PSQL: An Efficient Pictorial Database System

by

N. Roussopoulos, C. Faloutsos and T. Sellis
PSQL: AN EFFICIENT PICTORIAL DATABASE SYSTEM

Nick Roussopoulos, Christos Faloutsos and Timos Sellis

Department of Computer Science
and Systems Research Center
University of Maryland
College Park, MD 20742

ABSTRACT

Pictorial databases require efficient and direct spatial search based on the analog form of spatial objects and relationships instead of search based on some cumbersome alphanumeric encodings of the pictures. This paper briefly describes PSQL, a query language which allows pictorial domains to be presented to the user in their analog form and allow him to do direct manipulation on the objects found on those domains. Direct spatial search and computation on the pictures is done using efficient data structures, R- and R+-trees (multi-dimensional B-trees), which are excellent devices for searching spatial objects and relationships found on pictures.

---

This research was sponsored partially by the National Science Foundation under the grant CDR-85-00108 and by UMIACS.
1. Introduction

Traditional data processing has only dealt with alphanumeric data types, (i.e. numerals and strings), and numeric or string comparison operators. Since Database Systems emerged from the same environment, their data types are also limited to the very same class. So management is limited to the management of structured collections of alphanumeric values.

Pictorial databases have been introduced for more than a decade now. Chang in [1] provides an excellent survey of most of the attempts in this area. The techniques used in the design, implementation and access languages of pictorial databases were influenced by the corresponding techniques in alphanumeric databases. Some of the classical database techniques had to be extended in several respects to accommodate the pictorial requirements [2,3,4,5]. This approach was not well received by the traditional image processing community which felt that they were merely providing simple table look-ups of spatial facts and vector-based displays of digitized map data [6,7]. Their view was that more advanced but less structured capabilities are needed, and that new query languages and traditional data structures are needed to support pictorial databases. Such advanced query processing capabilities include pre-computation and utilization of spatial relationships, dynamic computation of spatial relationships from the pictures, and other specialized features which can be classified as expert routines for special purpose picture manipulation tasks.

Our approach is the middle of the road one. We believe that pictorial and alphanumeric databases must be integrated to provide a uniform interface but their representation and processing must be clearly distinguished. First, pictures are not naturally representable in their alphanumeric encodings and, thus, they should be queried and presented to the user in their analog form. Furthermore, user queries and search on pictorial objects and spatial relationships among them must be direct specified in terms of the analog form using appropriate pointing devices. This allows the user to do direct manipulations on the pictorial database. Alphanumeric data associated with pictures can be displayed on the picture to assist the user.

Second, although the manipulation language must be capable of specifying pictorial and alphanumeric processing, it should not mix the syntax of the two, simply because they are fundamentally different. Instead of forcing the pictorial processing syntax specification to fit a preexisting alphanumeric language, the query language interface should allow calls to specialized
spatial operators and functions which deal with the pictorial domains of the query. This simple interface between the two allows the use of natural query syntax for both pictorial and alphanumeric processing.

Third, the processing of pictures which requires special purpose processors and tailored indexing techniques must be left outside the database processor so that these special purpose processors can be replaced, modified, or improved, according to the requirements and the sophistication of the software and hardware.

The above three premises suggest a system architecture that can support from a very sophisticated, large and high resolution pictorial database to a very simple one using inexpensive graphics hardware found in today's micro-computers. The alphanumeric data processor and the pictorial processor are different but they need to exchange control and data. Figure 1.1 shows this architecture.

From the user's point of view, the following is a list of requirements that an integrated database must satisfy:

---

![Figure 1.1: Pictorial system architecture](image-url)
1. The database must be capable of supporting pictorial domains which consist of non-atomic, non-zero space objects in addition to the simple point and alphanumeric domains. Reference and manipulation of these spatial objects must be direct by pointing to the space they occupy and not by referencing to their internal encodings.

2. The database must support direct spatial search which locates the spatial objects in a given area of the picture. This accommodates queries of the form: “Find all the bridges in a given area”, where the area is specified on the picture by a graphics direct data entry device.

3. The database must support direct spatial computation which computes specialized simple and aggregate functions from the picture. This is needed to compute areas, distances, perimeters, closest objects, etc.

4. The database must support indirect spatial search which locates objects based on some alphanumeric attributes and use the associations between the alphanumeric data and the spatial objects to place them on the picture. This accommodates queries of the form: “Display the city location and its elevation if the population exceeds 2 million”, where elevation may be directly extracted from an elevation map.

5. The database must support a more advanced user interface which allows for a direct graphics input specification (pointing devices such as a mouse, joystick, etc.) and output display coordination between the pictorial and the alphanumeric data. Value specification during the query formulation requires the display of the background of the pictorial domain.

Direct spatial search requires more advanced indexing techniques [8] because of the non-atomicity of the spatial objects. For two-dimensional spatial objects, R-trees [9,10] are excellent devices with extremely good access performance. They can be thought of as two-dimensional B-trees, and, although they are similar in nature to quadtrees [11,12], they are more flexible and their dynamic nature can better deal with “dead-space” on the pictures which slows down the search. Furthermore, because their storage organization is based on B-trees, they are better than quadtrees in dealing with disk paging [9]. However, if R-Trees are built using the dynamic insertion algorithms, the structures may contain excessive space overlap and “dead-space” in the nodes that result in bad performance. A packing technique proposed in [10]
alleviates this problem for relatively static databases. For update intensive spatial databases with highly overlapping data objects a variation to R-trees, R\(^+\)-Trees [13,14], has been suggested. R\(^+\)-trees partition the space into subspaces that do not necessarily cover the spatial object. This partitioning of the space avoids the overlap problem and the benefit offsets by far the small increase in the height of the tree.

This paper describes PSQL an experimental query language system for pictorial databases. It is built on top of ADMS [15,16], a high performance relational system that uses intelligent cache techniques and incremental access methods. The language of PSQL is an extension of SQL [17] and supports user defined abstract data types that are used for defining pictorial domains. The database maintains the associations between pictorial and alphanumeric objects. This is necessary to support direct and indirect spatial search.

Data definition in PSQL is discussed in section 2. Data manipulation through PSQL is discussed in section 3. The power of the language is demonstrated through some examples. In section 4, we describe the indexing schemes, R- and R\(^+\)-trees, that PSQL uses for indexing two or higher dimensional pictorial objects. In the same section we briefly discuss performance issues for these search access methods. Section 5 contains the conclusions.

2. Data Definition in PSQL

The relational model requires that relations are defined over a set of domains each of which has one or another form of an alphanumeric data type. PSQL extends the definition of relations over spatial and other types of domains defined in a disciplinary way to be described in the next subsection.

Every domain in PSQL is as an abstract data type. The spatial comparison operators and functions defined on each pictorial domain hide from the user the low level implementation details which deal with the encoding of the low level representation of the domain. The advantage of this approach is that the representation of a domain can change (e.g. increase the resolution of images) without affecting the relations defined over them. In the sequel, domain and data type of the domain are used interchangeably.
2.1. Defining PSQL domains

Currently, PSQL supports three basic pictorial domains: points, line segments, and regions. Segments and regions are considered as objects whose internal representation and discretization is not explicitly modeled by the relational primitives of PSQL, but using special purpose external data structures appropriate for each domain. In addition to these three basic pictorial domains, PSQL also supports the standard alphanumerical domains, integers, reals, and strings.

A new (non-standard) PSQL domain is defined by defining each one of the following four:

a. the **form of the representation** (e.g. for a region, the representation could be a sequence of line segments making up its perimeter, or a bitmap)

b. the **operators for comparing** elements of the domain (e.g. two regions overlap if their bitmaps have common elements)

c. the **functions for computing** attributes of an element or a set of elements of the domain (e.g. the area of a region)

d. the **domain background** required to enter and/or display the elements of the domain (e.g. the US map if the domain describes states).

2.1.1. The form of the representation

The form of the elements can be arbitrary but it is not part of the relational model. Each element of a domain has its own identity, semantics, and internal structure. For the relational model, it is just a surrogate treated as an atomic value. For example, the form of the region domain has an internal storage representation which is outside the relational model. Furthermore, this internal representation may be different for different domains and/or databases. For example, in a relation

\[
\text{states}(\text{state}, \text{population-density}, \text{capital}, \text{state-region})
\]

storing information about states, the region of the state may be more appropriately represented by a bit-map in one application but a sequence of line segments in another. Each tuple in the relation \text{states} models a relationship between a geographic region on the US map and some alphanumerical data (state name, population-density and capital) associated with it, but considering each region as an *inseparable unit* even though in its internal storage structure it may consist
of many other smaller sub-components.

In storing the internal representation of points, segments and regions, we found it very convenient to use the internal ADMS storage structures for storing relations. However, the tuple records stored are ordered and grouped in classes each of which corresponds to a spatial object. For example, a line segment is stored by a sequence of X,Y coordinate pairs that correspond to its points. Similarly, one may choose to store regions by the X,Y coordinates of all the points of its perimeter. These internal storage structures are pseudo-relations because their tuples are ordered and grouped, in contrast to regular relations which consist of independent and unordered tuples. Pseudo-relations are not accessible by the user except through the special purpose domain dependent operators.

2.1.2. Domain comparison operators

The operators for comparison are necessary for cross-referencing. The operands are pairs of elements of the same or compatible domains or pairs of which one is an element of the domain and the other is a “compatible” constant or “threshold”. They return true or false. An example for the region domain is the overlap operator which evaluates to true if the first region operand overlaps with the second. A compatible threshold for the region domain may be a border line, such as a time zone separator.

The number of the comparison operators is very dependent on each individual domain, and the application. We made no attempt to come up with a universal set of operators. Instead, we made PSQL flexible enough to allow the user to define additional specialized domain operators.

Using this PSQL abstract data type definition capability, we implemented the following pictorial operators:

• point operators
  \(N, S, E, W, NE, NW, SE, SW\) (north, south, etc.)
  nearest
  furthest (negation of nearest)

• region operators
  cover
  not-cover
  covered-by
  not-covered-by
  overlap
  not-overlap
• segment operators
  \textit{intersect}
  \textit{not-intersect}
• mixed operators
  \textit{point/segment/region/window within segment/region/window}
  \textit{point/segment/region/window not-within segment/region/window}
  \textit{point/segment/region/window cross segment/region/window}
  \textit{point/segment/region/window not-cross segment/region/window}

2.1.3. Domain functions

Functions defined on a domain may be simple or aggregate. A simple function receives elements from a set of domains and returns a value of another domain (in the range of the function). They are very dependent on the domains and their form of representation because they operate directly on that internal representation. For example, the region domain has simple functions for computing the area of a region, measuring the length of its perimeter, etc., and these functions use techniques and algorithms found in image processing systems. Aggregate functions receive a set of domain elements and return a value of another domain. The returned (range) value, for functions, both simple and aggregate, may be from the same domain with the operand(s) or from a different domain.

The functions we have implemented in PSQL are:

• point functions
  \textit{nearest}
  \textit{furthest}
• segment functions
  \textit{length}
  \textit{slope}
• region functions
  \textit{area}
  \textit{perimeter}
• mixed operators
  \textit{distance (point/segment/region,point/segment/region)}
  \textit{cross-point (segment,segment)}

2.1.4. The domain background

The domain background defines the medium for entering and/or displaying constant elements of a domain. Since domains have been extended to more general types, the value specification of an element through the keyboard is not adequate and requires additional \textit{direct}
input devices, such as a mouse, a cursor, etc. For example, the region domain in the states relation needs a "pointing device" for specifying an element's value on the US map during the query formulation. But no "pointing device" can be used without displaying the background, medium which assists the user to enter a value. In the same states example, the background is the US map perimeter. For alphanumerical data types, the background is implicitly assumed to be the set of all strings, integers, reals, etc. The background in this case is easily understood and the user needs no assistance for specifying domain values. However, in the case of pictorial domains, such as state-region in states, without the background, in this case the perimeter of the US map, a user may initiate a search for a state in the middle of the Atlantic ocean.

The background is typically static and fixed during the definition of the domain. It could be an exemplar from the actual values with a sole purpose to provide the range of the domain values. The background is also used during the relation display to place the retrieved values on (after deleting any exemplar data from it).

2.2. Constant constructor functions

A constant constructor is a function that allows the construction of constants of a PSQL domain. Constant specification is necessary during query formulation. Contrary to the constants of alphanumerical domains, the specification of constants from complex PSQL domains can be very elaborate requiring special input devices which operate on the internal representation. For example, a region constant constructor window receives as argument

\{X-coordinate+X-displacement, Y-coordinate+Y-displacement\}

and produces a region contained within the above XY coordinates. The window constructor is defined over the set of points and the set of reals. Its range is the PSQL domain "region".

Currently PSQL has the following constant constructors:

point (X-coordinate, Y-coordinate)
segment (point, point)
circle (point, radius)
window (X-coordinate+X-displacement, Y-coordinate+Y-displacement)
2.3. Intra-Domain operators

Each domain has a set of intra-domain operators for creating, editing, modifying, deleting, selecting a particular element, etc., which directly manipulate the form and its internal representation. Intra-domain operators are outside the relational model and their interface is not part of PSQL although the latter allows a natural and smooth integration of intra-domain operators. This set of operators can be extended or modified without affecting PSQL.

2.4. Defining a PSQL relation

Relations are defined over alphanumerical and/or pictorial domains. They model inter-domain relationships. Every tuple models a relationship among those alphanumerical and pictorial elements of the domains.

The following examples of relations are defined over a set of pictorial domains (point, segment, region) all of which are elements of the US map. We assume a universal coordinate system for this example that allows “superimposition” of different scale maps.

- `cities(city, state, population, location)`
- `states(state, population-density, capital, state-region)`
- `time-zones(zone, hour-diff, zone-area)`
- `lakes(lake, area, volume, lake-region)`
- `highways(hwy-name, hwy-section, segment)`

The pictorial domain of location in the `states` relation is of type “point”. The domains of `state-region`, `zone-area` and `lake-region` are of type “region”, and that of `segment` in the `highways` relation is of type “segment”. All other columns are alphanumerical.

For the complete specification of a relation in PSQL, a display specification is necessary. A display specification consists of

a) the location
b) the granularity, and
c) the visual association.

Since domains are of an arbitrary form, each of which may need a different display, the location of every element of a relation tuple must be specified. The granularity of the display is required for specifying how many tuples must be displayed at any time. If a domain contains elements which are maps, the granularity must be set to one tuple at a time because each map would
overwrite the previous one. If on the other hand, the domain is the set of cities on a map, several cities can and should be displayed on it. Finally, the visual association specifies the alphanumeric attributes to be displayed along with the pictorial domains on each display location. This assists the user to visualize the correspondence between those two domains.

PSQL implements the associations between alphanumeric and pictorial domains using forward and backward (unique) identifiers of type pointer [18,5] which point to the location on the picture or the relation records. These identifiers are computed when the relations are generated or updated. The identifier value (pointer−value) is used to select the relation tuples during a forward direct search, i.e., when it retrieves tuples using the picture. Note that a pictorial relation could be associated with more than one picture. In this case, one identifier is required for each picture association of this relation. This increases the complexity of the updates, but provides higher data sharability.

2.5. Defining secondary indices

B−trees are used to speed up the search on alphanumeric columns. For the columns defined over pictorial domains PSQL uses R−trees, [9], packed R−trees [10], and efficient R+-trees [14,13]. Section 4 of this paper deals with these access methods and the performance obtained in spatial search.

3. The Data Manipulation Language

PSQL supports an interactive and a programmatic interface for accessing the database. The interactive query processor preprocesses PSQL statements into ADMS SQL statements with appropriate internal procedure calls to the pictorial operators and functions. These internal routines are implemented using the ADMS programmatic interface.

3.1. The retrieve mapping

PSQL is based on the SQL mapping extended appropriately:

    select <attribute-target-list>
    from <relation-list>
    [on <picture-list>]
    where <qualification>
The following example is a typical simple query in PSQL

Q1.  
\[
\text{select city, state, population, location} \\
\text{from cities} \\
\text{on us-map} \\
\text{where location within window(4±4,11±9)} \\
\text{and population > 450,000} 
\]

which selects all cities in the area \(4±4,11±9\) (Eastern US, entered by by a mouse or, as is illustrated above, using the window function) having population greater than 450,000. Figure 3.1 shows the alphanumeric result and the pictorial output of query Q1 displayed on two windows. Note that the object names are used as visual association and are displayed on the picture to assist the user to visualize their correspondence.

The \<attribute-target-list\> is the set of tuples resulting from the query. The \<relation-list\> defines the source relations that will be queried. The optional \textbf{on} \<picture-list\> in the mapping is a name list that specifies the picture the query is on. This is used when tuples of the same relation correspond to different pictures. Full name matching for identity on the \<picture-list\> is done, exactly as it is done for the \<relation-list\>. The \textbf{on}-clause can be omitted provided that the domains of the relations are associated with a unique picture. The search area on the picture is specified in the \<qualification\> clause. It can be either a bound variable or a location given in absolute constant coordinates or in variable coordinates. The location variable may just be the name of a location predefined outside the retrieve mapping.

<table>
<thead>
<tr>
<th>city</th>
<th>state</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Orleans, Louisiana</td>
<td>357,515</td>
<td></td>
</tr>
<tr>
<td>St Louis, Missouri</td>
<td>453,085</td>
<td></td>
</tr>
<tr>
<td>Memphis, Tennessee</td>
<td>646,354</td>
<td></td>
</tr>
<tr>
<td>Nashville, Tennessee</td>
<td>495,451</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Figure 3.1} Alphanumeric and pictorial result of a query
Furthermore a search area in the <qualification> may be followed by a pictorial operator “cover”, “overlap”, etc., followed by another area specification. The meaning of \( loc1 \ op \ loc2 \), is that the qualified tuples must have locations \( loc1 \) and \( loc2 \) that satisfy the operator \( op \).

3.2. Superimposition

A very powerful operation in PSQL is the superimposition or synthesis of information stored in multiple but yet referring to the same geographical area pictures. The following example illustrates this powerful feature by synthesizing information found on two pictures, i.e., information about cities associated with us-map and time-zones associated with a time-zone-map to obtain cities together with their time-zone.

Q2. select city, zone
    from cities, time-zones
    where location within zone-area

Figure 3.2 shows the two maps and Figure 3.3 the result of superimposition. The alphanumeric data consists of the complete relations cities and time-zones displayed next to each other if the geographic area of one spatial object (city in this case) is covered by the geographic area of the other (time-zone). Superimposition is performed by simultaneous search on the two (or more) spatial organizations which correspond to the same area. The entries can be superimposed if their associated locations satisfy the condition. The simultaneous use of several spatial organizations is analogous to the use of two or more secondary indexes during the query processing where

---

Figure 3.2: The cities and time-zones maps

---
the intersection of the indices speeds up the search.

Superimposition is a very powerful operator. Its semantics is similar to the relational join operator. For the join to be meaningful, the tuples of the two relations must refer to the same entity [10,17]. For the superimposition, it is sufficient that the two operand pictures refer to the same geographic area which in this case plays the role of the entity ("geographic join").

3.3. Nested mappings

PSQL mappings can have several nested levels by mapping from a deeper level to the next level. The query below illustrates the binding of two nested mappings. The state region is passed from the interior level and used to direct the search in the exterior one to produce those lakes in the Eastern states which are completely covered by states:

Q3. 
```
select lake,area,lake-region
from lakes
where lake-region covered-by
    select state-region
    from states
    where state-region covered-by window(4±4,11±9)
```

The binding of the top level window is dynamically done during the evaluation of the query.
3.4. Pictorial functions

Pictorial domain functions can be incorporated in either the <target-list>, or the <qualification> clause. The invocation of such a function returns the computed value that is then used for displaying or to further process the query. The following are examples of function usage.

Q4. Compute the area of all states

```sql
select state, area(state-region)
from states
```

Q5. Find all states whose area is above 30,000 square miles

```sql
select state
from states
where area(state-region) > 30000
```

Q6. Find the total length of each highway

```sql
select hwy-name, sum(length(segment))
from highways
group by hwy-name
```

3.5. More queries

In the rest of this section we give some additional queries to demonstrate the power of PSQL:

Q7. Find the highway section of I95 that is closest to Washington, D.C.

```sql
select hwy-section
from highways,cities
where city = 'Washington,D.C.'
and hwy-name = 'I95'
and distance(location,segment) = min ( 
  select distance(location,segment)
  from highways,cities
  where city = 'Washington,D.C.'
  and hwy-name = 'I95')
```

Q8. Find the cities that are on highway I95.

```sql
select city, location
from cities
where location within 
  select segment
  from highways
  where hwy-name = 'I95'
```

Q9. Display all states in Central time zone.
select state,state-region
from states
where state-region covered-by
    select zone-area
    from time-zones
    where zone = 'Central'

Q10. Display all states along with their population density within 1500 miles of Washington, D.C.

select state,state-region,population-density
from states,cities
where state-region overlap circle(location,1500)
and city-name = 'Washington,D.C.'
circle is assumed to be a constructor that generates a circle using the first argument as the center and the second one as the radius (see previous section).

Q11. Display all states that straddle the boundary between Mountain and the Pacific time zones.

select state,state-region
from states
where state-region overlap
    select zone-area
    from time-zones
    where zone = 'Mountain'
or zone = 'Pacific'
minus
    select state,state-region
    from states
    where state-region covered-by
        select state-region
        from time-zones
        where zone = 'Mountain'
or zone = 'Pacific'

Q12. Display all states that straddle any time zone boundary.

select state,state-region
from states,time-zones
where state-region not-covered-by zone-area
and state-region overlap zone-area

Q13. Display all highways that cross from one time zone into another.

select hwy-section,segment
from highways
where hwy-name in
    select hwy-name
    from highways,time-zones
    where segment cross zone-area

Note that the above query will display the whole highway, section by section, if any of its sections crosses a border of a time zone.
3.6. PSQL preprocessor

PSQL queries are preprocessed and translated into ordinary SQL queries. The only additional requirement from SQL is the capability of executing system defined procedures from within the where-clause. This feature is used to call the pictorial operators and functions during the execution of the query. The output of PSQL queries is directed to two output windows specified by the relation's display specification. The prototype we are currently developing uses an alphanumeric and a graphical output window on a SUN workstation. The graphical window displays the area of the picture containing the qualifying pictorial objects and the standard terminal displays the alphanumeric data. This is very useful for indirect spatial search because it allows the user to simultaneously visualize the correspondence between data about spatial objects and their picture.

4. The Foundation for the Direct Search of PSQL

Non-pictorial objects, i.e., strings and numerical fields, can be indexed with any traditional method. We choose B-trees [21], because they achieve logarithmic search time, automatic reorganization, good space utilization and they preserve the ordering. For pictorial data either R- or R+-trees can be used.

4.1. R-trees

R-trees [9] were proposed as a natural extension of B-trees in higher than one dimensions. They combine the nice features of both the B-trees and quadtrees. Like B-trees, they remain balanced, while they maintain the flexibility of dynamically adjustable windows that deal with "dead-space" on the pictures, like the quadtrees do. A second important feature of R-trees is the fact that, at the leaf level, they store full and non-atomic spatial objects. This feature provides a natural and high level object oriented search. Furthermore, because the storage organization of R-trees is similar to that of B-trees, they are efficient in dealing with paging [9].

R-trees have been proposed as an advanced indexing technique for direct spatial search that can deal with non-atomic spatial objects. They can, however, be used as a representation medium like quadtrees [12]. Moreover, they can be used as a storage structure, too, like B-trees do.
The decomposition used in R-trees is dynamic, driven by the spatial data objects. Therefore, if a region of an n-dimensional space includes dead-space, no entry in the R-tree is introduced. Leaf nodes of the R-tree contain entries of the form

\[(RECT, oid)\]

where \(oid\) is an object-identifier and is used as a a pointer to a data object and \(RECT\) is an n-dimensional minimal rectangle (called Minimal Bounding Rectangle or MBR) which bounds the corresponding object. For example, in a 2-dimensional space, an entry \(RECT\) will be of the form

\[(x_{low}, x_{high}, y_{low}, y_{high})\]

which represents the coordinates of the lower-left and upper-right corner of the rectangle. The possibly non-atomic spatial objects stored at the leaf level are considered atomic, as far as the search is concerned, and, in the same R-tree, they are not further decomposed into their pictorial primitives, i.e. into quadrants, line segments, or pixels.

Non-leaf R-tree nodes contain entries of the form

\[(RECT, p)\]

where \(p\) is a pointer to a successor node in the next level of the R-tree, and \(RECT\) is a minimal rectangle which bounds all the entries in the descendent node. The term branching factor (also called fan-out) can be used to specify the maximum number of entries that a node can have; each node of an R-tree with branching factor four, for example, points to a maximum of four descendents (among non-leaf nodes) or four objects (among the leaves). To illustrate the way an R-tree is defined on some space, Figure 4.1 shows a collection of rectangles and Figure 4.2 the corresponding tree built for a branching factor of 4.

In considering the performance of R-tree searching, the concepts of coverage and overlap [10] are important. Coverage is defined as the total area of all the MBR’s of all leaf R-tree nodes, and overlap is defined as the total area contained within two or more leaf MBR’s. Obviously, efficient R-tree searching demands that both overlap and coverage be minimized, although overlap seems to be the more critical of the two issues. For a search window falling in the area of \(N\) overlapping leaves, in the worst case, \(N\) paths from the root to each of the overlapping leaves have to be followed slowing down the search from \(h\) to \(hN\), where \(h\) is the height of the tree. Clearly, since it is very hard to control the overlap during the dynamic splits of R-trees,
efficient search degrades and it may even degenerate the search from logarithmic to linear. Minimal coverage reduces the amount of dead space covered by the leaves.

It has been shown, that zero overlap and coverage is only achievable for data points that are known in advance and, that using a packing technique for $R$–trees, search is dramatically improved [10]. In the same paper it is shown that zero overlap is not attainable for region data objects. To attack this problem we have derived a variation to $R$–trees, $R^+$–trees.

4.2. $R^+$–Trees

$R^+$–trees [13] avoid overlap at the expense of space which indirectly increases the height of the tree. The main difference here is that rectangles can be decomposed into smaller sub-rectangles in order to avoid overlap among minimal bounding rectangles. That is, in some cases,
in which a given rectangle covering a spatial object at the leaf level overlaps with another rectangle, we decompose it into a collection of non-overlapping sub-rectangles whose union makes up the original rectangle. All the pointers of the sub-rectangles point to the same object. These sub-rectangles can be judiciously chosen so that no bounding rectangle at any level need be enlarged. The same sub-splitting is propagated up to the non-leaf nodes, thus overlap is forced to stay zero. Figures 4.3 and 4.4 show how the boxes of Figure 4.1 will be organized in an $R^+$-tree, with a branching factor of 4.

$R^+$-trees can be thought of as an extension of K-D-B-trees [22] to cover non-zero area objects (i.e. not only points but rectangles as well). An improvement over the K-D-B-trees is the reduced coverage; the nodes of a given level do not necessarily cover the whole initial space. Moreover, compared to R-trees, $R^+$-trees exhibit very good searching performance, especially for

---

**Figure 4.3:** Some rectangles, organized in an $R^+$ tree

---

**Figure 4.4:** The corresponding $R^+$-tree

---

- 20 -
point queries, at the expense of some extra space (see following sub-section and [14]).

After this brief discussion to motivate the introduction of $R^+$-trees we move now to formally describe the structure. A **leaf node** is of the form

$$(\text{RECT}, \text{oid})$$

where $\text{oid}$ is an object identifier and is used to refer to an object in the database. $\text{RECT}$ is used as in $R$-trees to describe the bounds of data objects. An **intermediate node** is of the form

$$(\text{RECT}, p)$$

where $p$ is a pointer to a lower level node of the tree and $\text{RECT}$ is a representation of the rectangle that encloses.

The $R^+$-tree has the following properties:

(1) For each entry $(p, \text{RECT})$ in an intermediate node, the subtree rooted at the node pointed to by $p$ contains a rectangle $R$ if and only if $R$ is covered by $\text{RECT}$. The only exception is when $R$ is a rectangle at a leaf node; in that case $R$ must just overlap with $\text{RECT}$.

(2) For any two entries $(p_1, \text{RECT}_1)$ and $(p_2, \text{RECT}_2)$ of an intermediate node, the overlap between $\text{RECT}_1$ and $\text{RECT}_2$ is zero.

(3) The root has at least two children unless it is a leaf.

(4) All leaves are at the same level.

Let us assume that $M$ is the maximum number of entries that can fit in a leaf or intermediate node. One property satisfied by an $R$-tree but not an $R^+$-tree is that in the former every leaf node contains between $M/2$ and $M$ entries and each intermediate node contains between $M/2$ and $M$ nodes unless it is the root [9]. K-D-B-trees do not satisfy this property either. However, Robinson showed with his experimental results that storage utilization in K-D-B-trees remains in acceptable levels (60%, which is only 10% below the average B-tree utilization). Although, $R$-trees achieve better space utilization at the expense of search performance we believe that 10% degradation is a minimal price to pay for the the search improvement obtained in $R^+$-trees (see the following sub-section).

Another interesting comment here is due to the fact that populating the nodes as much as possible will result to a decrease in the height of the tree at the expense of more costly updates.
Therefore another parameter of the problem should be the initial packing algorithm used to populate an \( R^+ \)-tree and its reorganization techniques. A packing algorithm along with the insertion, deletion and split procedures are presented in detail in \[13\].

4.3. Performance Analysis

An approach that simplifies the analysis \[14\] is to transform the objects into points in a space of higher dimensionality \[23\]. For a rectangle aligned with the axes, four coordinates are enough to uniquely determine it (the \( x \) and \( y \) coordinates of the lower–left and upper–right corners). Since 4–d spaces are impossible to illustrate, we will examine here segments on a line (1–d space) instead of rectangles in the plane (2–d space), and we transform the segments into points in a 2–d space. Each segment is uniquely determined by \( (x_{\text{start}}, x_{\text{end}}) \), the coordinates of its start and end points. Obtaining formulas and results for line segments is a first step to the analysis of 2–d rectangles, or even objects of higher dimensionality. However, there are applications for line segments, also: Orenstein \[24\] suggests the \( z \)–ordering to map a multi–dimensional space to a 1–d space. Each rectangle is thus mapped to a set of line segments; the point– and region– queries in the multi–dimensional space directly correspond to point– and region– queries in the 1–d space.

For a given point, let \( \text{Density} \ D \) be the number of segments that contain it. For our analysis, we have assumed two sets of segments, set 1 with \( N_1 \) segments of size \( \sigma_1 \) and set 2 with \( N_2 \) segments of size \( \sigma_2 \). The segments of each set are uniformly distributed on the entire space. Due to this uniformity assumption, \( D \) is the same for every point in the space. Allowing more than one size for segments enables the analysis to account for realistic distributions where not all objects are of the same size. In \[14\] we have shown that the same analytical results still stand in the case of more than two sets of segments.

In the following we give some indicative results of the search performance of both \( R^- \) and \( R^+ \)–trees. We consider two types of queries:

\begin{itemize}
  \item \textbf{Point Queries}: Given a certain point in the space, find all objects that contain it
  \item \textbf{Region Queries}: Given a region (user window), find all objects that intersect it.
\end{itemize}

First, we show the number of disk accesses required to search an \( R^- \)–tree or \( R^+ \)–tree in case of a point query. Figure 4.5a–b shows the disk accesses required for searching an \( R^- \)–tree and a
corresponding $R^+$-tree used to index 100,000 segments with total density of 40. The first figure (4.5a) shows disk accesses required as a function of the large segment density when the large segments account for 10% of the total number of segments (i.e. $N_1=90,000$ and $N_2=10,000$). Figure 4.5b illustrates the number of disk accesses as a function of the number of small segments for a fixed small segment density ($D_1=5$). These figures show clearly the problem that $R$-trees have in handling many small segments but just a few lengthy ones. In figure 4.5a large density implies long segments. In such situations, an $R$-tree may require more than twice the page accesses required by an $R^+$-tree. Notice also that the performance of the $R^+$-tree is "immune" to changes in the distribution of the segment sizes.

In the second set of figures, Figures 4.6a–b, we illustrate the number of disk accesses needed when performing a segment query on an $R$- or $R^+$-tree. The query segment was chosen to be on the order of 2 small segments. This decision was made based on the fact that segment queries are mostly performed to isolate a few segments in a given space ("zooming"). Again, the graphs show that $R$-trees suffer in cases where few lengthy segments are present. Performance improvements (i.e. savings in disk page accesses) of up to 50% can be achieved. Of course, when

![Diagram](image)

**Figure 4.5:** Disk Accesses for Two-Size Segments: Point Query
(a) As a function of $D_2$; $N_2=10,000$  (b) As a function of $N_1$; $D_1=5$

- 23 -
the number of large segments approaches the total number of segments, $R^+$-trees will lose since
many lengthy segments cause a lot of splits to sub-segments. However, typical distributions do
not have this characteristic. On the contrary, lengthy segments are few compared with small
ones (e.g., in a VLSI design).

This concludes our presentation of some analytical results we have obtained. For a more
detailed description, the reader is referred to [6]. We are currently working on the experimental
verification of these results.

5. Conclusions

We believe that pictorial and alphanumeric databases must be integrated to provide a uni-
form interface but their representation and processing must be clearly distinguished. In this
paper we described PSQL, a system that allows pictures to be represented, stored and queried in
their analog form. PSQL allows the user to do direct manipulations on the pictorial database.
Alphanumeric data associated with pictures can be displayed on the picture to assist the user.
The query language interface allows calls to specialized spatial operators and functions which
deal with the pictorial domains of the query. Hence, a simple interface between the conventional
and pictorial fields allows the use of natural query syntax for both pictorial and alphanumeric processing.

The processing of pictures requires special purpose processors and tailored indexing techniques in order to achieve high performance. In section 4 we also discussed schemes for efficiently storing and indexing pictorial data. Both R–trees and R⁺–trees provide object oriented search which is essential in pictorial systems. Our analytical results agree with our intuition, that is, R–trees suffer in the case of few, large data objects, which force a lot of "forking" during the search. R⁺–trees handle these cases easily, because they split these large data objects into smaller ones. We are currently experimenting through simulation in order to verify the analytical results. In parallel, we study alternative methods for partitioning a node and compacting R⁺–trees.

6. References


