The Design and Implementation of DEDALE

Stéphane Grumbach   Philippe Rigaux   Michel Scholl
Luc Segoufin

November 30, 1999

1 Intoduction

This paper introduces DEDALE, one of the first implementations of a database system based on the linear constraint model. DEDALE is a long term project started in 1996 as a cooperation between the VERSO group at INRIA and the VERTIGO group at CNAM. DEDALE is intended to demonstrate the practical relevance of this model to handle geometric applications in various areas such as GIS or spatio-temporal databases. The system represents and manipulates multi-dimensional pointsets as linear constraint relations which are *nested* into classical relational tuples. Although the (global) dimension is arbitrary, the *orthographic dimension* (orthodim) (see [GRS99]) is at most 2. This restriction allows for a trade-off between efficiency of evaluation and representation power.

DEDALE has been implemented on top of the O_2 Database Management System. Although, in earlier versions, several O_2 high-level modules were used for supporting query expression and evaluation, in the current setting the system relies on the object database engine of O_2 (O_2API) which provides direct control on schema and objects, as well as low level facilities to access files, records, and indices. This means, in particular, that OQL is not used as a query language, since its evaluation mechanism cannot be adapted to the specific semantics of constraint-based evaluation.

One of the main motivations for using O_2 was the built-in facilities to define nested structures. Another useful feature of O_2 is the effective application development platform provided around the database kernel: DEDALE consists mainly of a query engine using the O_2 low level tools that provide access methods and schema management.

Figure 1 illustrates the architecture of the prototype. It consists of the following components:

1. A Graphical User Interface (GUI) which allows one to query the database with parameters such as points or rectangles, and to display query results including maps and thematic data.

2. The DEDALE optimizer composed of the following modules: (i) the lexical and syntactical analyzer of a query which requires the access to schema

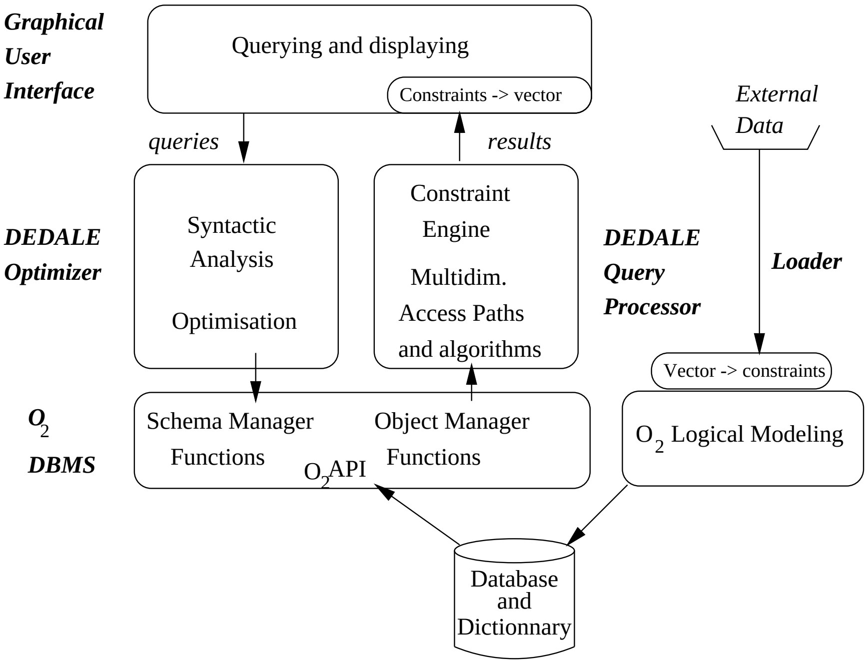


Figure 1: DEDALE Architecture.

information stored in the O_2 database, (ii) the semantic analyzer which performs the optimization phase according to rewriting rules and access path selection.

3. The query processor relies partly on functionalities provided by the O_2API, such as collection scans and B-Tree traversals, and partly on completely new modules that are specific to multi-dimensional data management, namely a constraint engine for processing queries on multi-dimensional data represented by constraints.

4. At the lowest layer, the O_2 DBMS provides data storage, access paths and schema information.

The final component, external to the query evaluation process, is the *loader* which allows importing into O_2 external data, which is usually provided in a vector-based representation. Loading data involves first converting the data into constraints, before storing it in O_2. The inverse process (from constraint to vector) is applied when displaying query results through the GUI.

The chapter is organized as follows. We first briefly describe the data model, along with its implementation in O_2. Data loading, together with the algorithms related to data conversion between representations are discussed. The user query language is then introduced and illustrated with a spatio-temporal example, after which the query evaluation mechanism is described, including some of the most important algorithms.

2 The Data Model

DEDALE handles geometric data as linear constraint relations represented in the first-order language $\mathcal{L} = \{\leq, +\} \cup \mathbb{Q}$.

The choice of restricting DEDALE to the semi-linear model, rather than the semi-algebraic one, is the result of a trade-off between complexity and expressive power. Even though, in both cases, the data complexity is in PTIME, the quantifier elimination procedure is a lot more efficient in the case of FO+LIN than in the case of FO+POLY. Furthermore, the semi-linear model seems to be sufficient for handling most of the practical applications that we are interested in.

In the following, we denote by $LCR(\mathbb{Q}^k)$ the set of linear constraint relations over $\mathbb{Q}^k$. Each formula representing a relation is converted to a quantifier-free disjunctive normal form (DNF), and is seen as a generalized relation of generalized tuples. We often blur the distinction between the columns or attributes of a constraint relation, and the corresponding variable names.

DEDALE extends the classical linear constraint relations by allowing relations that have both an uninterpreted domain D and an interpreted one $\mathbb{Q}$. The CDB concepts extend easily to such relations, where each attribute is of one of these sorts, and includes two sorts of constraints, (i) equality constraints over objects of D, and (ii) linear constraints over objects of $\mathbb{Q}$. In addition, pointset representation is restricted to be of *orthodim* 2 : the set of variables in a formula is partitioned into classes whose cardinality is at most 2, and each of these classes defines a so-called *component*. $LCR(D^{k_0}, \mathbb{Q}^{k_1}, \ldots, \mathbb{Q}^{k_n})$, with $k_i \leq 2$, $i \in \{1, \ldots, n\}$, denotes the set of linear constraint relations over $D^{k_0} \times \mathbb{Q}^{k_1} \times \cdots \times \mathbb{Q}^{k_n}$.

The data model integrates the linear constraint relations with non-first normal form relations, in order to deal more easily with the geometric extension of objects. For simplicity, nesting is limited to one level. Here is an example of a relation schema that represents objects in motion, which will be used in the examples later in this chapter. The atomic type *string* is used as an uninterpreted type.

$$
\begin{array}{lll}
PEOPLE = & \{[& \text{name} : \mathit{string}, \\
 & & \text{category} : \mathit{string}, \\
 & & \text{activity} : \{\text{name} : \mathit{string}, \text{time} : \mathbb{Q}\}, \\
 & & \text{trajectory} : \{\text{space} : \mathbb{Q}^2, \text{time} : \mathbb{Q}\} \\
 &]\} &
\end{array}
$$

In this example, the attributes *activity* and *trajectory* are nested relations in, respectively, $LCR(D, \mathbb{Q})$ and $LCR(\mathbb{Q}^2, \mathbb{Q})$. The first of these attributes denotes a 2-dimensional generalized relation, that has orthodim 1. It has two components, *name* over the uninterpreted domain of strings, and *time* over the interpreted domain $\mathbb{Q}$. The second attribute, *trajectory* denotes a 3-dimensional generalized relation of orthographic dimension 2. It has two components, *space* over $\mathbb{Q}^2$, and *time* over $\mathbb{Q}$.

Name	Category	Activity	Trajectory
John	Tourist	$(name = \text{"Sleep"})$ $\wedge\ 2 < t < 10)$ $\vee\ (name = \text{"Eat"})$ $\wedge\ 10 < t < 11)$ $\vee\ (name = \text{"Ski"})$ $\wedge\ 11 < t < 15)$ $\cdots$	$(\ x = 1230 \wedge y = 134$ $\wedge\ 1 < t < 11)$ $\vee\ (3x - 2y \leq 49$ $\wedge\ 23x + 2y \geq 134 \cdots$ $\wedge\ 11 < t < 16)$ $\cdots$

Figure 2: An object with nested attributes representing pointsets.

Figure 2 shows an instance of a tuple in relation *PEOPLE*: *name* and *category* are represented as classical atomic values, while the nested attributes *activity* and *trajectory* are represented as FO formulae. The variables t, x, and y denote, respectively, the variable associated with the *time* component, and the two variables of *space*. Observe that in *trajectory*, a constraint cannot bind t with x or y.

3 Constraint Representation and storage with O_2

The O_2 *data model* allows one to define complex types as an unrestricted nesting of the `TUPLE`, `SET`, and `LIST` constructors, and is thus appropriate to the one-level nesting in DEDALE. As an illustration, we give a schema for the relations `Resort`, representing the buildings of a ski resort, and `People`. Each geometric attribute is of type `Relation` (which denotes here a generalized relation).

```
CLASS  Resort
TUPLE [ name : STRING,
         geometry : RELATION]
```

```
CLASS People
TUPLE [ name : STRING,
        category : STRING,
        activity : RELATION,
        traj : RELATION]
```

Generalized relations are implemented using the following schema:

```
TYPE RELATION
TUPLE [ mbb: BOX,
        orthodim :
             LIST(CompType),
        Tuples : SET (Tuple)]
```

```
TYPE TUPLE
TUPLE [ mbb : BOX,
        components :
           LIST( Component),
        normal : BOOLEAN]
```

The schema is a straightforward implementation of the generalized relational model, augmented with the `orthodim` concept. The partition of variables is described by the attribute `orthodim` in type `Relation` which is a `LIST` of pairs `(name,dim)` (denoted as `CompType` in the schema). For instance the `orthodim` attribute for the relation storing a trajectory is instantiated as the list `<(space,2),(time,1)>`.

In the `TUPLE` type, `components` the attribute represents the list of the components whose type and dimension are described in the `orthodim` attribute. A `Component` is a list of constraints, where each constraint of dimension 2 is of the form "$ax + by + c\theta 0$," with θ one of $\{=, <, \leq, \geq, >\}$, and is represented by the tuple $[a, b, c, \theta]$. In the current implementation, a constraint is represented as a compact binary object. This allows for efficient clustering of constraints, and avoids the penalty of indirection in the O_2 `LIST` constructor.

The following information, mostly intended to speed up the evaluation process, has been added to the schema:

1. A first step towards optimization is achieved by associating with each tuple its *minimal bounding box* `mbb` defined as the tuple $((\min_1, \min_2, \ldots, \min_n), (\max_1, \max_2, \ldots, \max_n))$ of points in the n-dimensional space with minimal (maximal) coordinates satisfying the constraints. Using these boxes query evaluation allows for an initial fast filtering step. The minimal bounding box for the whole relation is also stored.

2. The `normal` attribute is a `BOOLEAN`, which is set to `true` if the tuple is in *normal form*, defined as follows:

   (a) There are no redundant constraints in a tuple. For instance, $x < 5$ and $x < 1$ cannot belong to the same normalized tuple.

   (b) There are no inconsistent constraints in a tuple. For instance, $x > 5$ and $x < 1$ cannot belong to the same normalized tuple. It follows that a normalized non-empty tuple is always satisfiable.

   (c) Although not central to normalization, the constraints in a normalized tuple are stored in a given order for tuple display. This is mostly important for polygons: see the following discussion on vector/constraint conversion.

3.1 Data conversion: Loading and displaying

Mapping non-convex objects to a constraint representation is necessary when loading data from a vector-based representation. The conversion is trivial for points, but more complex for polylines and polygons.

A *polyline* is an ordered collection of connected segments whose point-based representation is a list of points $\langle P_1, P_2, \ldots, P_n \rangle$. In DEDALE, a segment P_iP_{i+1} is represented as a tuple featuring a conjunction of 5 constraints: one for the equation of the line colinear to the segment, and 4 for the bounding box

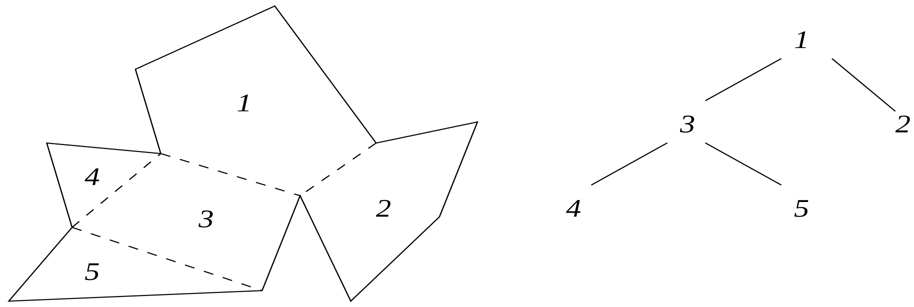


Figure 3: A simple polygon, its convex partition, and its adjacency graph.

The normal vector-based representation of a *polygon* is also the list of points $\langle P_1, P_2, \ldots, P_n, P_1 \rangle$ describing the boundary as a closed polyline. Since a generalized tuple must be a convex object, non-convex polygons must be partitioned into convex components. This is done in two steps: First the non-convex polygon is triangulated into $n-2$ triangles, second the number of convex components is reduced by merging adjacent triangles whose union is still convex.

Merging takes time linear in the number of vertices n of the initial polygon. The result is a set of convex polygons whose number is at most four times the optimal. The algorithm also returns the adjacency graph of the convex polygons (see Figure 3), and DEDALE stores this graph in order to recompute efficiently the polygon whenever needed. (However if a query modifies the polygon, it does not appear to be worthwhile to maintain the graph). A tuple is created for each resulting convex polygon: the boundary is scanned in counterclockwise order, and for each oriented edge e the constraint $a_1x + a_2y \leq b$ representing the half-plane on the left side of e is appended to the tuple. Each tuple thereby inserted in the database is normalized: There are no redundant nor inconsistent constraints. In addition, the storage order of the constraints in a $Polygon$ tuple is also the order of the segments along the boundary of the convex polygon. Thanks to this order, the vector representation can be recomputed by a simple scan.

Table 1 gives some statistics on the convex partitioning of polygons from a SEQUOIA dataset.

Type	No. of Polys	Avg No. points	% of notches	tuples per notch	No. of constr.	No. of Tuples
Agriculture	4 324	58	34.17	1.006	4.36	23
Forests	3 404	133	34.44	0.949	4.44	53
Residential	7 899	27	25.10	1.20	4.33	10

Table 1: Convex partitioning statistics

If r is the number of notches, then the optimal number of components is between $\lceil \frac{r}{2} \rceil + 1$ and $r + 1$. Table 1 shows that the number of tuples (convex components) is almost r (while the number of notches is 0.95) despite the fact that the algorithm is suboptimal. This shows that the implementation is efficient enough for real data, but the large ratio (approximately 30) between the number of convex and simple polygons illustrates the need for an efficient representation of generalized relations.

Displaying a simple (non-convex) polygon requires mapping an union of convex components into a point-based polygon representation. Two techniques are used:

- If the data comes directly from the database, for example, if it results from a *window* query, then DEDALE uses the adjacency graph stored in the database. From this graph the polygons can be easily reconstructed in time linear in the number of constraints;
- Otherwise, the data has been modified by a query, and the adjacency graph has not been maintained, for example after a *clipping* query. Maintaining the adjacency graph was rejected in DEDALE because it is complicated and expensive. DEDALE uses instead a function that, given a set of polygons, returns a set of non-intersecting, non-adjacent polygons. This can be done in time $O(n \log n)$ where n is the number of constraints involved. The idea is to compute the points from the constraints, to sort them according to their coordinates, and then use a *sweep-line* algorithm to reconstruct the polygon.

4 The Query Language

The query language follows a SQL-like syntax to construct algebraic expressions on nested attributes in the `SELECT` clause as well as boolean expressions on nested attributes in the `WHERE` clause. If p_1 and p_2 denote two pointset attributes (which can be either nested attributes or the results of algebraic expressions), then:

1. p_1 `JOIN` p_2 denotes a *natural* join on the components common to p_1 and p_2.
2. p_1 `UNION` p_2 denotes the union of p_1 and p_2.
3. p_1 `EXCEPT` p_2 denotes the difference of p_1 and p_2.
4. $p_1.c[.i]$ projects p_1 on the i^{th} coordinate of component c. When there is only one component, its name can be omitted.
5. $p_1.c.i$ `AS` j renames the i^{th} coordinate of c as j.

`JOIN` is the most general operation: depending upon the existence of common components in p_1 and p_2, one obtains a cross-product (no common components), a join (some common components), or an intersection (the same components). In the latter case the equivalent `INTER` keyword can also be used.

The renaming of components or variables is obtained with `AS` and allows us to control the semantics of the `JOIN`. Observe that spatial selection can be obtained as (p_1 `JOIN` *cst*) where *cst* is any formula that represents a pointset. We sometimes use the equivalent syntax `RESTRICT` p_1 `WITH` *cst*.

The Boolean counterpart of an operator `OP`, denoted `OP?`, is used in the `WHERE` clause. It returns **true** if the resulting set is non empty and **false** otherwise. Therefore the general form of a query is

SELECT $att_1, \ldots, att_n, E_1, \ldots, E_m$
  FROM $R_1[\ldots]$
  WHERE $B_1, \ldots, B_l$

where att_i denote atomic attributes, E_j algebraic expressions over nested attributes, and B_k Boolean algebraic expressions.

The language will not be formally described here, but rather illustrated by a sample of queries upon the *ski resort* database containing the relations `People` and `Resort` whose schema was presented above.

1. *Which building contains the point p(a,b)?*

   ```
   SELECT  name
     FROM  Resort
     WHERE geometry INTER? 'geometry.1=a AND
                                geometry.2=b'
   ```

   An important design choice of DEDALE is that it considers as *invalid tuples* those with empty nested sets. Thus the expression *geometry* `INTER?` ... is interpreted as **true** if the resulting set is non-empty, and **false** otherwise. Any Boolean query can be expressed in this way: this covers (using the standard GIS terminology) point queries, window queries, etc.

2. *Give the part of the skiing area contained in a given rectangle* rect (clipping).

   ```
   SELECT  geometry INTER rect
     FROM  Resort
     WHERE name = "Skiing Area"
   ```

   If the skiing area does not intersect the rectangle, the result is an empty set, which is automatically deleted.

3. *Give the part of those trajectories of people which are included within the skiing area.*

   ```
   SELECT  p.name, (p.traj JOIN r.geometry).space
     FROM  People p, Resort r
     WHERE r.name = "Skiing Area"
   ```

The join is performed on the component common to the attributes `p.traj` and `r.geometry`, namely `space`, and (...).`space` projects out the `time` component.

4. We now consider temporal queries and queries that merge space and time criterias.

   Where is John between 12:00 and 14:00?

```
SELECT  ( RESTRICT traj WITH '12 < time < 14').space
  FROM  People
  WHERE name = "John"
```

   This is a temporal query with spatial output: The trajectory of John. The conjunction of the constraint `'12 < time < 14'` with the trajectory of John on the attribute `time` is computed, and those tuples in the resulting generalized relation `traj` which are still satisfiable (if any) are projected on the spatial component.

5. *When does Monica stay at the bar's terrace?*

```
SELECT  (m.traj JOIN r.geometry).time
  FROM  Resort r, People m
  WHERE m.name = "Monica"
  AND   r.name = "Terrace"
```

   This is a spatial query with temporal output. The spatial join on the `space` component common to `t.traj` and `s.geometry` gives the part of Monica's trajectory inside the terrace's geometry, and the result is the projection of this onto `time`.

6. *Where is John while Monica is at the bar's terrace?*

```
SELECT  (m2.traj JOIN (m1.traj JOIN r.geometry)
                                        .time).space
  FROM  Resort r, People m1, People m2
  WHERE m1.name = "Monica"
  AND   m2.name = "John"
  AND   r.name  = "Terrace"
```

   This query is a composition of a spatial join and a temporal join. The internal join retrieves the time `t` during which Monica is at the terrace; `t` is the input argument to the temporal join with John's trajectory.

7. *Where did Monica and John meet?*

```
SELECT  (m1.traj  INTER  m2.traj).space
  FROM  People m1, People m2
  WHERE m1.name = "Monica"
  AND   m2.name = "John"
```

This join involves both `time` and `space`.

8. *When were Monica and John at the same resort?*

```
SELECT  (m1.traj JOIN r.geometry).time INTER
                      (m2.traj JOIN r.geometry).time
  FROM  Resort R r, People m1, People m2
  WHERE m1.name = "Monica"
  AND   m2.name = "John"
```

   Each object in the resort map is intersected with the trajectories of both John and Monica. This yields the places where both of them spent some time during the day. A further join on `time` restricts the result to the places where they both were at the same instant.

9. *Who ate in the skiing area, and when?*

```
SELECT  m.name, (((RESTRICT} m.activity WITH
                                'name = "Eat" ').time
                JOIN m.traj) JOIN r.geometry).time
  FROM  Resort r, People m
  WHERE r.name = "Skiing Area"
```

   The first internal temporal join between the two time-varying attributes of a person gives those places where the person eats. The external spatial join checks whether these places intersect the skiing area.

To illustrate some of the advantages of the DEDALE approach, compared to traditional GIS languages, consider the following query:

Find those buildings that are south of the skiing area

A typical way to express the query in a classical GIS query language or system would be:

```
SELECT  b.name
  FROM  Resort a, Resort b
  WHERE a.name = "Skiing Area"
  AND   b.name = "Building"
  AND   b.geometry SOUTH-OF a.geometry
```

This uses a built-in predicate, `SOUTH-OF`. In DEDALE, one could express the query as:

```
SELECT  b.name
  FROM  Resort a, Resort b
  WHERE a.name = "Skiing Area"
  AND   b.name = "Building"
  AND   (a.geometry JOIN b.geometry[1, 2 AS 3]) JOIN?
                                 'geometry.2 > geometry.3'
```

The semantics is as follows: The query outputs the buildings `b` such that, for some point `p1`, one can find a point `p2` in the skiing area `a` which is 'north of' `p1`, i.e., such that `p1.x=p2.x` and `p1.y<p2.y` hold.

The first part of the query, '`b.geometry[1, 2 AS 3]`' renames variables. Next, '`a.geo, JOIN b.geometry[...]`' is a join on the variables common to both nested attributes. One thus obtains a 3-dimensional pointset where the first coordinate is an x value common to `a` and `b`, and the second and third coordinates denote, respectively, the y component of `a` and `b`. The join with '`geometry.2>geometry.3`' completes the query (recall that `geometry.i` is the projection of the geometry component on its i-th variable).

Note that no assumption was made on the "geometric type" of spatial objects (point, line, polygon): this explains why one cannot use a simple syntax such as '`a.geometry[2]>b.geometry[2]`' which is correct only for point objects.

The renaming and join operations allow one to express directional queries (*south-of, north-west, etc*) with different choices of semantics.[1] This could be done via macro definitions, such as

```
DEFINE   SOUTH-OF (g1[2], g2[2]) AS
         (g1 JOIN g2[1, 2 AS} 3]) JOIN? ([2] < [3])
```

These examples illustrate some of the advantages of the approach taken in DEDALE. There is no limitation on the dimension, and DEDALE allows a homogeneous expression of queries for all attributes (including spatial and temporal ones). In addition, the language is abstract and general, and supports the expression of many queries which are hard to express directly in current GIS languages.

5 Query Processing

We now turn to the implementation of the query processing module of DEDALE. We outline the translation, rewriting, and evaluation techniques which are used to process a query, and then give some details on the implementation of the main constraint manipulation functions.

[1] For instance the above expression defines a *loose* south-of semantics. A strict *south-of* could also be defined, excluding buildings such that one point is north-of a point in the skiing area.

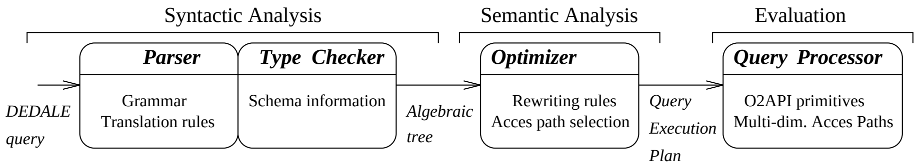


Figure 4: Evaluation of queries.

5.1 Translation, Rewriting, and Evaluation

The design of this part of DEDALE is standard and is illustrated in Figure 4. A user query is first translated into an equivalent algebraic expression by the *parser*. The *type checker* verifies the existence of relation names, attribute names, and attribute types using schema information. The third step is the *optimization* of the query according to rewriting rules and access paths selection. The optimizer delivers a Query Execution Plan (QEP) which is then evaluated. The QEP may include constraint-based functions to implement the algebraic operations on pointsets.

The optimizer is still under construction. The other modules are implemented on top of the O_2API. Query evaluation is illustrated on the following example:

> *Show the trajectory of people inside the part of the skiing area contained in the rectangle* rect.

The DEDALE query is:

```
SELECT  (p.traj JOIN r.geometry).space JOIN rect
  FROM  People p, Resort r
  WHERE r.type = "Skiing Area"
```

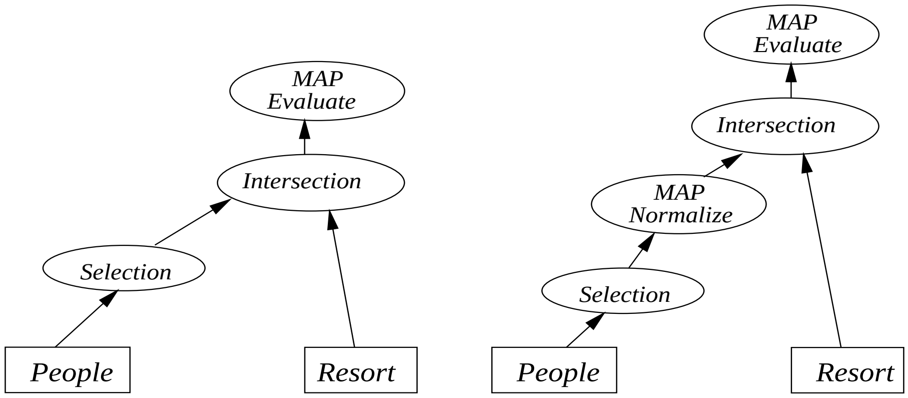
This query is first translated into the algebraic expression

$$\pi_1(\mathrm{MAP}_{\lambda X.[\sigma_{rect}(X.geometry \bowtie X.traj)]} \\ (People \times \sigma_{name=\text{“Skiing Area”}}(Resort))) \;.$$

That part of the algebraic expression inside the *MAP* construct that is over generalized relations is translated into an equivalent expression over bounding boxes of the arguments. The above expression is translated into

$$MAP_{\lambda X.[\sigma_{rect}(X.geometry \bowtie X.traj)]} \\ (\sigma_{rect}(People \bowtie_{mbb} \sigma_{type=\text{“Skiing Area”}}(Resort))) \;.$$

where $A \bowtie_{mbb} B$ is satisfied if the bounding boxes of their nested relations overlap. Such a transformation generalizes the so-called *filter step* in spatial database literature.



a. A first query execution plan *a. A second query execution plan*

Figure 5: Query execution plans.

Then the evaluation is carried out by alternating (1) a relational-like query execution plan (over the relations `People` and `Resort`), which includes an *approximate* evaluation on bounding boxes with (2) an algebraic expression over nested generalized relations with *exact* geometric evaluation.

We show in Figure 5 a typical (yet simplified) QEP. The first QEP sequentially scans the relation `People` and selects those objects whose bounding box intersects `rect`. For each such object p from `People`, the trajectory's bounding box is used to prune the index on the `Resort` collection (intersection node). This results in those objects in `Resort` whose bounding box intersects the minimum bounding box of p (the selection on the "Skiing Area" criterion is also performed there).

So far the execution only deals with relational tuples. At some point of the evaluation, one has to fetch the nested relations and perform the algebraic expression over the constraints. An important decision to take is when to perform the geometric evaluation: in the current implementation of DEDALE, it is always performed as the last operation, as in Figure 5.a, because it is likely that the number of relational tuples that one has to consider will be minimal at this point. An alternative is shown in Figure 5.b: a normalization (check for satisfiability and redundancy elimination) has been introduced after the selection on `People`. Since the bounding box is a bad approximation of a trajectory, it is likely that there are many *false hits* and that many inconsistent tuples can be removed. This is currently an issue of active research.

5.2 Constraint Manipulation

Tuple *normalization* is very important in the implementation of the DEDALE algebra. Normalization eliminates redundancy and checks whether the set of constraints is satisfiable. The normalization algorithm over 2-dimensional pointsets works as follows. The constraints are first scanned to count the number of constraints with $=$. Except for degenerate cases, this indicates the type of the result: If there are more than two such constraints, the tuple will be non-satisfiable; if the number is equal to two, the constraint will represent a point, if it is equal to one, the constraint represents a linear object, and finally, if it is equal to zero, every constraint defines a half-plane.

In this last case, a *divide-and-conquer* algorithm from computational geometry is used for normalizing a tuple. This is denoted as *half-plane intersection (HPI)*, and is done as follows: (i) the set of half-planes in the tuple is recursively split in two until one obtains singleton points – this results in a binary tree; (ii) the half-plane in each leaf singleton is intersected with a very large rectangle, resulting in a convex polygon in each leaf. The tree is then processed bottom up: at each node of the binary tree, the intersection of two convex polygons is computed.

The final convex polygon P computed by the algorithm is the intersection (if any) of all the half-planes. All that remains is to convert P towards a constraint representation. The core of the algorithm, excluding the final conversion, is given below.

Function *HPI (s : set of half-planes): a convex polygon*
begin
  if $Card(s) > 2$ **then**
    return $HPI(s/2) \cap HPI(s - s/2)$
  else
    return $s \cap$ *sufficiently large rectangle*
end

Figure 6 illustrates the process: starting from 5 constraints, one obtains the convex polygon i with four edges. The redundant constraint c has been eliminated. Since the intersection of convex polygons is linear in the number of vertices, the complexity of HPI is in $O(n \log n)$ where n is the initial number of constraints.

The *NORMALIZE* algorithm is given below:[2] *t.constraints* denotes the set of constraints in tuple t; *CEQ* denotes the subset of *t.constraints* such that *op* is '=' (straight lines), *CNE* is *t.constraints* $-$ *CEQ* (half-planes). The *MBB* function computes the bounding box of any point-based geometric object. *ToConstraints* converts a polygon into its constraints representation.

Function *NORMALIZE(tuple t): a normalized tuple* t'
begin
  $t'.constraints := \emptyset$; $t'.type :=$ *Unknown*; $t'.mbb := \emptyset$; $t'.normal := true$
  if $Card(CEQ) > 2$ **return** t' // *Unsatisfiable*
  else if $Card(CEQ) = 2$ **then** // *A point pt*
  begin

[2] To simplify the description, we assume that the input does not contain any degenerate cases such as parallel lines, or unbounded objects.

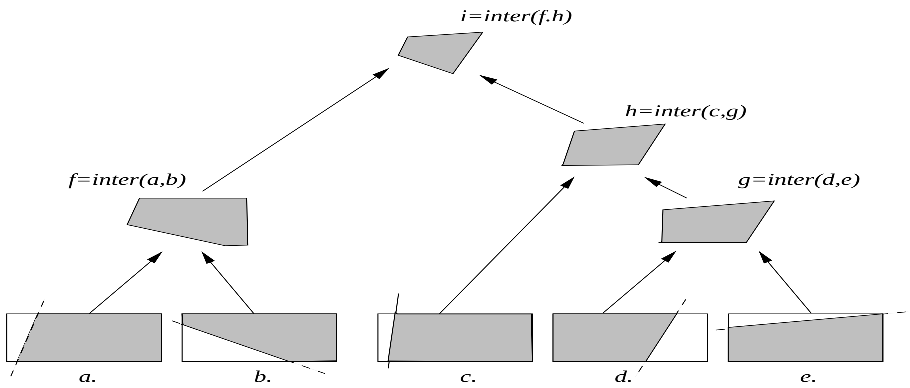


Figure 6: The Half-Plane Intersection Algorithm.

Check that each constraint in CNE contains pt
if *yes* **then**
$t'.type := Point$
$t'.constraints \leftarrow (x = pt.x \wedge y = pt.y)$
$t'.mbb := MBB(pt)$
else *return* t'
end
else if $Card(CEQ) = 1$ **then** *// A segment*
begin
t'.type := Segment; t'.mbb := MBB(HPI(CNE))
$t'.constraints \leftarrow CEQ + ToConstraints(MBB(HPI(CNE)))$
end
else if $Card(CEQ) = 0$ **then** *// A convex polygon*
begin
t'.type := Polygon; t'.mbb := MBB(HPI(CNE))
t'.constraint := ToConstraint(HPI (t.constraints))
end
return t'
end

The complexity of *NORMALIZE* is bound by that of *HPI* and is therefore $O(n \log n)$ as well.

At the relational level, two other functions are useful for constraint solving. The boolean *SATISFY* function is true if the generalized relation argument is non empty. *SATISFY* simply scans the tuples of the relation, normalizes each tuple and returns *true* as soon as it finds a satisfiable tuple. *OVERLAP*, takes as inputs two relations, R_1 and R_2, and performs a nested loop over both collections of tuples, taking the conjunction of each pair of tuples. A simple optimization is performed in order to eliminate some empty tuples: two tuples are concatenated only if the intersection of their respective bounding boxes is non empty.

Note that the tuples of the resulting relation are not normalized: some constraints in a tuple might be redundant or inconsistent.

5.3 Implementation of the Algebraic Operators

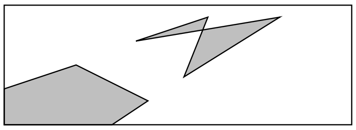
We now describe the implementation of the algebraic operators. Currently, DEDALE assumes that the data has orthographic dimension 2, which means that we need only consider physical operations on components of dimension 2.

Projection (π) The projection of a tuple in general can be computed with the Fourier-Motzkin algorithm. However, the restriction to dimension 2 allows a faster alternative. At the beginning, and during each normalization, the *mbb* of each tuple is computed. It then suffices to project the *mbb* onto the appropriate axis to get the result.

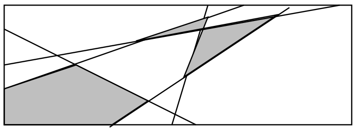
Join ($\bowtie$) and intersection ($\cap$) The join is implemented by a direct call to $OVERLAP$, and the resulting set is *not* normalized.

Union ($\cup$) Union is implemented by concatenating the tuples in its arguments.

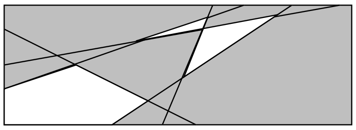
Selection (σ) $\sigma_F(R)$ is implemented as $OVERLAP(R, F)$. The result is not normalized.



a. The relation R



b. The cell decomposition obtained from R



c. The negation of R

Figure 7: Difference.

Difference ($-$) $r = r_1 - r_2$ is the set of points that are in r_1 but not in r_2. Since $r_2 = \bigvee \bigwedge c_{ij}^2$, the complement is given by is $\neg r_2 = \bigwedge \bigvee \neg c_{ij}^2$ which is a conjunctive normal form (CNF) formula. Transforming a CNF formula into one in DNF is exponential in general, but, for dimension 2 it is possible to compute the result in time $O(n^3)$ where n is the number of constraints of r_1 and r_2. The idea is to compute a *cell decomposition* of the plane using the constraints of r_1

and r_2. r_1 and r_2 are unions of such cells and therefore r is the union of the cells that are part of r_1 but not part of r_2 (see Figure 7).

6 Conclusion

DEDALE has been successfully implemented and tested with two different applications. It shows the practical relevance of the linear constraint model, and the efficiency of the constraint paradigm to handle uniformly several types of applications.

DEDALE was first tested with a GIS application. It is based on data from the French National Geographic Institute, IGN. It contains, for the Orange county in Provence, some typical data as commonly found in GISs, such as ground occupancy, roads, powerlines, rivers, etc. Figure 8 shows a screendump of this application handled by DEDALE.

DEDALE was also tested on a spatio-temporal application. The data includes an underlying map of a ski resort depicted in Figure 9, together with the trajectories and activities of a group of people, as represented in Figure 2.

It is yet too early to talk about performance because the prototype still relies on the DBMS O_2, and we would only be measuring the performance of O_2. A dedicated query processor is currently being implemented to provide a set of efficient multidimensional access paths and physical operators to manipulate the constraint database. In addition, the next version of DEDALE will be strongly oriented toward high-dimensional data (moving objects, elevation data) which integrate nicely in the constraint framework.

7 Bibliographic Notes

The DEDALE system is described in [GRSS] (implementation), [GRS98b] (data model) and [GRS98a] which illustrates the use of orthographic dimension in spatio-temporal application. The need for nested relations for spatial data has also been advocated in [BBC97].

The O_2 DBMS is described in [BDK92]. Bancilhon, Cluet, and Delobel [BCD89] describe OQL. The API as well as the spatial tools integrated in O_2 are respectively described in [Tec98a] and [Tec98b].

Many algorithms have been designed for polygon partition, mainly for triangulation of polygons [GJPT78, FM84, Cha91, Sei91]. The triangulation algorithm implemented in DEDALE is the *greedy* $O(n^2)$ algorithm described in [FPP85].

The general problem of polygon partitioning is covered in [CD85], and an algorithm for computing an optimal is proposed in [Kei85]. DEDALE uses in preference to this an algorithm that is suboptimal in terms of the number of components, but simple to implement and with linear running time [HM83]. The SEQUOIA [SFGM93] dataset was used for experiments on polygon partitioning.

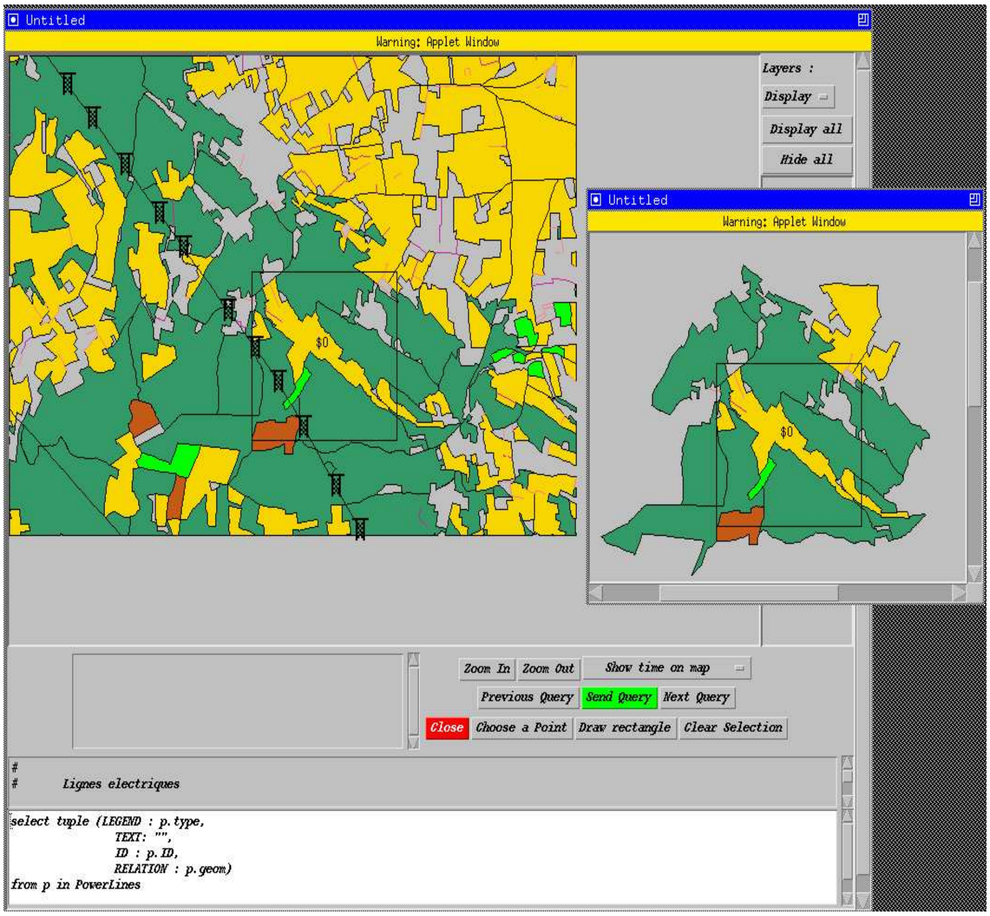


Figure 8: The Orange application in DEDALE.

Algorithms for half-plane intersection and convex polygon intersection are described in [FPP85]. [Gra93] is a comprehensive survey on query processing. Query evaluation in spatial databases is covered in-depth in [BG92]. Rewriting rules and optimization principles specific to DEDALE can be found in [GRS98b].

References

[BBC97] A. Belussi, E. Bertino, and B. Catania. Manipulating spatial data in constraint databases. In M. J. Egenhofer and J. R. Herring, editors, *Intl. Conf. on Advances in Spatial Databases (SSD'97)*, pages 115–141. Springer Verlag, LNCS 1262, 1997.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for an

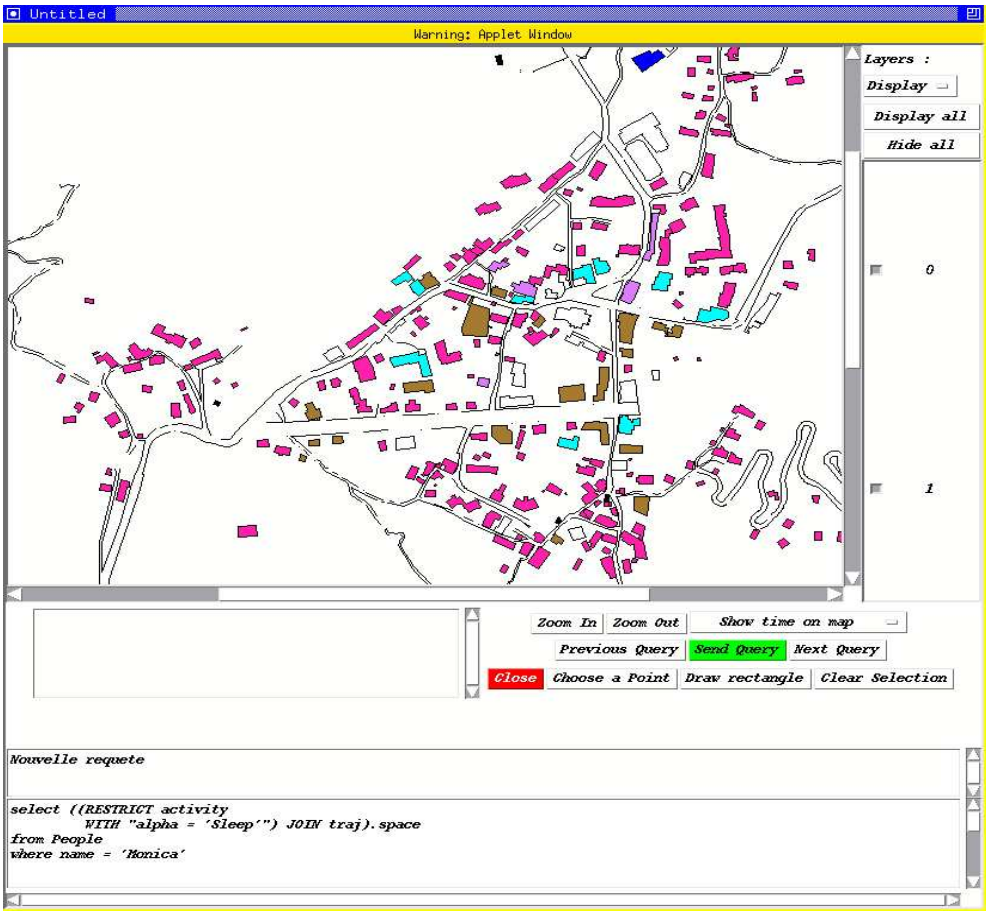


Figure 9: The ski resort application in DEDALE.

Object-Oriented Database System. In *2nd Int. Worshop on Database Programming Languages (DBPL)*, pages 301–322, 1989. Mexico.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.

[BG92] L. Becker and R.H. Güeting. Rule-Based Optimization and Query Processing in an Extensible Geometric Database System. *ACM Transactions on Database Systems*, 17(2):247–303, October 1992.

[CD85] B. Chazelle and D.P. Dobkin. Optimal Convex Decomposition. In G.T. Toussaint, editor, *Computational Geometry*, pages 63–133. North Holland, New York, 1985.

[Cha91] B. Chazelle. Triangulating a Simple Polygon in Linear Time. *Discrete and Computational Geometry*, 6:485–524, November 1991.

[FM84] A. Fournier and D.Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3:153–174, 1984.

[FPP85] M. I. Shamos F. P. Preparata. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[GJPT78] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan. Triangulating a Simple Polygon. *Information Processing Letter*, 7(4):175–180, 1978.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GRS98a] S. Grumbach, P. Rigaux, and L. Segoufin. Spatio-Temporal Data Handling with Constraints. In *ACM-GIS conference*, 1998.

[GRS98b] S. Grumbach, P. Rigaux, and L. Segoufin. The dedale System for Complex Spatial Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Estes Park, Colorado, USA, August 1997 1998.

[GRS99] S. Grumbach, P. Rigaux, and L. Segoufin. On the Orthographic Dimension of Constraint Databases. In *International Conference on Database Theory*, January 1999. Also INRIA Verso Report 141, 1998.

[GRSS] S. Grumbach, P. Rigaux, M. Scholl, and L. Segoufin. Dedale, a spatial constraint database. DBPL97.

[HM83] S. Hertel and K. Mehlorn. Fast Triangulation of Simple Polygons. In *Proc. Intl. Conf. on Foundations of Computer Theory*, LNCS No. 158, pages 207–218. Springer-Verlag, 1983.

[Kei85] J.M. Keil. Decomposing a Polygon into Simpler Components. *SIAM Journal of Computing*, 14(4):799–817, March 1985.

[Sei91] R. Seidel. A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. *Computational Geometry*, 1(1), 1991. See also the WWW site http://www.cs.unc.edu/ manocha/CODE/GEM/chapter.html.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The sequoia 2000 storage benchmark. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 2–11, 1993.

[Tec98a] O2 Technology. *O2Engine API Reference Manual*, 1998.

[Tec98b] O2 Technology. *O2Engine Spatial Toolkit*, 1998.