A Data-Centric Approach for Networking Applications

Ahmad Ahmad-Kassem[1], Christophe Bobineau[2], Christine Collet[2], Etienne Dublé[3], Stéphane Grumbach[4], Fuda Ma[1], Lourdes Martinez[2] and Stéphane Ubéda[4]
[1]*INRIA, INSA-Lyon, Villeurbanne, France*
[2]*Grenoble Institute of Technology, Grenoble, France*
[3]*CNRS, Grenoble, France*
[4]*INRIA, Villeurbanne, France*

Keywords: Declarative Networking, Programming Abstraction, Case-based Distributed Query Optimization.

Abstract: The paper introduces our vision for rapid prototyping of heterogeneous and distributed applications. It abstracts a network as a large distributed database providing a unified view of "objects" handled in networks and applications. The applications interact through declarative queries including declarative networking programs (e.g. routing) and/or specific data-oriented distributed algorithms (e.g. distributed join). Case-Based Reasoning is used for optimization of distributed queries by learning when there is no prior knowledge on queried data sources and no related metadata such as data statistics.

1 INTRODUCTION

The trend towards complex distributed applications is accelerated with wireless networking technologies interconnecting an increasing number of heterogeneous devices (mobile and wearable, energy constrained, personalized), which generate large amounts of data. Devices usually take part in dedicated ad hoc networks, where applications deployment, configuration and management are tedious and require significant human involvement and expert knowledge. To improve the application programming there is a need for high-level programming abstraction

Demonstration of portability, extensibility, integration and pervasive adaptation have been done with variants of the Datalog rule-based language applied to express communication algorithms and declarative overlays through queries (Loo et al. 2006; Condie et al., 2008; Chen et al., 2010). Several systems, such as TinyDB (Madden et al., 2005) or Cougar (Demers et al., 2003) sensor network systems, use the relational model to represent device features and application data and offer SQL-like languages to manage data application. These systems also offer solutions to perform efficient data dissemination and query processing (centralized but also distributed).

Our vision – materialized in the UBIQUEST system -- is to go a step further than the promising declarative networking approach providing a unified view of "objects" handled in networks and applications. The environment is perceived as a distributed database and the applications interact through declarative queries (Loo et al., 2006; Mao, 2010). However, some necessary adaptations have to be made: (i) to localize data or define the scope of a query, (ii) to consume, filter and aggregate data (continuous queries), (iii) to consider query operators that may correspond to protocols, (iv) to revisit query optimization. For this later aspect we propose to use Case-Based Reasoning (CBR) – providing a way to optimize queries when there is no prior knowledge on queried data sources and certainly no related metadata such as data statistics. This approach is well adapted to social systems (e.g. games, social networks), where data is pushed or pulled with incomplete knowledge in a dynamic environment.

The paper is organized as follows. Section 2 gives an overview of our approach and Section 3 presents its key elements. Section 4 gives a general presentation of the UBIQUEST system. Section 5 concludes the paper.

Ahmad-Kassem A., Bobineau C., Collet C., Dublé E., Grumbach S., Ma F., Martinez L. and Ubéda S..
A Data-Centric Approach for Networking Applications.
DOI: 10.5220/0004111301470152
In *Proceedings of the International Conference on Data Technologies and Applications* (DATA-2012), pages 147-152
ISBN: 978-989-8565-18-1

2 DATA-CENTRIC APPROACH

The declarative approach is used to abstract the network as a large distributed database providing unified view of "objects" handled by both the networks and applications. Such a database stores information about the declarative programs, routers configuration, states and characteristics of the network. Rule-based programs correspond to network operations or protocols triggered by data updates. Rules are evaluated over local data and may communicate results to other nodes in the network using communication primitives.

The UBIQUEST approach we propose merges the strengths of two areas (i) databases, and (ii) declarative networking. With this approach a programmer can specify the behavior of the system / application (*the what*) rather than having to describe the details of the system (*the how*). This allows going one step further in the overlapping approaches for example with destinations of messages resulting of a query.

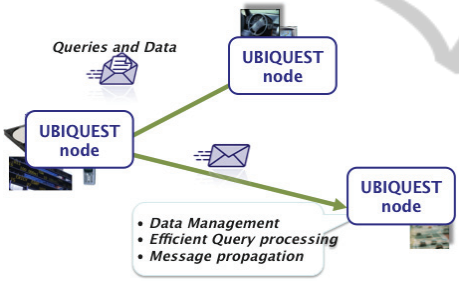

Figure 1: The UBIQUEST approach.

An UBIQUEST system runs on a set of computing devices interconnected through a wireless network (Figure 1). Every device embeds a virtual machine in charge of data management, processing queries (data selection and updates) and messages propagation. The UBIQUEST virtual machine (VM) (see Figure 2) is composed of:

- a Local Data Manager System (DMS) to manage application data, network data and additional information for distributed query evaluation,
- an UBIQUEST Engine to efficiently process queries and rule programs,
- APIs and Communication modules to exchange queries and data between nodes.

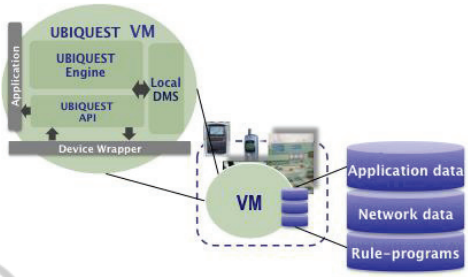

Figure 2: UBIQUEST node components.

All exchanges between nodes related to communication protocols, to resource discovery or to any other applicative aspects are carried out by queries and data. This blurs the traditional distinction between communication and application layers. Queries are defined using either rule-based languages for network data query expressions or declarative query languages for querying application data with a global point of view. Query optimization is based on a CBR-based approach and pseudo-random query plan generation. The CBR paradigm means that we learn the cost of query plans (case) while executing them. These cases are used for generating plans for further similar queries. If there is no convenient case, we use classical heuristics and random choice (e.g. when there is no statistics for join ordering and selection of algorithms).

To illustrate our approach, let us consider an application concerning a virtual world game divided in areas and having some avatars that are located within a single area at a time (see Figure 3). Each node of an UBIQUEST system has information on its own avatars and their neighbors (avatars located in the same area). Such information is stored in an *Itemset* data structure of type:

Positions(Avatar avatar{key}, Int Area, NodeID owner)

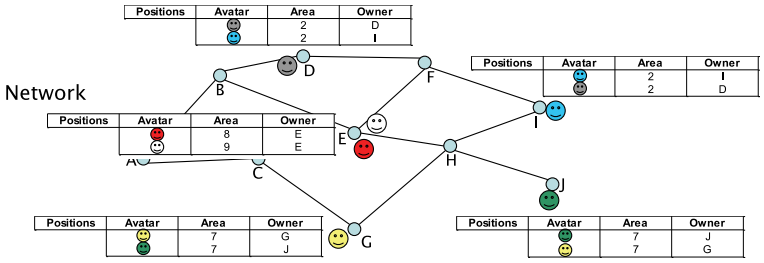

Figure 3: Application scenario.

For example, at node D the Position table (see Table 1) has two tuples showing that it is the owner of the *Grey* avatar that is in the area 2; also the *Blue* avatar is in the same area but owned by node I.

Table 1: **Positions** table at node D.

Positions	Avatar	Area	Owner
		2	D
		2	I

Such a **Positions** table is actually a fragment of a virtual table that maintains the information on all the avatars, giving for each of them, the area in which it is and its owner node. Table 2 gives the virtual global **Positions** table for our application example.

Table 2: Global **Positions** table.

Positions	Avatar	Area	Owner
		7	J
		7	G
		8	E
		2	I
		2	D
		9	E

Considering this global view, the query to select all avatars in the virtual world and the zone where they are located is: SELECT Area, Avatar
FROM Positions;

This query will be executed globally.

Let us assume now that the *Yellow* avatar, owned by node G, is moved from area 7 (where avatar *Green* owned by node J is localized) to area 8 where the *Red* avatar (node E) is localized. Table 3 shows the **Position**s table after this operation.

Table 3: Global **Positions** table after modification.

Positions	Avatar	Area	Owner
		7	J
		8	**G**
		8	E
		2	I
		2	D
		9	E

The movement is coded by several updates executed at node G (owner of *Yellow*) for cleaning area 7, changing the Area attributes of the avatar and finally for storing the new area exploration. The update for cleaning the area 7 is:

```
Delete from Positions
Where Area = (LOCAL Select Area
                from Positions
                where Avatar = 'Yellow')
and     Area not in (LOCAL Select Area
                from Positions
                where Avatar <> 'Yellow'
                and Owner = SELF)
Stored on SELF;
```

The keyword *LOCAL* indicates that the query has to be evaluated by the node over local data only. The sub-queries are local and the delete operation too as it concerns only data stored on SELF. Such a query is executed at the node level and processed in a distributed way with the following principles:

1. No centralized control. Query processing is performed in an environment that is highly dynamic, and has to adapt to and recover from the network evolution. The control needs to be fully distributed over the network.
2. Scarce metadata. The network being highly dynamic, there is no stable knowledge on the data organization. Resource discovery is combined with networking protocols.
3. Everything in the database. The network management is done through queries.

To conclude there is a need for adapting query expression and execution to network condition, application needs and query load.

3 DATA STRUCTURES AND LANGUAGES

Network and application data in UBIQUEST are *Itemsets* (cf. Section 3.1). Data distribution is discussed in Section 3.2 and nodes exchanged messages in Section 3.3. Section 3.4 describes the rule-based languages for writing programs that are installed on each node, where they run concurrently. Finally, Section 3.5 presents the SQL-like query language for describing queries or updates.

3.1 Items and Itemsets

The *item* is the unit of data manipulation: rules (in programs) are evaluated for each new item (new fact), and a query is processed item by item following the classical iterator model. An Item is composed of a set of attribute/value couples, values are taken in predefined data types including *integer*, *float*, *string*, *date* and *NodeId* (node identifier type). The predefined attribute *LocalID* value (of type *NodeId*) is the identifier of the current node.

An *Itemset* is a set defined by a name and the structure shared by its items, i.e. a set of attributes (name, type). Key attributes are used to identify one specific item. An example of *Itemset* is the Positions table in our application example.

3.2 Data Distribution

UBIQUEST supports only horizontal fragmentation

of *Itemsets*. This means that a global *Itemset* is distributed over several (maybe all) UBIQUEST nodes. Any participant node stores a (local) *Itemset* or a fragment with the same schema as the global one. Each item of the global *Itemset* is actually stored in one or more nodes. Data distribution is *application-driven*: applications decide on which node(s) items have to be stored. This is possible using either rule-based programs or *DLAQL* updates (see Sections 3.4 and 3.5).

3.3 Message Structure

A message is the unit of communication among nodes. A message has two main parts: (i) a *networking information* and (ii) a *payload* where the content of the message (i.e. queries or items) is embedded. The *networking information* may contain a *logical destination* of the message defined as a query.

The *payload* has three parts: (i) A *tag* identifying the type of content (e.g. declarative query, query results, facts); (ii) A *ContentId* identifying in a unique way the content; (iii) The *Content* itself: declarative queries or data (query results or facts).

3.4 Rule Languages

The proposed rule-based languages (Netlog and Questlog) extend *Datalog* with communication primitives, as well as aggregation and non-deterministic constructs standard in network applications. The computation of rule programs is distributed and the facts deduced can be either stored locally on an UBIQUEST node on which the rules run, or sent to other nodes.

3.4.1 Netlog

Netlog (Grumbach and Wang 2010) programs are sets of recursive rules of the form *head:-body*, where the head is derived when the body is satisfied. Rules are evaluated in forward chaining and are triggered by insertion of new items in the database. The execution of rules may lead to transmission of items to neighboring node(s). For example, the following Netlog rules that are deployed on all nodes, compute all routes in the network:

↕Route(SELF, dest, dest, 1):- Link(SELF, dest).
↕Route(self, dest, neigh, l2):- Link(SELF, neigh),
Route(neigh, dest, _, l1), l2 := l1 + 1.

The first rule computes trivial routes to neighbors, stores them locally on the relation *Route(Self, destination, nextHop, numberOfHop)*, and broadcasts them to neighbors. When received by neighbors, the second rule is satisfied and new routes with an increasing number of hops is deduced, stored locally, and broadcasted to neighbors. This process continues recursively until getting a fix-point where no new route is obtained.

3.4.2 Questlog

Questlog programs are also sets of recursive rules but evaluated in backward chaining to answer queries coming from a local application or a distant node. The execution may lead to a sub-query emission to neighboring node(s), and may imply the return of items as the result of queries.

For example, the following two rules can be used to compute (on demand) routes between node *orig* and *dest*; *nh* and *l* being free variables.

↕route(@orig, dest, dest, 1) ← link(orig, dest).

If the destination is a neighbour, then the body is satisfied, that results in a route stored locally and sent back to the origin node of the query.

↕route(@orig, dest, nh, l+1) ← ¬link(orig, dest); link(orig, nh); ?route(@nh, dest, _, l).

This rule is applied when the destination is not a neighbor of the node "orig" (*∅link(orig, dest)*). The rule sends a sub-query to all neighbouring nodes to ask them if they know a route to the destination (*Link(orig, nh), ?Route(@nh, dest, _, l)*). The @ operator ahead of a variable represents the node where the sub-query will be sent. When a sub-query returns a result, the second rule is applied recursively to propagate the result to the origin node.

3.5 Data Location Aware Query Language

The Data Location Aware Query Language (DLAQL) extends the well-known SQL2 data manipulation language to conform to the data distribution policy of UBIQUEST. This means that a DLAQL expression may explicitly indicate on which UBIQUEST node data has to be stored.

3.5.1 DLAQL as a Subset of SQL2

With DLAQL, one can express classical SELECT-FROM-WHERE queries using nested sub-queries, aggregation functions, arithmetic expressions, and selection, join and union operations. However, SELECT expressions in DLAQL cannot use: group by/having clauses, nested queries except in the WHERE clause, synchronous sub-queries, and

EXISTS and UNIQUE condition operators.

3.5.2 Scope of DLAQL Queries

A *DLAQL* expression is defined considering a global schema of *Itemsets*. It is evaluated considering the value of the *Itemsets* (union of the fragments). The *LOCAL* clause can be used to indicate that only local data has to be used to evaluate conditions (*WHERE* clause). Furthermore, if *LOCAL* is used with update queries, modifications are applied on local *Itemset* fragments only. For example, the following query updates only local avatars of the Positions *Itemset* that are located in *Area* number 5.

LOCAL UPDATE Positions SET …
WHERE Area = 5;

3.5.3 DLAQL and Data Locality

The *STORE ON* clause can be used in *INSERT* / *DELETE* / *UPDATE* query expressions to indicate where the data has to be inserted, or where is the deleted or updated data. The clause specifies a list of node identifiers that are explicitly given (values of type *NodeIds)* or calculated as the result of a query.

For example, the following DLAQL query inserts the new item *('MyAvatar', 5, SELF)* in the Positions *Itemset* stored at the local node *SELF* (the current node where the query is executed) and in the Positions *Itemset* of any node owning an avatar in the area number 5.

INSERT INTO Positions
VALUES ('MyAvatar', 5, SELF)
STORE ON SELF, (SELECT Owner FROM Positions WHERE Area = 5);

4 THE UBIQUEST SYSTEM

An UBIQUEST node is a device equipped with an UBIQUEST Virtual Machine (UBIQUEST VM) complemented with a Device wrapper that allows device/VM interaction. The UBIQUEST VM is composed of: (i) a Local DMS, (ii) an UBIQUEST API, and (iii) an UBIQUEST Engine responsible of efficient execution of UBIQUEST programs/queries.

4.1 Local DMS

The Local DMS stores and manages data as *Itemsets*: application data (e.g. sensed data), network data (e.g. routing tables, neighbor table), rule-programs (e.g. distributed algorithms that can be dynamically loaded/removed to/from the system), and internal data (e.g. device specific data) used for running other UBIQUEST VM components.

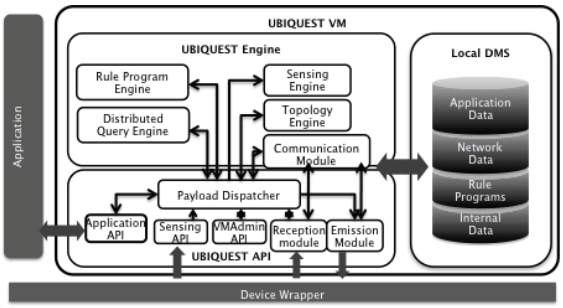

Figure 4: UBIQUEST node components.

4.2 UBIQUEST API

The UBIQUEST API manages all interactions between the UBIQUEST Engines and the rest of the world: local applications, device sensors and other UBIQUEST VM through message exchange

As shown in Figure 4, the API is composed of: (I) the *Application API*, in charge of the interaction with applications running on the local node, (ii) the *Reception* and *Emission* modules to deal with message exchange among UBIQUEST nodes, (iii) the *Sensing API* that locally stores data coming from sensors embedded in the physical device, and (iv) *the Payload Dispatcher*, which manages *Payload* exchange among UBIQUEST VM sub-components according to their *Tag*.

4.3 Distributed Query Engine

The *Distributed Query Engine* evaluates DLAQL queries. It is composed of: (i) a *Query Scheduler*, (ii) a *Query Optimizer* and (iii) an *Execution Engine*. The *Query Scheduler* rewrites a global query into a set of sub-queries and schedules their evaluation (e.g. an update global query is decomposed into a sequence of select, delete and insert sub-queries to read the old value, delete it and inserting the new value). Moreover, this module rewrites a query considering local and distant *Itemset* fragments generating a query (or set of queries) equivalent to the original one.

The *Query Optimizer* is based on the Case-Based Reasoning (CBR) machine learning approach (Stillger, Lohman, Markl and Kandil, 2001). It retrieves and adapts query plans using the experiences gained from the execution of past similar queries. When no knowledge is available it randomly generates query plans using classical heuristics (Ioannidis 1996). Query plans are trees

with computation operators, classical data access operators or rule-based program invocations.

The Execution Engine executes a query plan P using the well-known Iterator model (Graefe 1993) for the physical operators. It also coordinates the local and distant sub-queries and constructs a final result from sub-query results. During the execution, the cost parameters (energy, time, memory etc.) are measured and a new case is built.

4.4 Rule Program Engine

The *Program Engine* receives payloads from the *Payload Dispatcher* and has to treat their *Contents* containing items (new facts or query results) or predicates corresponding to a query.

If a *Content* contains new facts, the *Program Engine* identifies which rule-program has to be triggered by comparing new facts with predicates in the rule head, then it retrieves the corresponding rules from the DMS and evaluates them in forward chaining mode till a fixpoint is reached.

If a *Content* contains a predicate corresponding to a query, *the Program Engine* identifies which rule-program has to be triggered by comparing the predicate with rule bodies, then it retrieves the corresponding rules from the DMS and evaluate them in backward chaining till the full query result is computed. If a *Content* contains query results, these results are exploited to continue query evaluation.

In addition, the *Program Engine* during processing propagates new items or new queries to other nodes, through the UBIQUEST API, and/or stores new items in the DMS. The *Program Engine* has some additional functions, such as timers, necessary for networking protocols, it also uses optimization techniques, such as the triggering of rules by new facts, which avoid unnecessary computations, when there are no changes in the input of rules.

5 CONCLUSIONS

This paper proposes a unified abstraction for the management of application and network data, which abolishes the separation between application and communication layers. Applicative data, network management operations, even configuration are treated as transactional queries or updates. The integration between queries and communication protocols is one of the fundamental contributions of the approach.

This allows the definition of rule programs for networking protocols, which optimize queries or query optimization strategies, which optimize network distribution.

We are currently working on a better integration of the two kinds of languages and on the implementation of an UBIQUEST system prototype. We also develop a simulation and emulation environment for a detailed analysis and evaluation of queries for a large class of algorithms and protocols.

ACKNOWLEDGEMENTS

This work has been supported by the ANR-09-BLAN-0131-01 UBIQUEST Project (http://ubiquest.imag.fr), financed by the French National Research Agency (ANR).

REFERENCES

Chen, X., Mao, Y., Z. Mao, M., Van der Merwe, J., 2010. Decor: Declarative network management and operation. *SIGCOMM Comput. Commun. Rev.*, 40:61-66.

Condie, T., Chu, D., Hellerstein, J. M., Maniatis, P., 2008. Evita raced: metacompilation for declarative networks. *Proc. VLDB Endow.*, 1:1153-1165.

Demers, A. J., Gehrke, J., Rajaraman, R., Trigoni, A., Yao, Y., 2003. The cougar project: a work-in-progress report. *SIGMOD Record*, 32(4):53-59.

Graefe, G. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys*, vol. 25, Issue 2.

Grumbach, S., Wang, F., 2010. NetLog, a rule-based language for distributed programming. *In M. Carro and R. Pea, editors, PADL, volume 5937 of Lecture Notes in Computer Science*, pages 88-103.

Ioannidis, Y., 1996. Query optimization. *ACM Comput. Surv.*, 28(1):121-123.

Loo, B. T., Condie, T., Garofalakis, M. N., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I., 2006. Declarative networking: language, execution and optimization. *In ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, USA.

Madden, S., Franklin, M. J., Hellerstein, J. M., Hong, W., 2005. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1).

Mao, T., 2010. On the declarativity of declarative networking. *SIGOPS Oper. Syst. Rev.*, 43:19{24}.

Stillger, M., Lohman, G., Markl, V., Kandil, M., 2001. Leo - db2's learning optimizer. *In: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 19-28, San Francisco, CA, USA.

Skyline Query Processing on Heterogeneous Data
A Conceptual Model

Nurul Husna Mohd Saad, Hamidah Ibrahim, Fatimah Sidi and Razali Yaakob
Department of Computer Science, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 UPM, Serdang, Selangor, Malaysia
nhusna.saad@gmail.com, {hamidah, fatimacd, razaliy}@fsktm.upm.edu.my

Keywords: Data Heterogeneity, Probabilistic Skyline Query, Data Management.

Abstract: Skyline queries have been aggressively researched recently due to its importance in realizing useful and non-trivial application in decision-making environments. However, existing researches so far lack methods to compute skyline queries over heterogeneous data where each data can be represented as either a single certain point or a continuous range. In this paper, we tackle the problem of skyline analysis on heterogeneous data and proposed method that will reduce the number of comparisons between objects as well as gradually compute the probabilistic skyline of each object to be a skyline object. Our model employs two techniques (*local pruning* and *global pruning*) for probabilistic skyline query.

1 INTRODUCTION

In recent years, preference evaluation methods have gained a tremendous popularity amongst the database research community as they have been found to be useful in decision-making environments. Almost everybody, either in their daily lives or in professional scenarios, will face with multiple conflicting criteria that need to be evaluated in order to make a decision. Users will find that cost or price of an item or service is usually the main criteria to be considered in any decision makings and most of the time, it will be in conflict with some other measure of quality, which is also typically another criterion in making the decisions. The most prominent example used in this research area is hotel selection. In selecting a hotel, price, distance, and rate may be some of the main criteria a user might consider. Users mostly want a hotel that is cheap and near to certain places, i.e., beaches. It is rare to have the cheapest hotel to be the nearest hotel to the beach. Thus, this is where the implementation of preference evaluation methods will benefit the user as it leads the process of decision makings to more informed and better results.

Skyline query is one of the most popular preference evaluation methods that had been receiving various interests in the literature (Börzsönyi et al., 2001; Papadias et al., 2003; Kian-Lee et al., 2001; Kossman et al., 2002; Godfrey et al., 2005; Lee et al., 2007). Skyline queries return a set of interesting multi-dimensional (multiple-criteria) objects. In n-dimensional objects, we say object U is more interesting than another object V if U is better than or equal to V in all dimensions and U is also better than V in at least one dimension. Normally, it will be assumed whether lower or higher value is preferred for all dimensions. However, most researches that have been done in this area have only been focusing on homogeneous data, but in a real world application, the existence of heterogeneous data could not be avoided. The study of heterogeneity of data in skyline queries would usually by default make this study fall under the scope of uncertainty in skyline queries. Nevertheless, in our study so far, we have found that this is not the case as the study of data uncertainty in skyline queries did not fully incorporated the heterogeneity of data in their research. Consider Figure 1 which shows examples of homogeneity of data in the studies of skyline queries for both certain and uncertain data. Both of the data shown in the figure obviously have the same structure of data in both dimensions x and y, and while the data in Figure 1b did not have the same structure in both of its dimensions, still, all the data in dimension x have the same structure, thus making the process of skyline queries on this data quite straightforward (although, it is still not as straightforward as the conventional skyline query processing on certain