proposed in the reference [1] whose authors consider the problem of efficiently computing a SkyCube, Which consists of skylines of all possible non-empty subsets of a given set of dimensions. However, the SkyCube will expand exponentially if the data set and the number of dimensions of tuples become more enormous. And unluckily, the SkyCube is can not use further as original data set is changed. So, if usrs want to use this SkyCube again, it will return the wrong result. The most direct method to solve this problem is to try to compute the full SkyCube again. But it is impossible since it is of no effect at all.

In this paper, we propose a novel incremental maintenance algorithm of SkyCube, called *IMASCIR*(Incremental Maintenance Algorithm of SkyCube based on Identify and Refresh). *IMASCIR* splits the maintenance work into two phases: propagate and refresh. Each materialized SkyCube view has its own table which stores the net change to the view due to the change to the original data set. In the phase of propagate, we propagate the source changes to their tables. Then in the phase of refresh, each materialized SkyCube view is refreshed individually by applying its table.

2 Terminology and Notations

In this section, we give several concepts corresponding to Skyline and SkyCube.

Definition 1 (n-dominate). Let Ω is the set of n-dimensional tuples, and there are two tuples: $T(d_{1t}, d_{2t},\ldots, d_{nt})$, $S(d_{1s},d_{2s},\ldots,d_{ns})\in\Omega$. We call T *n-dominate* S(and denoted as $P \succ Q$), if they satisfy as follows($1\le i\le d$):

1) $\forall\, i\ (d_{it}=d_{is} \vee d_{it} \succ d_{is})$;
2) $\exists\, i\ (d_{it} \succ d_{is})$.

Definition 2 (Skyline Set). The *skyline set* of Ω is denoted as $\nabla(\Omega)$, and it satisfy:

$$\nabla(\Omega)=\{\ P\in\Omega \mid \neg\exists\ Q(Q\in\Omega \wedge Q \succ P)\}.$$

In order to discuss SkyCube conveniently, we must extend the domain of each dimension and two elements, i.e. "$\perp$" and "*ALL*", are added in it. "$\perp$" corresponds to empty, and the meaning of "*ALL*" is the same in the reference [9].

Definition 3 (n-dimensioanl Preference Function). Let Ω is the set of n-dimensional tuples, and $\Psi=\{O_1(d_1,d_2,\ldots,d_n),\ O_2(d_1,d_2,\ldots,d_n),\ldots,\ O_k(d_1,d_2,\ldots,d_n)\}\subseteq\Omega$. A function f^R is called a *n-dimensioanl preference function* if it satisfies as follows:

1) $f^R : d_1\times d_2\times\ldots\times d_n \rightarrow R^+$;
2) $\forall O_i,O_k(O_i \prec O_t \Rightarrow f^R(O_i) < f^R(O_t))$。

From definition 4, we see that a n-dimensioanl preference function makes the set of tuples which is out of order into an orderly one. For instance, given $f^R(O_1) < f^R(O_2) <\ldots< f^R(O_k)$, then the object sequence corresponding to f^R is $\Theta<O_k, O_{k-1},\ldots, O_1>$.

Definition 4 (∇ Filter Function). $\hbar_{\nabla} : 2^{d1\times d2\times\ldots\times dn}\rightarrow d_1\times d_2\times\ldots\times d_n$ is called ∇ *filter function* over the power set of $d_1\times d_2\times\ldots\times d_n$(to the set $d_1\times d_2\times\ldots\times d_n$) if only if there exists a function $\hbar_{\nabla}^{assist} : (d_1\times d_2\times\ldots\times d_n)\times(d_1\times d_2\times\ldots\times d_n)\rightarrow d_1\times d_2\times\ldots\times d_n$ (called ∇ filter assistant function) which satisfies:

$$\forall\ \Gamma\in 2^{d1\times d2\times\ldots\times dn},\Gamma^2\subseteq\Gamma,\quad \hbar_{\nabla}(\Gamma)=\hbar_{\nabla}^{assist}(\hbar_{\nabla}(\Gamma^2),\hbar_{\nabla}(\Gamma-\Gamma^2)).$$

Definition 5 (∇ AggregationFunction). *∇ aggregation function* $\xi_{aggr}=\hbar_{\nabla}\oplus f^R$. It combine n-dimensioanl preference function f^R with ∇ filter function $\hbar_{\nabla}$. First, it filters tuples on Ω using $\hbar_{\nabla}$ **,** then makes the remaining tuples orderly using f^R.

Definition 6 (SkyCube). A *SkyCube* is defined as a 7-tuple <ϖ,D,Dom,O,O_ Dom,f, ξ_{aggr}>, where

1) ϖ is an identifier of a SkyCube;
2) D={ $d_1,d_2,\ldots,d_n$}, called the set of identifiers of dimensions;
3) Dom=dom(d_1)×dom(d_2)×…×dom(d_n), called the domain over n-dimensional space;
4) O is object identifiers over n-dimensional space;
5) O_ Dom is the domain of objects over n-dimensional space;
6) f : Dom→O_ Dom is partial mapping over Dom to O_ Dom;
7) ξ_{aggr} is a ∇ aggregation function over Dom.

3 SkyCube Maintenance Algorithm

The SkyCube maintenance problem is the problem of keeping the contents of the materialized SkyCube view consistent with the contents of the original data set as the original data set is modified. Maintenance of a SkyCube view may involve several steps, one of which brings the view table up-to-date. We call this step refresh. A SkyCube view can be refreshed within the transaction that updates the original data set, or the refresh can be delayed. The former case is referred as *immediate SkyCube view maintenance*, while the latter is called *deferred SkyCube view maintenance*.

In this section, we employ the deferred mode, with the source changes received during the day applied to the SkyCube views in a nightly batch window. The work involves updating the original data set, and maintaining all the materialized SkyCube views. During that time the original data set is unavailable to users. The main aim of maintenance is to minimize the batch window.

Based on these points above, we propose a novel incremental maintenance algorithm of SkyCube, called *IMASCIR*(Incremental Maintenance Algorithm of SkyCube based on Identify and Refresh). *IMASCIR* splits the maintenance work into two phases: *identify* and *refresh*. All the materialized SkyCube views share two tables, called δ tables, which stores the net change to the view due to the change to the original data set. In the phase of identify, we identify and store the source changes into δ tables. Then in the phase of refresh, each materialized SkyCube view is refreshed individually by applying these tow δ tables. Identify phase can take place without locking the materialized SkyCube views so that the original data set can continue to be made available for

querying by users. Materialized SkyCube views are not locked until the refresh phase, during which time the materialized SkyCube view is updated from the δ tables. So the identify phase can occur outside the batch window.

3.1 Phase Ⅰ: Identify

Multiple SkyCube views can be arranged into a (partial) lattice. Figure 1 shows a lattice wich corresponds to four dimensions A, B, C, D. And cube ABCD is the ancestor of all other cubes. Obviously, for a set of N-dimensional tuples, there exists at most 2^N -1 SkyCube views. Furthermore, the theorem 1 and the corollary 1 proposed in [1] show that offspring SkyCube views can be computed by ancestor SkyCube views.

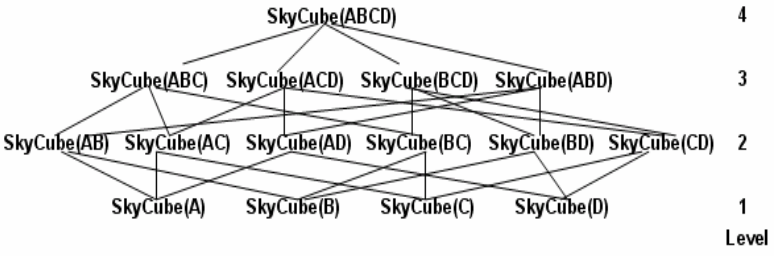

Fig. 1. Lattice Structure of a SkyCube

Theorem 1. Given a set S of data points on dimension set D, U and V are two sub dimension sets (U,V $\subseteq$ D), where U $\subset$ V. On dimension set V, each skyline point q in $SkyCube_u(S)$ is

1) either dominated by another skyline point p in $SkyCube_u(S)$;
2) or a skyline point in $SkyCube_v(S)$.

Corollary 1 (Distinct Value Condition). Given aset S of data points on dimension set D. For any two data points p and q, if $p(a_i) \neq q(a_i)$ ($\forall a_i \in D$), then for two sub dimension sets U and V, (U, V $\subseteq$ D), where U$\subset$ V, $SkyCube_u(S) \subseteq SkyCube_v(S)$.

In this phase (i.e. identify phase), we use two tables which are shared by all the materialized SkyCube views to store the net change to the views due to the change to the original data set. One of these two tables is called insertion δ table (denoted as $\delta^{\triangleleft}$ table), which is used for tuples added into the original data set. Another one is called deletion δ table (denoted as $\delta^{\triangleright}$ table), which is used for tuples removed from the original data set.

It is important to note that in the OLAP applications [10], in order to store the net change, each materialized DataCube view must has its own tables, that is, it needs $2\times|MV|$ tables for all the materialized views. The main reason is that in the OLAP applications, each materialized DataCube view has different net change when the original data set is changed. However, in the SKYLINE query applications, when

the original data set is changed, all the materialized SkyCube views have the same net change. Hence, we only need two shared tables in this phase.

3.2 Phase Ⅱ: Refresh

The phase of refresh can be processed as soon as the first phase finished.

Given a directed acyclic graph G(V,E), then there exists a sequence $v_1<v_2<\ldots<v_t$ which is topologically orderly, that is, v_i precedes v_{i+1}, where $i\in[1,t-1]$. On the other hand, it is obvious that not only SkyCube views lattice but also δ −lattice are the directed acyclic graphs. So, we can refresh offspring SkyCube views after it's ancestor SkyCube views. Our algorithm, i.e. *IMASCIR*, maintain the cube in topological order, hence, we can guarantee that ancestor SkyCube views are "clean" and can be used in the case of recomputation.

In the following part of this section, we'll discuss how to refresh a single materialized view. The procedures to refresh theoriginal data set are rather easy and will not be mentioned here.

For each materialized view, refresh procession includes two procedures as follows:

1) *Ref_Insert*: mainly modifying the SkyCube view with the $\delta^{\triangleleft}$ table.
2) *Ref_Delete*: mainly modifying the SkyCube view with the $\delta^{\triangleright}$ table.

Ref_Insert procedure applies the changes represented in the $\delta^{\triangleleft}$ table to the SkyCube view. Firstly, Ref_Insert procedure compares tuples in the $\delta^{\triangleleft}$ table and filters those tuples which are dominated by the others which are in the $\delta^{\triangleleft}$ table. Secondly, Ref_Insert procedure uses the remaining tuples in the $\delta^{\triangleleft}$ table to filter those tuples in the SkyCube view which are dominated by the remaining tuples in the $\delta^{\triangleleft}$ table. Finally, Ref_Insert procedure modifies the object sequence generated by the n-dimensioanl preference function f^R of this SkyCube view. Ref_Insert procedure describes as follows.

Ref_Insert procedure
  Z := *Skyline_Computing*($\delta^{\triangleleft}$ table);
  Chang from the object sequence $\Theta<O_k,O_{k-1},\ldots,O_1>$ to the set Φ $\{O_1,O_2,\ldots,O_k\}$ using the n-dimensioanl preference function f^R of the SkCube view;
  H=*Skyline_Computing*($\Phi \cup Z$);
  Chang from the set H $\{O_1,O_2,\ldots,O_h\}$ to the new object sequence $\Theta'<O_h,O_{h-1},\ldots,O_1>$ using the n-dimensioanl preference function f^R of the SkCube view;
END

Skyline_Computing (similar to the one proposed in the reference [4])
Input: the set of N-dimensional tuples;
Output: the set which are not dominated by the others which are in the input set;

```
Description:
1. list := Sort_set(Input) using the operator η = Σ_{i=1}^{N}(π_i(O));
2.  Z := ∅;
3.  t := 1;
4.  Z := Z⋃{list[0]};
5.  list.delete(0);
6.  While(list.count>0) do
7.      f := false;
8.      For j := 1 to t do
9.          If (∀w∈[1,N],(π_∂w(O)≥π_∂w(list[i]))  then
10.             f=true;
11.             break;
12.      If (f = false) then
13.          Z := Z⋃{list[0]};
14.          t := t+1;
15.      list.delete(0);
16.  Return(Z).
```

Ref_Delete procedure applies the changes represented in the $\delta^{\triangleright}$ table to all the SkyCube views. According to theorem 2, Ref_Delete recomputes the SkyCube views with the topological order, i.e. $v_1<v_2<\ldots<v_t$ (v_t is the ancestor of all SkyCube views). And it can guarantee all the SkyCube views are correct after processing Ref_Delete procedure.

Theorem 2. Let $G(V, E)$ is a directed acyclic graph, where V={all the SkyCube views} and E={all the edges which are from the parent SkyCube view to the child SkyCube view}. If a tuple is deleted from a child SkyCube view, when recomputing this SkyCube view, we only need to use the tuples which are in it's parent SkyCube view instead of the whole original data set.

For a single SkyCube view Υ, Ref_Delete procedure first compares the the $\delta^{\triangleright}$ table with the SkyCube view, and return immediately if all the tuples which are in the $\delta^{\triangleright}$ table are not in the SkyCube view. However, if a tuple O is deleted from the SkyCube view, some tuples which are dominated by the tuple O originally may become new skyline objects now. So, the part work of Ref_Delete procedure is to process these tuples in it's parent SkyCube view of Υ according to theorem 2. Note that when we compute the SkyCube view γ^{root} which is the root vertex in the SkyCube-lattice (that is, γ^{root} is the ancestor of all other SkyCube views), Ref_Delete procedure must use the original data set since γ^{root} does not have the parent SkyCube view.

Ref_Delete procedure describes as follows.

Ref_Delete procedure

```
1. Chang from the object sequence Θ<O_k,O_{k-1},…,O_1> to the set
   ϕ{O_1,O_2,…,O_k} using the n-dimensioanl preference
   function f^R of the SkCube view Υ;
2. If Υ = γ^root then
     Φ := {the original data tuples};
```

```
3.  Else
4.   Search for the parent SkyCube view of ϒ denoted as γ^Parent ;
     /* the cardinality of γ^Parent is minimal among all the
        parent SkyCube views of ϒ */
5.   Chang from the object sequence Θ'<P_m, P_{m-1}, …, P_1> to the
     set Φ{P_1, P_2, …, P_m} using the n-dimensioanl preference
     function f^R of the SkCube view γ^Parent ;
6.  For each tuple O in the δ^▷ table do
7.    If O in the SkyCube view ϕ then
8.      For each tuple T in Φ−ϕ
9.        If T is not dominated by O then
10.         add t into ϕ;
11. ϕ := ϕ- δ^▷ ;
12. Chang from the set ϕ{O_1, O_2, …, O_h} to the new object sequence
    Θ'<O_h, O_{h-1}, …, O_1> using the n-dimensioanl preference
    function f^R of the SkyCube view.
```

4 Experiments

In this section, we report the results of our experimental evaluation in terms of time for maintaining the SkyCube views.

The databases used in all our experiments are generated in a similar way as described in [2]. We compare our algorithm with the naïve method which processes changes of all the SkyCube views by recomputing the original data set, and use tuples of dimensions 2, 5, 8 and 10. For the sake of simplicity, the type of all the attributes of tuples is integer. Moreover, we let the cardinality of $\delta^{\triangleleft}$ table equals to the one of $\delta^{\triangleright}$ table.

The first case: we fix the cardinality of the original data set (number of tuples=10^5), and the number of refreshing data tuples changes from 5×10^3 to 3.0×10^4. Figure 2 shows the result of the experiment of this case. From Figure 2, we see that the runtime of naïve method is much more than our algorithm, i.e. *IMASCIR*, in particular, the number of dimensions increase to 10. When the number of dimensions of tuples equals to 10, runtime of naïve method is over 13200 seconds since the naïve method processes changes of all the SkyCube views(about 2^{10}-1) by recomputing the original data set. Although the number of SkyCube views is also about 2^{10}-1 in our algorithm, recomputing of each SkyCube view only need the data set of its parent SkyCube view. So, our algorithm saves fairly much time. In Figure 2(d), runtime of *IMASCIR* does not exceed 3000 seconds, and only about 22 percent of the naïve method. Futhermore, we observe that the slope of curve produced by the naïve method equals to 0 since the naïve method always recompute the original data set whose cardinality is a constant (i.e. 10^6) and $|\delta^{\triangleleft}|=|\delta^{\triangleright}|$.

The second case: we fix the size of refreshing data (size=2×10^4), and the cardinality of the original data set changes from 0.5×10^5 to 3×10^5. Figure 3 shows the result of the experiment of this case. From Figure 3, we see that the runtime of naïve method is

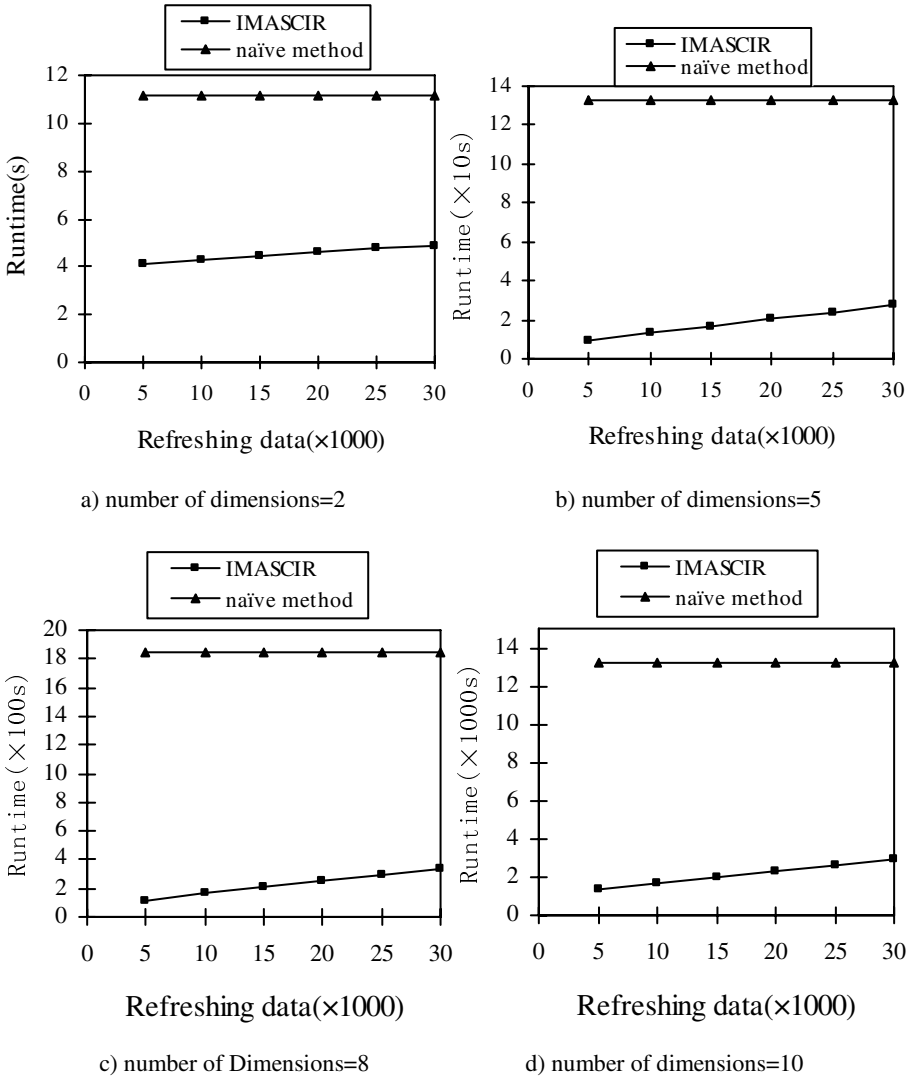

Fig. 2. Runtime vs. the Cardinality of Refreshing Data

much more than our algorithm, i.e. IMASCIR.. And the superiority of our algorithm in comparison with the naïve one is more distinct with the increase of the cardinality of the original data set and the number of dimensions. For example, when the number of dimensions of tuples is 10 and the cardinality of original data set is 3×105, the runtime of naïve method is over 118800 seconds, however, IMASCIR does not exceed 21000 seconds, and only about 17 percent of the naïve method. Furthermore, we observe that the slope of curve of IMASCIR is fairly more small than that of the naïve method.

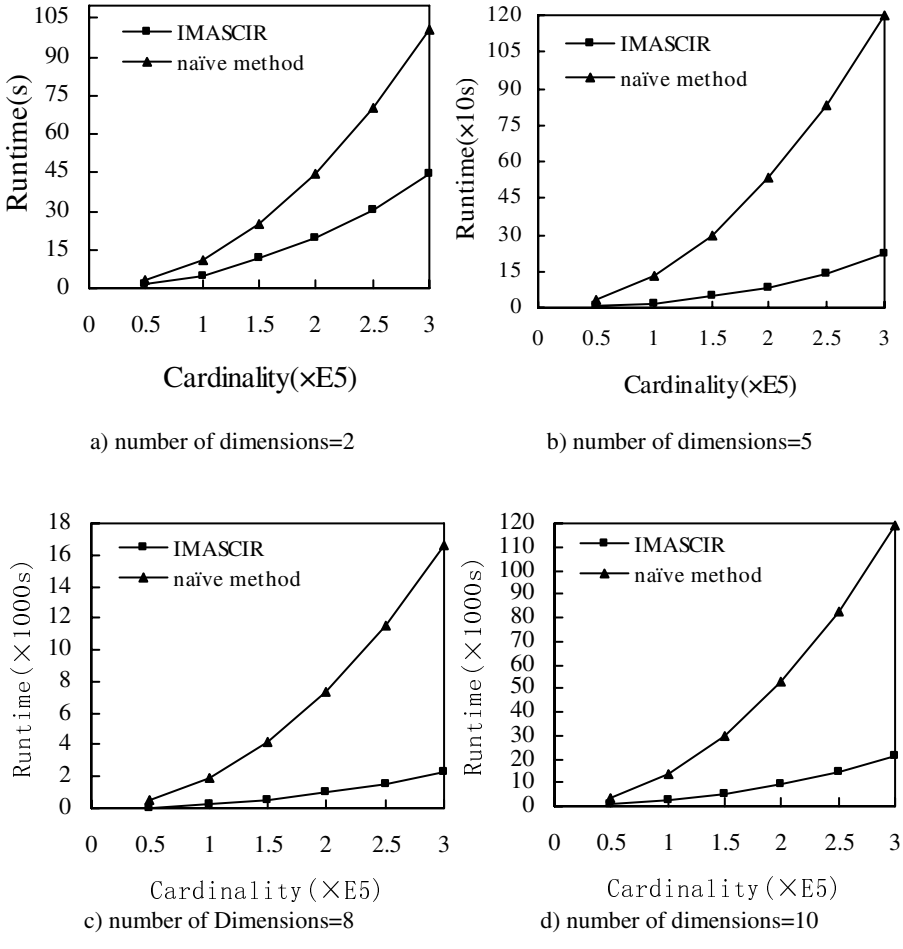

Fig. 3. Runtime vs. the Cardinality of Original Data Set

5 Conclusions

In this paper, we are interested in a class of new applications of skyline queries, i.e. SkyCube. It is obvious that the SkyCube will expand exponentially if the data set and the number of dimensions of tuples become more enormous. And unluckily, the SkyCube is can not use further as original data set is changed. So, if usrs want to use this SkyCube again, it will return the wrong result. The most direct method to tackle this problem is to try to compute the full SkyCube again. But it is impossible since it is of no effect at all. Based on these considerations, we propose a novel incremental maintenance algorithm of SkyCube, called *IMASCIR*. *IMASCIR* splits the maintenance work into two phases: identify and refresh. All the materialized SkyCube views share two tables which store the net change to the view due to the change to the original data set. In the phase of identify, we identify and store the source changes to these two tables.

Then in the phase of refresh, each materialized SkyCube view is refreshed individually by applying these two tables. And our experiment demonstrated that *IMASCIR* is both efficient and effective.

References

[1] Yuan, Y., LIN, X., Liu, Q., Wang, W., Yu, J. X., Zhang, Q.: Efficient Computation of the Skyline Cube. *VLDB*, 2005.
[2] Borzsonyi, S., Kossmann, D., Stocker, K.: The Skyline Operator, *International Conference on Data Engineering ICDE*, 2001.
[3] Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient Progressive Skyline Computation. *VLDB*, 2001.
[4] Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline With Pre-sorting. *International Conference on Data Engineering ICDE*, 2003.
[5] Kossmann, D., Ramsak, F., Preparata, F. P.: Shooting Stars In The Sky : An Online Algorithm For Skyline Queries. *VLDB*, 2001.
[6] Papadias, D., Tao, Y., Fu, G., Seeger, B.: An Optimal And Progressive Algorithm For Skyline Queries. *SIGMOD*, 2003.
[7] Balke, W-T., Guntzer, U., Zheng, J.X.: Efficient Distributed Skylining For Web Information Systems, *EDBT*, 2004.
[8] Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms For Middleware. *PODS*, 2001.
[9] Gray, J., Chaudhuri, S., Bosworth, A.: Data Cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1997.
[10] Mumick, D.Q., Mumick, B.: Maintenance of Data Cubes and Summary Tables in a Warehouse. *International Conference on Data Engineering ICDE*, 1997.