2008

Chameleon: Context-Awareness inside DBMSs

Hicham G. Elmongui

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Mohamed F. Mokbel

Report Number:
08-028

Chameleon: Context-Awareness inside DBMSs

Hicham Elmongui
Walid Aref
Mohamed Mokbel

Chameleon: Context-Awareness inside DBMSs

Hicham G. Elmongui*, Walid .G. Aref*, Mohamed F. Mokbel**

*Department of Computer Science, Purdue University

{elmongui, aref}@cs.purdue.edu

**Department of Computer Science and Engineering, University of Minnesota

mokbel@cs.umn.edu

October 17, 2008

Abstract

Context is any information that can be used to characterize the situation of an entity. Examples of contexts include, but are not limited to, time, location, identity, and activity of a user. This paper proposes a general context-aware DBMS, named Chameleon, that will eliminate the need for having specialized database engines, e.g., spatial DBMS, temporal DBMS, and Hippocratic DBMS, since space, time, and identity can be treated as contexts in the general context-aware DBMS. Moreover, in Chameleon, we will be able to combine multiple contexts into more complex ones using the proposed context composition, e.g., a Hippocratic DBMS that also provides spatiotemporal and location contextual services. As a proof of concept, we construct two case studies using the same context-aware DBMS platform within Chameleon. One case study treats identity as a context to realize a privacy-aware (Hippocratic) database server while the other case study treats space as context to realize a spatial database server using the same proposed constructs and interfaces of Chameleon. To enable context-awareness and to be able to combine contexts, we need to realize various context-aware operators. We introduce a skyline operator as a necessary and important operator for efficient combination of contexts within a context-aware query processing engine. We study and address the performance bottlenecks that result from the interaction between the skyline operator and other query operators, e.g., joins. Although there is still a long way to go, the two case studies and the proposed skyline operators demonstrate that context-aware DBMSs are a viable and scalable approach.

1 Introduction

According to the Merriam-Webster Online Dictionary, the term "context" is defined as *the interrelated conditions in which something exists or occurs* [1]. Many researchers have tried to define context. However, their definitions are

1

often by using examples of context (e.g. [41]) or by using synonyms [3, 40]. In the Ubiquitous Computing community, context is defined by Dey and Abowd as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" [14]. Examples of contexts include, but are not limited to, time, location, identity, and activity of a user.

Context-aware computing (introduced in [41]) describes a system to be context-aware if "it uses context to provide relevant information and/or services to the user, where relevancy depends on the users task" [14]. Context-aware personalization is about tailoring services to better adapt to user preferences with knowledge about the user and his/her environment.

In this paper, we introduce *context-awareness inside DBMSs*. In contrast to all existing work about context aware Systems that are built on top of an infrastructure that provides just the data (e.g., database or data stream), we propose to incorporate the context awareness inside the DBMS. Several specialized DBMSs already exist that manage data and answer queries related to one context type. For instance, a spatial DBMS is optimized to manage and query objects in space. A temporal DBMS has a built-in temporal data model. Tailoring such database engines is not an easy task. Moreover, supporting and combining multiple contexts in one tailored database engine is not within reach. By including the concept of contexts inside the DBMS, we will not need to tailor specialized engines towards a certain context (let alone multiple contexts), but rather we will be able to have systems that support user-defined complex and composite contexts.

This paper presents the design and implementation of Chameleon, a Context-Aware DBMS. Chameleon[1] supports multiple contexts as well as user preferences and has a generic interface to define and process the context information. Contexts in Chameleon are classified according to their properties. As a proof of concept, Chameleon is used to support basic contexts, such as location and identity and show how to compose complex contexts from basic contexts.

The composition of multiple contexts may involve imposing a certain order on the retrieved data. There are many possibilities to produce the desired order for the resulting multi-dimensional data. It may be as simple as ordering the data based on one dimension, and in case of ties using the other dimensions. Another possibility is to use ranking algorithms, where all dimensions are comparable and can be reduced to one-dimensional value using a scoring function (e.g., weighted average). In the case that the dimensions cannot be aggregated (e.g., price and distance), skyline queries give good insights on the data. Incorporating the first two options inside DBMSs is already underway. However, there is no previous work in integrating skyline queries at the core of query operators or database systems. We introduce and study the performance of a new SkylineJoin pipelined query operator that is built inside Chameleon

---

[1]The *real* Chameleons also change their color and appearance based on the context they are in.

to produce the skyline of data whose dimensions are stored in different tables. SkylineJoin is essential for efficient query processing when combining multiple contexts.

The contributions of this paper are as follows:

- We introduce new SQL constructs to define general contexts within database management systems. This generic context definition is suitable not just for expressing user preferences, but also for other contexts such as identity, location, and time.

- We demonstrate how the proposed SQL constructs can be combined to form complex and composite contexts, e.g., how to instantiate a database management system that is aware of both the user location, its identity, as well as its preferences.

- We give the conceptual evaluation of the proposed context-aware DBMS. This conceptual evaluation shows how queries will be evaluated according to the user/data context.

- We realize all the proposed techniques within Chameleon, a new prototype context-aware database management system based on PostgreSQL. We report on the experimental evaluation of this realized system.

- We present two case studies, Hippocratic databases and spatial databases, to show ease and feasibility of instantiating tailored systems using Chameleon. The added value is in the ease of realization of contexts, the ability to combine multiple contexts within the same system, and in processing context-aware queries efficiently.

- We introduce a new SkylineJoin operation as a necessary and important operation for the efficient composition of contexts, and hence, for the realization of context-aware query processing engines. We design a pipelined query operator to realize this operation. The experimental results show that SkylineJoin outperforms the traditional skyline algorithms by at least 300%.

This rest of the paper is organized as follows. In Section 2, we present a classification of contexts. Section 3 illustrates the different constructs needed to realize a context-aware DBMS. Section 4 presents the conceptual evaluation of SQL queries that use the new context constructs. Two case studies for realizing privacy-aware (Hippocratic) databases and spatial databases using Chameleon are presented in Section 5. In Section 6, we present the new SkylineJoin operator and its logic. Section 7 provides experimental evaluation. Section 8 presents the related work, and we conclude the paper with Section 9.

1.1 Running Example

In this section, we describe a running example that we use to demonstrate Chameleon's proposed syntax and semantics. This example is a simplistic preference-based system for ease of presentation. *Still, we apply sophisticated*

row	title	year	category	cover	stock
1	book01	2004	Science Fiction	HC	√
2	book02	2002	Travel	PB	√
3	book03	2001	Medicine	HC	×
4	book04	2000	Cooking	PB	√
5	book05	1997	Science Fiction	PB	√
6	book06	2001	Medicine	HC	√
7	book07	1995	Cooking	PB	×
8	book08	1996	Travel	PB	√
9	book09	2000	Science Fiction	PB	√
10	book10	2003	Medicine	PB	√
11	book11	2005	Travel	HC	√
12	book12	2006	Cooking	HC	×
13	book13	2004	Medicine	PB	√
14	book14	2006	Science Fiction	HC	√
15	book15	2005	Travel	HC	√
16	book16	2006	Cooking	HC	√
17	book17	1976	Medicine	PB	√
18	book18	2001	Travel	PB	√
19	book19	2007	Science Fiction	HC	√
20	book20	1988	Cooking	PB	×
21	book21	1993	Science Fiction	PB	√
22	book22	2006	Medicine	HC	×
23	book23	1999	Cooking	PB	×
24	book24	2006	Medicine	HC	√
25	book25	2006	Travel	PB	√
26	book26	1988	Cooking	HC	√
27	book27	2004	Science Fiction	PB	√
28	book28	2001	Travel	HC	√
29	book29	2004	Medicine	PB	×

Table 1: The books table

case studies to realize a spatial DBMS and a Privacy-aware DBMS using the same context definition constructs in Section 5. Table 1 gives a projection on the table "books" that contains information about books in a certain bookstore. Among other pieces of information included in this table, we can find the name of a book, the years of its publishing, the category under which this book falls, the type of the cover (HC for hardcover or PB for paperback), as well as whether or not the book is in stock. The scenario in consideration consists of a user asking about the books available in a certain bookstore. According to her preferences, she gets different results.

2 Classification of Contexts

Two main entities are involved in a context-aware database system. The first entity is the query issuer; the user. The second entity is the data being queried. Both these entities may have their own contexts.

$$\text{Context type} \begin{cases} \text{object context} \\ \text{user context} \end{cases}$$

2.1 Object Context

The object context is the context of an entity. Considering houses, the context might be the location of a house, its price, its color or the list of those who are interested in buying it. Reflecting on books, the book context might be the library in which one may find it. It might also be the number of its copies in stock in a certain bookstore.

The object context might be an attribute, or a set of attributes, already existing along with the other attributes of the object. Nevertheless, the object context might not be included with the object's other attributes in the same relation, or even one might not have enough privileges to alter the entity's relation to include it. In such cases, the object context will need to be defined, and instantiated, as we show in the constructs that are needed to enable context awareness.

2.2 User Context

User context refers to the context of the query issuer. It can be its location, its identity, or its preferences. In fact, the user context may be any information relevant to the user. When an object context conforms with the user context, this object is returned when its table is queried.

We classify user contexts according to three dimensions. These dimensions will be used when the application developer defines a context in Chameleon. These dimensions will reflect in the access method selection of any query on the tables that are affected by that context.

Dimension 1: Context Sign

The sign of a user context is either "positive" or "negative". Positive context defines what the context *is*. For instance, if the context is location, an instance of a positive context is the preferred locations by the user, e.g., specified as a range. On the other hand, a negative context defines what the context *is not*. An instance of a negative location context is the locations or regions not desired by or prohibited to the user.

In the running example, an instance of a positive context is the willingness to buy hardcover books only. However, trying to avoid science fiction books is a case for a negative context.

$$\text{Context sign} \begin{cases} \text{[S] positive} \\ \text{[G] negative} \end{cases}$$

Dimension 2: Contextual Relation

The contextual relation is the relation among the contextual data. This relation mainly shows the order of relevance of the contextual data. The contextual relation can be an *equivalence relation*. In this case, data that comply with all contextual values are reported with no special ordering. Besides, the contextual relation can also be a *total ordering relation*. This relation would reflect on the data being reported to the user. The data will be sorted on the rank of the contextual values with which the data conform. Moreover, the contextual relation can be a *partial ordering relation*. In contrast to the previous relation, the rank of the contextual values here will follow a partial order rather than a linear order.

Referring to the books table, an example of an equivalence relation is the equal willingness to buy a science fiction book or a travel book. However, if the user is interested in new books, a total ordering relation would be much appropriate to retrieve the latest books first. If the user prefers cooking books over science fiction books, and travel books over medicine books, with no specific preference among the other combinations, she would need to specify her context to contain partially ordered contextual values. Partially ordered values may be transformed into linear ordered values using an appropriate (possibly online) topological sort algorithm. This is out of the scope of this paper, but we add the modeling part here for completeness.

$$\text{Contextual relation} \begin{cases} \text{[Q] equivalence relation} \\ \text{[T] total ordering relation} \\ \text{[P] partial ordering relation} \end{cases}$$

Dimension 3: Listing of Data

By listing of data we refer to how the data should be listed. Specifically, should the data that does not conform to the user context be excluded from the listed data? Or, should those data be included but come last? The former case is termed "unlisted excluded", whereas the latter is termed "unlisted included".

Consider the bookstore example, if the user context is the willingness to buy travel books only, the user context gets the unlisted (other book categories) excluded. Nevertheless, an unlisted included context can be illustrated by the preference to buy hardcover books but still get the paperback books down in the list — after retrieving all hardcover books).

In the location context example, if the user context is the willingness to buy houses that lie within a certain geometric region, say r, then "unlisted excluded" means that houses outside r are not reported to the user, whereas "unlisted included" lists the houses outside r after listing the houses inside r.

$$\text{Listing of Data} \begin{cases} \text{[X] unlisted excluded} \\ \text{[N] unlisted included} \end{cases}$$

2.3 User Context as a 3D Point

Each user context is looked upon as a point in the 3D space defined above. For instance, in the bookstore example, one might be willing to buy only science fiction or travel books with no particular preference between these two types. This is an example of a positive user context having an equivalence contextual relation with the unlisted contextual values excluded. Whenever a user with the aforementioned context selects all tuples from the table books, only rows 1, 2, 5, 8, 9, 11, 14, 15, 18, 19, 21, 25, 27, and 28 are retrieved. If the user defines the same context to have a total ordering relation instead of an equivalence relation such that science fiction have higher rank than travel books, the retrieved rows will be: 1, 5, 9, 14, 19, 21, 27, 2, 8, 11, 15, 18, 25, and 28.

All points in this 3D space are valid when the user context is positive. However, when the user context is negative, only contextual values with the unlisted included are valid. This restriction is due to the definition of a negative user context; user is specifying what context values are not current, and hence all the others should be current (or nothing will be ever returned). Moreover, for a negative user context, since the user only describes the complement of her positive context, no rank is explicitly specified for that actual positive context. Therefore, the equivalence relation would be implicitly understood for the contextual values.

3 SQL Constructs for Context Awareness

In this section, we describe the different constructs that are needed to enable context awareness inside a DBMS. Chameleon provides support for all these constructs.

Creating Object Contexts

Chameleon uses the `CREATE OBJECT CONTEXT` statement to define an object context. Note that when the object context is part of the object relation, one does not need to define it explicitly.

```
CREATE OBJECT CONTEXT contextname (
        {col_spec | table_constraint} [, ...]
        , table_binding
);
```

Contextual values will be stored inside relations to be easily incorporated within the query processor. The `CREATE OBJECT CONTEXT` statement has similar constructs to those in `CREATE TABLE` statement. For instance, col_spec refers to the specification of a column such as name, data type, default values, and so on. On a similar vein, table_constraint refers to any constraints on the whole context table such as check constraints.

The construct table_binding is the main construct that connects the object with its context. Specifically, table_binding has the format below.

```
BINDING KEY ([col_name [, . . . ]])
      REFERENCES ref_table [( ref_col [, . . . ] )]
      WITH bool_expr
```

The first part of the `BINDING KEY` is similar to the `FOREIGN KEY`. There are three main differences between these two types of keys. The first difference is that a foreign key in a table has to reference to a *primary key* in another table. This constraint does not exist for the binding key. A binding key binds the contextual value to possibly more than one object, since more than one object may exist in the same context. The second difference is that the decision to bind a contextual value with an object does not have to be an equality with a column value in the referenced table. The `WITH` construct defines a Boolean expression that helps as the binder in case the expression evaluates to *true*. The third difference is that the binding key might not contain any context attribute referencing an attribute in the base table, but rather only the Boolean expression that might also contain attributes from any object context to the referenced table. Examples will be shown in the case studies section to illustrate this further.

Creating User Contexts

Similar to object contexts, each user context will materialize to a relation. Chameleon uses the following syntax to define a user context.

```
CREATE [context_sign] CONTEXT contextname (
        {col_spec | table_constraint} [, . . . ]
        , table_binding [, . . . ]
        , substituting_key [, . . . ]
) [AS contextual_relation_clause]
[WITH UNLISTED unlisted_status];


context_sign: positive
            | negative
contextual_relation: equivalence
                   | total order [USING ordering_func]
                   | partial order
unlisted_status: excluded
               | included
```

For each table affected by a user context, a binding key is used to show how the context reflects on the table. Therefore, there might be more than one binding key in a user context.

Upon the creation of a user context, an implicit column is created to hold the *user_name* of the current user. Therefore, each contextual value is associated with a certain user. Also, if an ordering relation is used for the contextual relation, then another implicit column is created to hold the *rank* of that contextual value. This rank can be either input by the application while acquiring contex-

tual data, or it can be computed using an ordering function *ordering_func*. In the latter case, the rank column does not need to exist.

Chameleon builds default indexes for context relations. Object contexts get non-clustered indexes on the context keys. User contexts are clustered in a B-tree index using the clustering key (user_name, context_key) if the contextual relation is equivalence. If the contextual relation is a total ordering relation, then the user context is clustered on (user_name, context_key) if the unlisted are to be included and on (user_name, rank) if the unlisted are to be excluded.

The substituting_key will be discussed in details in the next construct. Populating the contextual relations will be made using standard SQL `INSERT` statements. Also, other data manipulation statements will still work on the contextual relations.

Global Substitution Construct

Some attributes need to be modified for presentation purposes if we want to enable context awareness. For instance, if the context is the location of a user, and the user is currently in France, then we might want all prices, in all tables, to be converted to Euro. This conversion is called *global substitution*, since the substitution occurs for all tables according to the current context. The substituting key defines such conversion, and is defined while defining the user context as follows.

`SUBSTITUTE` table_name(col_name)
        `BY` expression;

The expression that substitutes the attribute can be any expression in which attributes from table_name, its object contexts, as well as the user context may appear.

Setting Active Contexts

The application user may have many contexts, not all of them need to be current all the time. Therefore, we introduce the construct `SET ACTIVE CONTEXT` to define the current contexts to be taken into account for that user. The user_name has the CURRENT_USER as a default.

`SET ACTIVE CONTEXT [FOR USER` user_name]
`AS` context_name [, ...];
{ [`WITH RANKING ORDER` context_name [, ...]]
  | [`WITH RANKING EXPRESSION` expression
  | [`WITH SKYLINE OF` expression {`MAX`|`MIN`} [, ...]] };

The `SET ACTIVE CONTEXT` context provides for composing complex contexts from basic ones. If all the basic contexts, which are used to compose a bigger context, have equivalence contextual relations only, then the order of executing the contexts is given by the order the contexts are listed in the `AS` clause.

We provide three mechanisms for ordering in case we have more than one context that is trying to impose an ordering on the data. The first mechanism

is to sort the column based on a context, and then use the following context in the case of ties, and so on. The `WITH RANKING ORDER` clause is used to provide such order of contexts.

On the other hand, for some applications, this type of sorting does not have a clear physical meaning. Some applications need the data to be sorted based on a ranking function that combines all ranks of individual contexts. The `WITH RANKING EXPRESSION` clause is used to provide such ranking expression or function. Our previous work [23] presents an adaptive execution of RankJoin, which provides for retrieving the data sorted based on a ranking function. The ranking function includes ranks coming from different relations that are also joined as in our case.

Nevertheless, such a ranking function does not always exist or such function might not have a reasonable meaning. This occurs especially when the ranks of different contexts are inversely correlated. As a result, we provide for the third context composition mechanism that retrieves the tuples that form the skyline of the data. The skyline operator [2] might be used for that purpose. In section 6, we introduce a new operator SkylineJoin, which not only provides the skyline of the data but also seizes the opportunity that the ranks come from different tables to produce the results fast. The `WITH SKYLINE` clause is used to specify to the skyline operation which expressions to use as the input ranks in the computation.

4 Conceptual Evaluation

In this section, we show why the above constructs enable context-aware query processing. We continue with our running example where someone is accessing the database of a bookstore. Only the books in stock that are relevant to the user's context are retrieved. Examples of contexts are given, their definition using the above constructs are provided, and then we show how they are evaluated to give the desired results. First, we start by simple contexts, and later we show how these contexts are combined together to compose complex contexts. In all the scenarios below, the user is executing the query in Table 2, and the results are the relevant tuples.

```
SELECT *
FROM books
WHERE books.stock;
```

Table 2: Query Q_u issued by the user at the bookstore

Context 1 *The user has a preference for only books of a certain category (e.g., Science fiction).*

This context may be defined as:

```
CREATE POSITIVE CONTEXT ctxt_category_SQX (
      category varchar(20),
      BINDING KEY (category)
                  REFERENCES books(category)
) AS EQUIVALENCE WITH UNLISTED EXCLUDED;

SET ACTIVE CONTEXT AS ctxt_category_SQX;
```

We give the suffix `SQX` to the context name above to emphasize that it is a positive `[S]` context with an equivalence `[Q]` contextual relation and that the unlisted categories in the context are to be excluded `[X]`. For the above example, when the user issues Q_u above, the actual query that is executed is given below. Typically, the binding key is used to join the books table with the context table, and only the books whose category exists in the context are to be returned.

```
SELECT T.*
FROM books T
    INNER JOIN ctx_category_SQX C1
        ON(T.category = C1.category
        AND C1.user_name = CURRENT_USER)
WHERE T.stock;
```

Context 2 *The user's preference is for books published in 2005, and then those published in 2006 before all other books.*

This context may be defined as:

```
CREATE POSITIVE CONTEXT ctxt_year_STI (
      year integer,
      BINDING KEY (year) REFERENCES books(year)
) AS TOTAL ORDER WITH UNLISTED INCLUDED;

SET ACTIVE CONTEXT AS ctxt_year_STI;
```

Again, the suffix `STI` of the current context emphasizes that it is a positive `[S]` context with a total order `[T]` contextual relation and that the unlisted years in the context are to be included `[N]`. For the above example, in response to Q_u, the actual query that is executed is given below. Typically, the binding key is used to join the books table with the context table. In this case, the type of join is a left outer join, and therefore, all books will be returned at the end. The output rows are to be sorted based on the year *rank*, which is specified implicitly in the context as it is an ordering context. Rows with NULL context rank appear later in the list.

```
SELECT T.*
FROM books T
    LEFT OUTER JOIN ctx_year_STI C1
        ON(T.year = C1.year
```

```
         AND C1.user_name = CURRENT_USER)
WHERE T.stock
ORDER BY C1.rank;
```

Context 3 *The user prefers hardcover books over paperback ones.*

This context may be defined as:

CREATE POSITIVE CONTEXT ctxt_cover_STX (
     cover integer,
     BINDING KEY (cover) REFERENCES books(cover)
) AS TOTAL ORDER WITH UNLISTED EXCLUDED;

SET ACTIVE CONTEXT AS ctxt_cover_STX;

For the above example, in response to Q_u, the actual query that is executed is given below. Typically, the binding key is used to join the books table with the context table. The output rows are to be sorted based on the cover *rank*, which is specified implicitly in the context as it is an ordering context.

```
SELECT T.*
FROM books T
    INNER JOIN ctx_cover_STX C1
        ON(T.cover = C1.cover
        AND C1.user_name = CURRENT_USER)
WHERE T.stock
ORDER BY C1.rank;
```

Context 4 *The user does not prefer (wants to avoid) any science fiction books.*

This context may be defined as:

CREATE NEGATIVE CONTEXT ctxt_category_GQI (
     category integer,
     BINDING KEY (category)
             REFERENCES books(category)
) AS EQUIVALENCE WITH UNLISTED INCLUDED;

SET ACTIVE CONTEXT AS ctxt_category_GQI;

In response to Q_u, the actual query that is executed is given below. Rows in books, whose category exists as any of the contextual values of this context, are eliminated from the answer set.

```
SELECT T.*
FROM books T
WHERE T.category NOT IN (
      SELECT C1.category
      FROM ctxt_category_GQI C1)
```

```
WHERE T.stock;
```

The basic contexts, which are not composed from other contexts, reflect in the actual executed query according to table 3. This table shows whether an `ORDER BY` clause is necessary, and which type of join we need according to the context properties. We use the same symbols of the context classification in Section (`[G]` for negative context, `[S]` for positive context, and so on).

Context Class	ORDER BY	Join Operation
GQN	$\times$	NOT IN
SQN	$\times$	LEFT OUTER JOIN
SQX	$\times$	INNER JOIN
STN	$\checkmark$	LEFT OUTER JOIN
STX	$\checkmark$	INNER JOIN
SPN	$\checkmark$	LEFT OUTER JOIN
SPX	$\checkmark$	INNER JOIN

Table 3: Effect of basic contexts in queries

Next, we compose complex contexts from the above basic contexts. We start with the following context.

Context 5 *The user prefers books published in 2005, and then those published in 2006 before all other books. For the books that are similarly ranked, the user prefers hardcover books over books with paperback cover.*

This context may be viewed as the composition of ctxt_year_STI and ctxt_cover_STX. Therefore, we do not need to define a new context. Conversely, we just need to set the active context appropriately to reflect to the desired context.

`SET ACTIVE CONTEXT FOR` user1
`AS` ctxt_year_STI, ctxt_cover_STX
`WITH RANKING ORDER` ctxt_year_STI, ctxt_cover_STX;

As a result of this combined context, queries to select tuples from books will work as if the query below was executed. First, the books in stock will be sorted based on the rank of the years, and then in case of ties, the cover type will be taken into consideration.

```
SELECT T.*
FROM books T
    LEFT OUTER JOIN ctx_year_STI C1
        ON(T.year = C1.year
        AND C1.user_name = CURRENT_USER)
    INNER JOIN ctx_cover_STX C2
        ON(T.cover = C2.cover
        AND C2.user_name = CURRENT_USER)
```

pid	name	age	address	phone
1	Alice Adams	10	1 April Ave.	111-1111
2	Bob Blaney	20	2 Brooks Blvd.	222-2222
3	Carl Carson	30	3 Cricket Ct.	333-3333
4	David Daniels	40	4 Dogwood Dr.	444-4444

Table 4: The patient table

recipient	purpose	pid	pid_pref	name_pref	age_pref	address_pref	phone_pref
charity	solicitation	1	√	√	√	√	√
nurse	treatment	1	√	√	√	×	√
account clerk	billing	1	√	√	×	√	√
charity	solicitation	2	×	×	×	×	×
nurse	treatment	2	√	√	√	×	√
account clerk	billing	2	√	√	×	√	√
charity	solicitation	3	√	×	×	√	√
nurse	treatment	3	√	√	√	×	√
account clerk	billing	3	√	√	×	√	√
charity	solicitation	4	√	√	×	×	×
nurse	treatment	4	√	√	√	×	√
account clerk	billing	4	√	√	×	√	√

Table 5: The patient_privacy_pref object context

```
WHERE T.stock
ORDER BY C1.rank, C2.rank;
```

5 Proof-of-Concept - Realizing Privacy-aware Databases and Spatial Databases Using Chameleon

In this section, as a proof of concept, we illustrate how one may realize specialized database systems using Chameleon. We begin with the first case study: privacy-aware databases. Then, we present Spatial databases as our second case study.

5.1 Privacy-Aware Databases

In this section, we show how we can limit disclosure, as what happens in Hippocratic Databases, using context awareness in Chameleon. In Table 4, we use the same patient table used in the limiting disclosure work aforementioned [30]. This table contains patient personal information.

Consider a healthcare facility that owns this data. Whenever a patient is admitted to the facility, he/she has to sign a privacy policy. The privacy policy

specifies which information is to be released to which recipient. Moreover, the policy also specifies for which purposes the information is to be released. On an opt-in basis, the healthcare facility also allows patients to choose if they want any of their personal information to be released to other recipients. For instance, a nurse who is treating a patient is allowed to see the patient's name, age, and phone, but is not allowed to see his/her address for any reason. The patient may opt-in and choose that only his/her age is to be released to charity for solicitation.

Beside limited disclosure, limited retention is also modeled using context awareness. For simplicity, and without loss of generality, we assume that patient data is to be retained for 90 days only. By the end of this period, the patient data should have fulfilled the purposes for which the data has been collected. After this period, different recipients cannot retrieve the data.

It is important to make it clear that the patients *in this context* are the objects. Object contexts are the contexts of the patients. Moreover, users are those that use an application at the healthcare facility to retrieve patients' data. To model the above example of limiting the disclosure and retention of patients' data in Chameleon, we define the object contexts patient_privacy_pref and patient_policy_signature as follows.

```
CREATE OBJECT CONTEXT patient_privacy_pref (
          recipient varchar(30),
          purpose varchar(30),
          pid integer,
          pid_pref boolean,
          name_pref boolean,
          age_pref boolean,
          address_pref boolean,
          phone_pref boolean,
          BINDING KEY(pid) REFERENCES patient(pid)
);
CREATE OBJECT CONTEXT policy_signature (

          pid integer,
          sign_date date,
          BINDING KEY(pid) REFERENCES patient(pid)
);
```

Let the object context patient_privacy_pref contain the contextual data in Table 5. The following user context enforces the limited disclosure and limited retention of patients' data. Table 6 gives the context of three users. If the three users execute the query "SELECT * FROM patient;", they will retrieve the data shown in Table 7.

```
CREATE POSITIVE CONTEXT identity_activity (
      job varchar(30),
      activity varchar(30),
    BINDING KEY(job, activity) REFERENCES
```

15

```
      patient_privacy_pref(recipient, purpose)
  SUBSTITUTE patient(pid)
    WITH (CASE WHEN patient_privacy_pref.pid_pref
        AND today() <= policy_signature.sign_date + 90
        THEN patient.pid ELSE NULL)
  SUBSTITUTE patient(name)
    WITH (CASE WHEN patient_privacy_pref.name_pref
        AND today() <= policy_signature.sign_date + 90
        THEN patient.name ELSE NULL)
    ⋮
) AS EQUIVALENCE WITH UNLISTED EXCLUDED;
```

user_name	job	activity
user1	charity	solicitation
user2	nurse	treatment
user3	account clerk	billing

Table 6: identity_activity contextual values

	pid	name	age	address	phone
u1	1	Alice Adams	10	1 April Ave.	111-1111
	3			3 Cricket Ct.	333-3333
	4	David Daniels			
u2	1	Alice Adams	10		111-1111
	2	Bob Blaney	20		222-2222
	3	Carl Carson	30		333-3333
	4	David Daniels	40		444-4444
u3	1	Alice Adams		1 April Ave.	111-1111
	2	Bob Blaney		2 Brooks Blvd.	222-2222
	3	Carl Carson		3 Cricket Ct.	333-3333
	4	David Daniels		4 Dogwood Dr.	444-4444

Table 7: Result of "SELECT * FROM patient;" for all users u1, u2, and u3

5.2 Spatial Databases

Spatial databases are optimized to store and query data related to objects in space. This type of databases has more complex geometrical data types, e.g., points, lines, and rectangles.

Consider a real-estate database containing information about houses. The *houses* table has the following schema: houses(id, bedrooms, price, city). An

application developer is interested in providing some spatial queries to this database, but has no privileges to add the location of the house to this table. An object context is created to add the location of houses.

Range Queries

Let the user context be the willingness to buy a house in certain regions. As a result, a user context is created in the CA-DBMS to declare that only houses contained in relevant regions are to be returned.

Below is the definition of both contexts; house_loc and houses_in_region. The function *contained* retrieves any house with location (x, y) that exist with the rectangular region (x1, y1, x2, y2).

```
CREATE OBJECT CONTEXT house_loc (
        id integer,
        x integer,
        y integer,
        PRIMARY KEY(id),
        BINDING KEY id REFERENCES houses(id)
);


CREATE POSITIVE CONTEXT houses_in_region (
        x1 integer,
        y1 integer,
        x2 integer,
        y2 integer,
        BINDING KEY() REFERENCES house_loc
                WITH contained(house_loc.x, house_loc.y
                                x1, y1, x2, y2)
) AS EQUIVALENCE WITH UNLISTED EXCLUDED;
```

Nearest Neighbor Queries

Another class of queries in spatial databases is the nearest neighbor query. In this class, the user want to retrieve the object that is nearest to a pivot location. An extension to this class of queries is the k nearest-neighbors query. The answer of this query is the k objects that are nearest to the pivot location. In the real estate database, a user willing to retrieve the houses listed by proximity to a point may declare her context as follows:

```
CREATE POSITIVE CONTEXT nearby_houses (
        x integer,
        y integer,
        BINDING KEY() REFERENCES house_loc
                WITH true
) AS TOTAL ORDER USING dist(x, y, house_loc.x, house_loc.y)
WITH UNLISTED EXCLUDED;
```

The equivalent SQL query with the awareness of this context would be:

```
SELECT T.*
FROM houses T
    INNER JOIN house_loc OC1
        ON(T.id = OC1.id),
    nearby_houses C2
ORDER BY dist(C2.x, C2.y, OC1.x, OC1.y)
```

Skyline Queries

Skyline queries emerge in spatial databases. Assume that *user2* wants to buy a house that is close to his work downtown and that is also cheap (or at least reasonable) in price. Since it is not easy to combine such preferences in a ranking expression, user2 decides to select from the skyline houses.

Such context can be defined as the composition of several contexts, namely houses_in_region, nearby_houses, and the context *price* already incorporated in the houses table. The first context will include a bounding box representing downtown. The second and third contexts will be used to compute the skyline. This composition is instantiated by setting the active context as follows:

```
SET ACTIVE CONTEXT FOR user2
AS houses_in_region, nearby_houses
WITH SKYLINE OF nearby_houses.rank MIN, houses.price MIN;
```

6 Context Composition

As pointed out in the previous sections, a complex context may be composed from basic ones. Such composition may involve compiling more than one context whose contextual relation is an ordering relation. We provide three mechanisms to resolve the conflict among the different orders of object imposed by these contexts. The first mechanism is already implemented in existing database systems; using the `ORDER BY` clause. The second mechanism entails the use of ranking algorithms. *top-k* queries have been well studied in various fields. Also, there have been numerous algorithms for embedding *top-k* queries into database operators.

We investigate the third mechanism, which outputs the skyline of the objects. We begin with the definition of a skyline. Furthermore, we discuss the type of data for which the skyline is computed in the case of the context-aware queries being generated by Chameleon. We explain why computing the skyline using existing algorithms is not efficient in our situation. The proposition of a new query operator is given next to make the computation more efficient. At the end of this section, we provide the results of some experiments performed to evaluate the new operator.

Skyline queries occur when the multiple parameters are independent and their ranks cannot be aggregated together. Skyline queries are defined as follows.
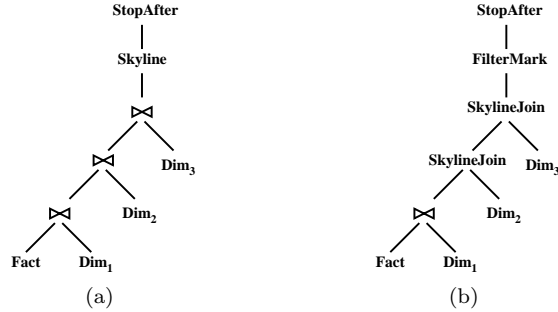
Figure 1: (a) Dimension tables are joined with the fact table before the skyline operation is performed. (b) The operators Skylinejoin and FilterMark inside a query pipeline.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in a d-dimensional space S. Let $p_{i\delta}$ denote the δth dimension of a point p_i. For each dimension δ, we assume that we have a total ordering relation on its domain values, denoted by $\succ_\delta$. Based on the user preference, $\succ_\delta$ can be either of $\{>, <\}$.

Definition 1 *A point p_j is said to dominate another point p_i on S if and only if $\forall$ dimension $\delta \in S$, $p_{j\delta} \succeq p_{i\delta}$ and $\exists \gamma \in S$ such that $p_{j\gamma} \succ p_{i\gamma}$*

Definition 2 *A point p_i is a skyline point on S if and only if there does not exist any other point p_j dominating it.*

The input of the skyline operation is a set of objects, each of which has d ranks. Therefore, each object can be viewed as a d-dimensional point. In our case, we have a main table, and all its ranks come from different relations (the user preferences from the various ordering context relations). Consequently, these relations have to be joined before a traditional skyline algorithm can be applied on them. Figure 1(a) shows a possible query execution plan of this *star-join* when we have three dimensional space. The fact table is our object table, and the dimensions of the star-join are the different ordering contexts.

The skyline operator needs all object tuples be joined with all dimensions before it outputs the first skyline object. The reason for this behavior is that for an object to be considered a skyline, it needs to be compared with all other objects to make sure none of them dominates it.

In addition to that, we want to point out the fact that with higher dimensional data, the probability that an object dominates another object becomes lower. As a result, the number of skyline points become too numerous to produce insightful results. Consequently, many users prefer to limit the result size to k objects. Yet, they will have to pay the price of joining all the tables together completely.

6.1 SkylineJoin Pipelined Query Operator

We propose SkylineJoin, a join operator that not only performs the join operator between two relations, but also marks tuples to be part of the skyline based on the dimensions joined so far. The operator is a pipelined operator. It can be integrated with the existing query engines. It adheres to the open-next-close protocol. SkylineJoin will substitute the join operators shown in Figure 1(a). On top of the pipeline, and instead of a Skyline operator, we will use a new FilterMark operator that outputs only any marked tuple that is input to it. Unmarked tuples will be discarded by FilterMark.

For the same example of a fact table (object table) and three dimension tables representing the ranking contexts, the new query pipeline containing the new operators will look like Figure 2.

In the following, and for the sake of presentation, we assume that we want to compute the skyline of d dimensions. We denote the ith dimension by dim_i. The query optimization of queries involving skyline queries is out of the scope of this document. We present the query execution phase, and thus, we take an arbitrary join order for presentation purposes.

The query pipeline is built as follows. The first dimension is joined with the fact table. We include in the required physical properties of the selected join operator that we want its output tuple to be sorted based on the ranks of the dim_1. Notice that there is a clustering index on the ordering contexts. Consequently, this property may be enforced upon the selection of the physical operators either by using that index as the outer input of the join operator and using an order-preserving join operator, or by building a sort operator on top of other join realizations. The query evaluation tree is left deep and will contain all the other dimensions as the inner input of the SkylineJoin operators. On top of the last SkylineJoin, a FilterMark operator is created to produce the skyline results.

Skyline Join works in two phases. The first phase is the join phase, where any order-preserving join may be used (e.g., nested loop join, indexed nested loop join, hash join). The second phase is the phase in which SkylineJoin decides whether or not to mark a tuple to be part of the answer set. To clearly present SkylineJoin, we omit the details of the first phase and include only the second phase. Actually, the second phase can be put as an extension to the existing order-preserving join. The second phase is modeled as a finite state machine (FSM) given in Table 8. When the FSM calls the GetNext() function, it receives the next tuple available from Phase 1.

The Skyline operator has an accompanying state. Its state contains not only a tuple store that stores intermediate results, but also a set of skyline tuples found so far (skylineSet). When a tuple is retrieved from the tuple store, it no longer belongs to the tuple store. The operator also keeps track of whether it has exhausted all the tuples from its input, which is the output of Phase 1.

The finite state machine has five states. The starting state is SKJ_INIT. In this state, the operator gets the first tuple, puts it in the tuple store, and changes its state to SKJ_GROWING. If there is no such tuple, it declares it
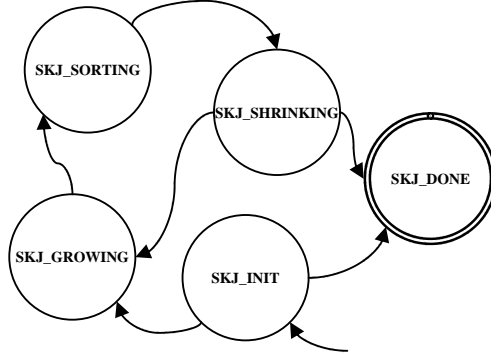
Figure 2: Finite state machine representing Phase 2 of Skylinejoin.GetNext()

ran out of input tuples and goes to a final state SKJ_DONE. This final state just returns NULL to indicate that no tuples will be output further from this operator.

The FSM transitions from a growing state, where it acquires more tuple from Phase 1, to a sorting state and then to a shrinking state, where it produces output tuples. The shrinking state may go back to the growing state, and so on. The machine goes to the final state (SKJ_DONE), when it discovers, during the shrinking state, that there is one tuple only in the tuple store and that it has exhausted its input.

When the FSM is at the growing state, SKJ_GROWING, it gets tuples from its input and puts them in the tuple store. This behavior is interrupted either when it runs out of input or if the input tuple is marked as a skyline. Conversely, the FSM behaves during the shrinking phase as follows. First, it gets a tuple from the tuple store. Second, it checks whether it needs to mark the tuple as a skyline. Last it returns the tuple. The tuple is marked as a skyline in two situations. The first situation occurs where this tuple is the first tuple retrieved from the tuple store after the sorting state. The second situation is when *isskyline* evaluates to true. The function *isskyline* compares this tuple against the set of skyline tuples found so far. It returns true if this tuple is not dominated by any existing skyline tuples.

The sorting state is visited whenever the finite state machine transits from the growing state to the shrinking state. During this visit, the tuple store is sorted according to $\succ_1$, $\succ_2$, ..., $\succ_i$; assuming that this SkylineJoin operator receives dim_i as its inner input.

A detailed example and sample step-by-step execution of the algorithm are omitted for space considerations. The reader is referred to [15] for further detail.

7 Experimental Evaluation

We evaluate the performance using a Sun Blade 2000 machine. This machine has two processors, each of them is a 1.2GHz UltraSPARC-III+ with 8MB cache. Solaris 8 operating system in 64-bit mode runs on this 2GB memory computer. The hard disk on which the data is stored runs at 10K rpm.

We implement all context-aware constructs and all the proposed extensions to SQL to support contexts inside the extensible PostgreSQL code base. In the resulting context-aware DBMS, named Chameleon, we also implement several operators to combine multiple contexts, mainy, the SkylineJoin and FilterMark operators introduced in the previous section. The fact table has the schema fact(fact_id, fact_desc, d1, d2, d3, ..., d15). We have 15 dimension tables. The schema of the dimension tables is the same. For instance, dim_1 has the schema dim1(dim_id, dim_desc, rank). The fact table hss 1,000,000 tuples.

In the set of experimental results presented in the paper, we focus on how contexts are combined. We compare the SkylineJoin algorithm to a naive skyline algorithm (referred to thereafter as Skyline) that computes the skyline tuples after joining the tables. The Skyline algorithm compares each tuple in the joined data with all other tuples output from the last join operator. There is no other alternative to compare with since there were no indexes on the output of the join. Also, we could not sort the output of the join as it requires getting all the data and then materializing it, which would not be feasible with big tables and high dimensionality. That is why we opt to use the Skyline algorithm as a base line for comparison.

The first experiment compares the performance of both SkylineJoin and Skyline algorithms. We investigate the scalability of both algorithms with the increasing number of dimension tables.
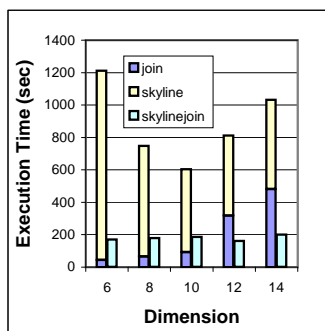

Figure 3: Scalability with respect to the number of dimensions.

Figure 3 gives the execution time of retrieving the first k skyline tuples using both algorithms. The output size is limited to 100 tuples. It is intuitive and clear from the figure that with the higher dimensionality we have, the more it takes to

22

join these tables. We can also notice that it takes more time to retrieve the first 100 skyline tuples when the number of dimensions is low. When the number of dimensions increases, the number of available skyline points increases as we point out before. The reason is that the probability for a tuple to be dominated by another tuple decreases. A significant consequence of this phenomenon is that the execution time needed to find the first k skyline tuples saturates eventually with the increase in the dimensionality. From this experiment, we realize that the proposed algorithm outperforms the available traditional skyline algorithm by at least 300%.

The following experiment is to inspect the performance and scalability of both algorithms when the output size, k, changes. Figure 4 gives the results of this experiment when 10 context dimension tables join with the fact table. We change the output size, number of reported skyline tuples, from 100, 200, ..., 500.
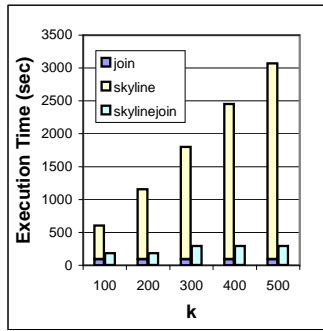

Figure 4: Scalability with respect to the output size.

Since the number of dimensions is fixed to 10, we can see that the time it takes to join the data is independent from the output size. This trend occurs as Skyline needs to join all tables completely before it outputs the first skyline tuple. The time it takes to output the first k skylines increases with both algorithms. This direct increase is due to more iterations happening in the case of Skyline to produce more skyline points. On the other hand, the increase in the case of SkylineJoin is not steep. Because of the behavior of SkylineJoin, the join operation occurs as long as we are in the growing states. Once we reach a shrinking state, we may produce several skyline tuples without the need to do any join or sort operations. The cost of these operations is amortized among several output skyline tuples. This is why we see the execution time increases in steps. In all instances of this experiment, SkylineJoin really surpasses Skyline.

8 Related Work

There have been several definitions of context and context-awareness (e.g., see [4, 14, 22, 37]). Most of these definitions define the context in terms of examples with special emphasis on the location context. Similarly, there have been several definitions of context-aware applications that include various synonyms, e.g., adaptive applications [41], reactive applications [13], responsive applications [16], situated applications [22], contented-sensitive applications [39], and environment directed applications [18]. In this paper, we adhere with the most formal definitions given in [14].

Recently, there has been interest in adding the context-awareness to relational database systems and query processors (e.g., see [27, 43]). However, the main focus is either on the modeling of the context information and how to integrate it into the query definition, or on very specific examples that consider only one type of context. None of the previous work have discussed or proposed a full-fledge realization of context-awareness inside a DBMS.

There has been several work for presenting preferences in terms of relational calculus, first order logic, and query languages (e.g., see [11, 25, 47, 29]). In terms of query processing, there are two extremes for preference-aware queries, namely, *top-k* and *skyline* queries. *Top-k* queries have been well studied in various fields (e.g., [6, 9, 17, 35]). Also, there have been numerous algorithms for embedding *top-k* queries into database operators (e.g., see [5, 10, 19, 23, 31]). On the other hand, the term *skyline queries* has been coined in the database literature [2] to refer to the secondary storage version of the maximal vector set problem [28, 33]. Due to its practicality, various versions of *skyline* queries have been studied in the literature, e.g., sorted data [12], partially-ordered domains [7], high-dimensional data (e.g., [8, 38, 46, 49, 50]), progressive and online computations (e.g., [26, 36, 44]), sliding window [32, 45], continuous skyline computations [21, 34, 48], mobile ad-hoc networks [20], spatial skylines [42], and data mining [24]. Unlike the case for *top-k* queries, there is no previous work in integrating *skyline* queries at the core of query operators or database systems.

9 Conclusion

In this paper, we proposed incorporating context awareness inside database management systems. Enabling such incorporation not only removes the need to tailor specialized systems towards a certain context, but also allows for combining several contexts together to form a complex one. We introduced Chameleon, a prototype context-aware database management system built by extending PostgreSQL. We provided SQL constructs to enable context awareness, and we gave the conceptual evaluation of this system to show how to use the system to define contexts. We gave two case studies: Hippocratic databases and Spatial databases as a proof of concept in using Chameleon to instantiate specialized systems. A key operation in the composition of contexts is the skyline operation. We designed a pipelined query operator SkylineJoin in Chameleon to

benefit from the fact that several tables are joined together prior to the computation of the skyline. This operator led to up to 300% gain in performance.

Chameleon serves as a proof-of-concept general platform for expressing and realizing contexts within a DBMS. We need to show that the performance of systems instantiated by Chameleon is comparable to that of tailored systems, e.g., Chameleon-instantiated spatial DBMS vs. a tailored spatial DBMS server, and a Chameleon-instantiated privacy-aware DBMS vs. a tailored Hippocratic DBMS. Addressing these performance issues will be the target for future research.

References

[1] Merriam-Webster Online Dictionary. http://www.m-w.com/.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 2001.

[3] P. Brown. The Stick-e document: a framework for creating context-aware applications. *Electronic Publishing*, 8(2&3), 1996.

[4] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware Applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 4(5), 1997.

[5] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *TODS*, 27(2), 2002.

[6] N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries over Web-Accessible Databases. In *ICDE*, 2002.

[7] C. Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified Computation of Skylines with Partially-Ordered Domains. In *SIGMOD*, 2005.

[8] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-Dominant Skylines in High Dimensional Space. In *SIGMOD*, 2006.

[9] K. C.-C. Chang and S. won Hwang. Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In *SIGMOD*, 2002.

[10] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The Onion Technique: Indexing for Linear Optimization Queries. In *SIGMOD*, 2000.

[11] J. Chomicki. Preference Formulas in Relational Queries. *TODS*, 28(4), 2003.

[12] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *ICDE*, 2003.

[13] J. R. Cooperstock, K. Tanikoshi, G. Beirne, T. Narine, and W. Buxton. Evolution of a Reactive Environment. In *Proceeding of the International Conference on Human Factors in Computing Systems, CHI*, 1995.

[14] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on the What, Who, Where, When, and How of Context-Awareness, CHI*, 2000.

[15] H. G. Elmongui and W. G. Aref. Skyline-Aware Join Operator. Technical Report CSD TR 08-007, Purdue University, 2008.

[16] S. Elrod, G. Hall, R. Costanza, M. Dixon, and J. des Rivières. Responsive Office Environments. *Communications of ACM*, 36(7), 1993.

[17] R. Fagin, R. Kumar, and D. Sivakumar. Comparing Top k Lists. *SIAM Journal on Discrete Mathematics*, 17(1), 2003.

[18] S. Fickas, G. Kortuem, and Z. Segall. Software Organization for Dynamic and Adaptable Wearable Systems. In *International Symposium on Wearable Computers*, 1997.

[19] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD*, 2001.

[20] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline Queries Against Mobile Lightweight Devices in MANETs. In *ICDE*, 2006.

[21] Z. Huang, H. Lu, B. C. Ooi, and A. K. Tung. Continuous Skyline Queries for Moving Objects. *TKDE*, 18(12), 2006.

[22] R. Hull, P. Neaves, and J. Bedford-Roberts. Towards Situated Computing. In *International Symposium on Wearable Computers*, 1997.

[23] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, H. G. Elmongui, R. Shah, and J. S. Vitter. Adaptive Rank-Aware Query Optimization in Relational Databases. *TODS*, 31(4), 2006.

[24] W. Jin, J. Han, and M. Ester. Mining Thick Skylines over Large Databases. In *PKDD*, 2004.

[25] W. Kießling. Foundations of Preferences in Database Systems. In *VLDB*, 2002.

[26] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, 2002.

[27] G. Koutrika and Y. E. Ioannidis. Personalized Queries under a Generalized Preference Model. In *ICDE*, 2005.

[28] H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of ACM*, 22(4), 1975.

[29] M. Lacroix and P. Lavency. Preferences: Putting More Knowledge into Queries. In *VLDB*, 1987.

[30] K. LeFevrey, R. Agrawaly, V. Ercegovac, R. Ramakrishnan, Y. Xuy, and D. DeWitt. Limiting disclosure in Hippocratic databases. In *VLDB*, 2004.

[31] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *SIGMOD*, 2005.

[32] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE*, 2005.

[33] J. Matousek. Computing Dominances in E^n. *Information Processing Letters*, 38(5), 1991.

[34] M. D. Morse, J. M. Patel, and W. I. Grosky. Efficient Continuous Skyline Computation. In *ICDE*, 2006.

[35] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting Incremental Join Queries on Ranked Inputs. In *VLDB*, 2001.

[36] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1), 2005.

[37] J. Pascoe. Adding Generic Contextual Capabilities to Wearable Computers. In *International Symposium on Wearable Computers*, 1998.

[38] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces. In *VLDB*, 2005.

[39] J. Rekimoto, Y. Ayatsuka, and K. Hayashi. Augment-able Reality: Situated Communication Through Physical and Digital Spaces. In *International Symposium on Wearable Computers*, 1998.

[40] T. Rodden, K. Chervest, N. Davies, and A. Dix. Exploiting Context in HCI design for Mobile Systems. In *HCI*, 1998.

[41] B. N. Schilit and M. M. Theimer. Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5), 1994.

[42] M. Sharifzadeh and C. Shahabi. The Spatial Skyline Queries. In *VLDB*, 2006.

[43] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Adding Context to Preferences. In *ICDE*, 2007.

[44] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, 2001.

[45] Y. Tao and D. Papadias. Maintaining Sliding Window Skylines on Data Streams. *TKDE*, 18(2), 2006.

[46] Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient Computation of Skylines in Subspaces. In *ICDE*, 2006.

[47] G. K. Werner Kießling. Preference SQL - Design, Implementation, Experiences. In *VLDB*, 2002.

[48] T. Xia and D. Zhang. Refreshing the Sky: The Compressed Skycube with Efficient Support for Frequent Updates. In *SIGMOD*, 2006.

[49] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient Computation of the Skyline Cube. In *VLDB*, 2005.

[50] Z. Zhang, X. Guo, H. Lu, A. K. H. Tung, and N. Wang. Discovering Strong Skyline Points in High Dimensional Spaces. In *CIKM*, 2005.

SkylineJoin[i].FSM.GetNext()
//The subscript i corresponds to the operator whose //inner input is dim_i, according to Figure 2.

Initialization:
    this→state = SKJ_INIT;
    this→inputExhausted = false;
    this→skylineSet = {};

```
LOOP
    state: SKJ_INIT:
            this→state = SKJ_GROWING;
            slot = GetNext();
            IF slot is NULL
                this→state = SKJ_DONE;
                this→inputExhausted = true;
                continue;
            put slot in a tuple store;
            continue;
    state: SKJ_GROWING:
            slot = GetNext();
            IF slot is NULL
                this→state = SKJ_SORTING;
                this→inputExhausted = true;
                continue;
            put slot in a tuple store;
            IF slot is marked
                this→state = SKJ_SORTING;
                continue;
            continue;
    state: SKJ_SORTING:
            this→state = SKJ_SHRINKING;
            sort tuple store according to ≻₁, ≻₂, ..., ≻ᵢ;
            continue;
    state: SKJ_SHRINKING:
            IF tuple store contains only one tuple
                IF this→inputExhausted == false
                    this→state = SKJ_GROWING;
                    continue;
                ELSE
                    this→state = SKJ_DONE;
            slot = get tuple from tuple store
            IF this is the first tuple after the sorting OR
                                isskyline(this, slot)
                slot→mark = a skyline tuple;
                include slot in this→skylineSet;
            ELSE
                slot→mark = nothing;
            return slot;       29
    state: SKJ_DONE:
            return NULL;
ENDLOOP
```

Table 8: Finite state machine representing the logic of Phase 2 of Skyline-Join.GetNext()