Université d'Ottawa · University of Ottawa

Efficient Associative Data Structures for Bitemporal Databases

By

Ashraf Eid

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Computer Science
In
Computer Science
Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
Faculty of Engineering

April 2002

Abstract

Most applications require storing multiple versions of data and involve a lot of temporal semantics in their schema. This requires maintenance and querying of temporal relations. A Bitemporal DBMS will simplify a lot the development and maintenance of such applications by moving temporal support from the application into the DBMS engine. The success of such Bitemporal DBMSs relies mainly on the availability of high performance indices that handle update and search operations efficiently.

A successful associative data structure (index) is the one that can efficiently partition the space of the attributes that are used within the keys. Temporal attributes have unique characteristics and should support now-relative intervals. These intervals grow as time grows and thus we need an index that can handles attributes with variable values. The proposed bitemporal index partitions the bitemporal space into four subspaces according to the end value of the temporal intervals. This results in separating those keys that have variable intervals from those that have fixed interval(s). In this thesis we have used on-the-shelf index that successfully indexes spatial attributes. But instead of representing the two temporal dimensions as a rectangle, we have represented them as 4 dimensional points. This results in better partitioning of each subtree space and in better search performance.

Acknowledgements

I would like to thank Professor Sivarama Dandamudi for his supervision, review and guidance for the work in this thesis.

I would like to thank Professor Leopoldo Bertossi and Professor Ahmed Karmouch for reviewing the thesis and providing valuable feedback during the thesis defense.

I would like also to thank Dr. Stan Matwin and Dr. Luigi Logrippo for their supervision and time.

Special thanks to my colleague Hicham Ouahid for reviewing my thesis and providing valuable feedback.

I would like to express my gratitude to my colleagues at Cisco Systems for literally taking much of my workload onto their shoulder.

I would like to dedicate this thesis to my family who suffered a lot to provide me with the support and the atmosphere to be able to complete this work. I am most indebted to my wife, Sally, for providing the devotion and encouragement without which this thesis could not have been completed. Special love to my children, Sara and Sayfullah, had to sacrifice part of their childhood.

Table Of Contents

Table Of Figures

Table Of Tables

Chapter 1 Introduction

Conventional database systems capture only a single snapshot of the modeled reality. While serving many applications well, they are not sufficient for applications that require the support of time varying information (past and/or future data).

Currently, most applications of database technology are temporal in nature. Examples include financial applications such as portfolio management, accounting, and banking; record-keeping applications such as personnel, medical-record, and inventory management; scheduling applications such as airline, train, and hotel reservations and project management; and scientific applications such as weather monitoring. Applications such as these rely on temporal databases, which record time-referenced data.

Electronic commerce is another emerging field that relies on the capability of the database to store the history of all transactions.

Temporal DBMSs add built-in support for storing and querying multiple versions of data to conventional DBMSs that only provide built-in support for one (the current) version of data. Multiple versions of data are useful in many application areas as described above.

The temporal support for handling multiple versions of data is today typically implemented in an ad-hoc fashion in the application code. This support is implemented anew for each application being developed. Implementing temporal support in the application is time consuming and difficult using SQL-92 and conventional relational DBMSs. On the contrary, a temporal DBMS makes it very easy to handle multiple versions of history, current, and future data, because the query languages for such DBMSs have been enriched with high-level constructs for exactly these purposes. The code for handling multiple versions of data is thus moved from the applications to the

DBMS. This move reduces the complexity of application development. In this way, the productivity of application programmers is improved when using a temporal DBMS. With the clear benefits of temporal DBMSs, these systems should already be commercially available. However, this is not the case, in part because the temporal database research community has mostly focused on the design of temporal data models and query languages. Relatively little attention has been paid to implementation issues. The research community has not provided evidence of the benefits of a temporal DBMS substantial enough for the major DBMS vendors to extend their conventional DBMSs to become temporal. Much less research has addressed the design of efficient temporal-query processing techniques. This thesis addresses the need for efficient indexing of temporal data.

1.1 Time Dimensions

It is widely recognized that time is not just like any other type of information, but it needs a special treatment, both from the theoretical point of view (that is, defining new models for databases) and from the practical point of view (that is, extending DBMSs in a special way). In particular, at least two orthogonal dimensions of time have been recognized [SA86]:

- *Valid time.* This corresponds to the time when a given fact is true. The user usually supplies valid times. This fact is stored in the database at some point in time, and it is current until logically deleted.
- *Transaction time.* The transaction time of a fact, which is a pair of dates, specifies when that information has to be considered logically stored in the database. If the delete time is not specified then the tuple must be considered *logically* stored in the

database. No information is ever physically deleted from the database: a deletion of a tuple at a time corresponds to setting the deletion time of the tuple, so that the tuple is only logically deleted from the database. Thus the user can restore the state of the database at a given time *t* in the past *(rollback operation)*; in this case, the system must consider only those tuples whose transaction time includes ***t***, i.e. all the data that have been inserted into the database before ***t*** and such that the delete time is not specified or is after ***t***. Transaction times may be implemented using transaction commit times, and are system-generated and –supplied.

There are also other times that literatures refer as "User-defined". Unlike *valid-time* and *transaction-time*, it does not have built-in support in the temporal DBMSs.

From this point forward, we will use VT to denote Valid time and TT to denote transaction time.

1.2 Temporal Data

Apart from some primary keys and keys that rarely change, many attributes evolve and take new values over time. For example, in an employee relation, employees' titles may change as they take on new responsibilities, as will their salaries as a result of promotion or increment. Table 1 and Table 2 show changes in the title and salary for employee Emp1.

EmpID	Title	EffectiveStart	EffectiveEnd
Empl	S/W Engineer	Jan. 1, 1997	Jun. 1, 1998
Empl	S/W Development Mgr.	Jun. 2, 1998	Dec. 31, 1999
Empl	S/W Development Dir.	Jan. 1, 2000	Now
....			

Table 1 - EmpTitle Relation

EmpID	Salary	EffectiveStart	EffectiveEnd
Empl	50,000	Jan. 1, 1997	Dec. 31, 1997
Empl	55,000	Jan. 1, 1998	Jul. 31, 1998
Empl	60,000	Aug. 1, 1998	Dec. 31, 1999
Empl	65,000	Jan. 1, 2000	Aug. 31, 2000
Empl	70,000	Sep. 1, 2000	Now
....			

Table 2 - EmpSalary Relation

Traditionally, when data is updated, its old copy is discarded (physically deleted) and the most recent version is captured. Conventional databases that have been designed to capture only the most recent data are known as *snapshot databases*. With increasing awareness of the values of history of data, maintenance of old versions of records becomes an important feature of database systems. Due to its complexity, this functionality is only partially supported by snapshot database technology, resulting in the use of ad-hoc and expensive solutions for each application.

In an enterprise, the history of data is useful not only for control purposes, but also for mining new knowledge to expand its business or move on to a new frontier. Historical data is increasingly becoming an integral part of corporate databases despite its maintenance cost. In such databases, versions of records are kept and the database grows as the time progresses. Data is retrieved based on the time for which it is valid or/and recorded. Databases that support the storage and manipulation of time varying data are known as *temporal databases* [C+98].

Temporal databases are classified according to the time dimensions that they can support:

- *Rollback* database

  A database that supports transaction time may be visualized as a sequence of relations indexed by time and is referred to as a *rollback* database [B+97]. The reader should distinguish between the rollback database and the transaction rollback in traditional snapshot databases. The transaction rollback is just a mechanism to move the database state back to the start of the transaction. Once rolled back, the transaction info is lost. While the rollback database allows users to view the database in any previous state over and over again. The transaction rollback is however available in the rollback databases as well.

- *Historical* database

  A database that supports valid time, records history of the enterprise being modeled as it is currently known. Unlike rollback databases, the historical databases allow retroactive changes to be made to the database as errors are identified.

- *Bitemporal database*

A database that supports both time dimensions is known as *bitemporal* database. While a rollback database views records as being valid at some time as of that time, and a historical database always views records as being valid at some moment as of now, a bitemporal database makes it possible to view records as being valid at some moment relative to some other moment.

1.3 Indexing Temporal Data

Conventional indexes have long been used to reduce the need to scan an entire relation to access a subset of its tuples, to support the selection algebraic operator, and temporal joins. Indexes are even more important in temporal relations due to their size. Many temporal indexing strategies are available [ST99]. Many of the indexes are based on B^+-trees, which index on values of a single key. While the remainder indices are mostly based on R-trees, which index on ranges (intervals) of multiple keys.

1.4 Indexing Bitemporal Data

The majority of the proposed temporal indices are for transaction-time data, and only few support valid-time data. Significantly less research has been done on creating indices that support data with both valid and transaction time, so-called bitemporal data.

Some of the existing bitemporal indices follow the *partial persistence* [DRI89] approach and do not treat valid and transaction time symmetrically, but obtain bitemporal support by making a valid-time access partially persistent. Such indices do not efficiently support range queries on transaction time. Another approach is to base bitemporal indices on spatial indices, due to the similarities between bitemporal and spatial data: the combined

valid and transaction time of a fact can be treated as a region in two dimensional space. Several existing proposals [KTF95] and [KTF98] are based on the R*-tree [BKSS90], which is known as the most efficient member of the R-tree family of spatial indices.

1.5 Indexing Now-Relative Bitemporal Data

An aspect of time that has been intriguing philosophers for centuries and that is difficult to describe fully is the concept of the current time, which we term *Now* [CLI97]. This concept is unique to time; indeed, there really does not exist any other notion quite like it. Among its properties, the current time is ever-increasing, all activity is trapped at the current time, and the current time separates the past from the future. The spatial equivalent, *here*, simply fails to enjoy the properties of *now*. As Merrick Furst puts it, "The biggest difference between time and space is that you can't reuse time." The uniqueness of *now* is one of the reasons why techniques from other research areas are not readily, or not at all, applicable to temporal data; *now* offers new data management challenges, which are particular to temporal databases.

Most of existing bitemporal indices fall short in efficiently supporting *now-relative* data. Only two of the existing indices support both now-relative transaction and valid-time data: GR-Tree[BJSS98] and the 4-R Tree [BJSS00] described in section 2.5.1.2.

1.6 Queries

In general, access methods are closely related to the queries they are designed for. Various types of queries for temporal databases have been discussed in the literature [GS93], [S94], [SKKM94], and [SOL94]. Like any other applications, temporal indexing structures must be able to support a common set of simple and frequently used queries

efficiently. The prevailing queries may be purely temporal, purely attribute based, or a combination. For temporal queries, a further distinction of point versus interval queries is possible.

1.7 Contributions Of Thesis

Existing research shows that regular indices such as B+-trees are unsuited for temporal data [ST99]. Recently, a number of indices for temporal data have been proposed, the majority of them are for transaction-time data, and only a few support valid-time data. Significantly less research has been done on creating indices for bitemporal data. Also, most of these bitemporal indices fall short in efficiently supporting now-relative data [CLI97], data for which the end of the valid time and/or transaction time tracks the progressing current time.

Two of the existing indices, the 2-R index and the Bitemporal R-Tree [KTF98], efficiently support now-relative transaction-time, but not now-relative valid-time. And only two of the existing indices support both now-relative transaction and valid-time data: GR-Tree[BJSS98] and the 4-R Tree [BJSS00]. The GR-Tree is a new index that is not supported by any existing DBMS and adding it to a DBMS such as DB2, Oracle or Sybase would require an extension of the DBMS's Kernel. The 4-R Tree, on the other hand, uses the R*-tree which is supported by most DBMS's Kernels but it suffers from a degradation in performance due to the un-proportionally long minimum bounding rectangles in 2 of the 4 R* Trees.

In this thesis, we will extend the 4-R Tree further in order to enhance its performance by eliminating the un-proportionally long minimum bounding rectangles.

Extensive experiments have proved that the proposed index has the best query performance for different combinations of workload and queries.

1.8 Overview Of Thesis

The rest of the thesis is organized as follows. Chapter 2 "Related Work", starts by describing the two fundamental and orthogonal temporal aspects of data, namely valid time and transaction time followed by surveying the existing work related to the indexing of temporal data. Chapter 3 "Point-based 4-R Index for Now-Relative Bitemporal Data", describes both the original 4R-Tree and our proposed access method to index Now-relative bitemporal data. Chapter 4 **"Experimental Design"**, present the performance studies and result details, while Chapter 5 "Conclusions and Future Work" concludes and points to research directions.

Chapter 2 Related Work

Temporal database management is a vibrant field of research, with an active community of several hundred researchers who have produced some 2000 papers over the last two decades. Most of these papers are listed in a series of seven cumulative bibliographies. The seventh Temporal Database Bibliography Update [WJW98] collects 331 new temporal database papers published between 1995 and 1998 and provides pointers to its predecessors. The papers are classified into 12 categories, namely: (1) Models; (2) Database Design; (3) Query Languages; (4) Constraints; (5) Time Granularities; (6) Implementations; (7) Access Methods; (8) Real-Time Databases; (9) Sequence Databases; (10) Data Mining; (11) Concurrency; and (12) Other Papers.

The field has produced a comprehensive glossary of terminology [C+98], an edited volume which captures state of the art circa 1993 [T+94], and three workshop proceedings [CT95], [EJS98], and [S93]. The near complete SQL3 standard includes a Part 7, SQL/Temporal [M96]. The topic of temporal databases is now covered in textbooks (for example, [AHV95], [C99], [EN94], [LL99], [Z+97]) and an encyclopedia [TW99]. Most recently, the first book dedicated entirely to temporal databases has appeared [S00].

2.1 Temporal Database

Many real-world applications need to deal with historical data. A number of temporal models were proposed to handle those historical data [SA86] [BFG97] [BWJ95] [CC97] [GOS97] [JS96] [T97] [S97]. While these models are different in many ways, they have common representation of their time dimension: it is normally represented by using the

concepts of discrete time points and time intervals. A time interval, denoted by $[T^s, T^e]$, is defined to be a set of consecutive equidistant time instants (points), where T^s is the beginning time instant and T^e is the ending time instant of the interval (i.e. $T^s < T^e$).

2.2 Bitemporal Databases

Bitemporal databases support both valid and transaction times. They record not only the history of tuples in temporal tables, but also record the history of the databases themselves. This allows corrections/predictions to tuples while also maintaining the history of the database itself. It also allows asking queries based on different states of the database and/or tuples.

As a simple example, consider a video store where customers, identified by `CustomerIDs`, rent video tapes, identified by `TapeNums`. The video store keeps a record of these rentals in a relation termed `CheckedOut`. We consider a few rentals during June 2001:

- On the 2nd, customer C101 rents tape T1234 for three days. The tape is subsequently returned on the 5th.

Rec #	CustomerID	TapeNum	Transaction Time		Valid Time	
			TT^s	TT^e	VT^s	VT^e
1	C101	T1234	2	UC[1]	2	4

[1] UC denotes "Until Changed" and means that the information is valid until the current time. More details can be found in section 3.1.

- On the 5th, customer C102 rents tape T1245 with an open-ended return date.

Rec #	CustomerID	TapeNum	Transaction Time		Valid Time	
			TT^s	TT^e	VT^s	VT^e
1	C101	T1234	2	UC	2	4
2	**C102**	**T1245**	**5**	**UC**	**5**	**Now**

- The tape is eventually returned on the 8th.

Rec #	CustomerID	TapeNum	Transaction Time		Valid Time	
			TT^s	TT^e	VT^s	VT^e
1	C101	T1234	2	UC	2	4
2	C102	T1245	5	**7**	5	Now
3	**C102**	**T1245**	**8**	**UC**	**5**	**7**

- On the 9th, customer C102 rents tape T1234 to be returned on the 12th.

Rec #	CustomerID	TapeNum	Transaction Time		Valid Time	
			TT^s	TT^e	VT^s	VT^e
1	C101	T1234	2	UC	2	4
2	C102	T1245	5	7	5	Now
3	C102	T1245	8	UC	5	7
4	**C102**	**T1234**	**9**	**UC**	**9**	**11**

- On the 10^{th}, the rental period is extended to include the 13^{th},

Rec #	CustomerID	TapeNum	Transaction Time		Valid Time	
			TT^s	TT^e	VT^s	VT^e
1	C101	T1234	2	UC	2	4
2	C102	T1245	5	7	5	Now
3	C102	T1245	8	UC	5	7
4	C102	T1234	9	**9**	9	11
5	**C102**	**T1234**	**10**	**UC**	**9**	**13**

- but this tape is not returned until the 16^{th}.

Rec #	CustomerID	TapeNum	Transaction Time		Valid Time	
			TT^s	TT^e	VT^s	VT^e
1	C101	T1234	2	UC	2	4
2	C102	T1245	5	7	5	Now
3	C102	T1245	8	UC	5	7
4	C102	T1234	9	9	9	11
5	C102	T1234	10	**15**	9	13
6	**C102**	**T1234**	**16**	**UC**	**9**	**15**

2.3 Temporal Queries

Temporal DBMSs provide built-in support to temporal queries. James Allen [A84] has introduced the following set of temporal primitives:

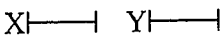
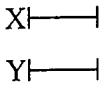
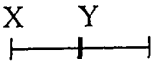
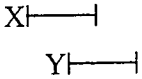
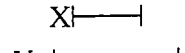
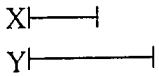
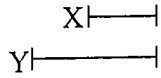
Relation	Visual Example
X *before* Y Y *after* X	
X *equal* Y	
X *meets* Y Y *met by* X	
X *overlaps* Y	
X *during* Y	
X *starts* Y	
X *finishes* Y	

Table 3 - Temporal primitives adapted from [A84]

Later, Richard Snodgrass started working with the ANSI and ISO SQL3 committees in late 1994 to introduce SQL/Temporal [SKKM94], which is not yet fully approved. Richard Snodgrass named his temporal query language TSQL2 that introduces capabilities beyond those available in SQL. Follows are some of these capabilities.

- Better support for time periods,
- Support for multiple granularities,
- Support for multiple representations,
- Support for multiple languages,
- Support for multiple calendars,
- Support for temporal indeterminacy, and

- Support for historical time.

The query language also supports Allen's temporal primitive (Table 3) through keywords like BEFORE, AFTER, WHILE, FIRST, LAST, etc.

2.4 Bitemporal Queries

Bitemporal queries allow answering queries posed in the past, present and future for both the valid and transaction time. A query that finds all versions valid during a given time interval [VT^l, VT^u] as of a given transaction time TT is called "**Bitemporal time-slice query**". This query comes in four types depending on the search operation.

2.4.1 Intersection Queries

Given a time interval [VT^l, VT^u] and a time point TT, retrieve all the versions whose valid time intervals intersect the given interval as of transaction time TT. For example, in the previous 'Video Rental' example, find all tapes that were out of the store during June 3 and June 7 as of June 5, 2001 would return records # 1 and 2 .

2.4.2 Inclusion queries

Given a time interval [VT^l, VT^u] and a time point TT, retrieve all the versions whose valid time intervals are included in the given interval as of transaction time TT. For example, in the previous 'Video Rental' example, find all tapes that were out of the store between June 2 and June 13 as of June 15, 2001 would return record # 1, 3, and 5.

2.4.3 Containment queries

Given a time interval [VT^l, VT^u] and a time point TT, retrieve all the versions whose valid time intervals contain the given interval as of transaction time TT. For example, in the previous 'Video Rental' example, find all tapes that were out of the store from June 5 to June 6 as of June 15, 2001 would return record # 3.

2.4.4 Point queries

Given two specific time points VT and TT, retrieve all the versions whose valid time intervals contain VT as of transaction time TT. Point queries can be viewed as a special case of intersection queries or containment queries where the time interval [VT^l, VT^u] is reduced to a single time instant VT. For example, in the previous 'Video Rental' example, find all tapes that were out of the store on June 5 as of June 15, 2001 would return record # 3.

Another type of bitemporal query is the "**Bitemporal key-range time-slice query**". This type of queries find all versions which are in the given key range [K^l, K^u] that are valid during the given time interval [VT^l, VT^u] as of a given transaction time TT. Like the time-slice query, the time slice part of the query can assume one of the predicates described in subsections 2.4.1 to 2.4.4. For example, in the previous 'Video Rental' example, find all tapes that were rented by customer C102 during June 2 and June 12 as of 'Now' would return record # 3 and 6.

Generic Notation

In this section, we present a notation scheme (proposed in [TSJ98]) used to classify bitemporal queries. For our purposes, a tuple is characterized by three entries:

1. a nontemporal key
2. a valid-time interval, and
3. a transaction-time interval.

In general, a tuple may have many nontemporal attributes, but for simplicity, we assume only one.

A query is classified using the notation: Key/Valid/Transaction. This notation specifies which entries are involved in the query and in what way. Each entry can be described as a "point", "range", "*", or "-". A "point" for the key entry means that the user has specified a single value to match the key attribute, while point for the Valid or Transaction entry implies a single time instant is specified for this entry. Similarly, "range" of values for the key entry, or an interval for the Valid/Transaction entries. A "*" means that any value is accepted for this entry, while "-" means that this entry is not applicable.

Bitemporal time-slice queries described in 2.4.1 to 2.4.3 can be represented using the notation "*/range/point". While bitemporal time-slice query described in 2.4.4 can be represented using the notation "*/point/point". On the other hand, the bitemporl key-range time-slice queries are represented using the notation "range/range/point".

2.5 Access Methods

One of the challenges for temporal databases is to support efficient query retrieval based on time and key. To support temporal queries efficiently, a temporal index that indexes and manipulates data based on temporal relationships is required. Like most indexing structures, the desirable properties of a temporal index include efficient usage of disk space and speedy evaluation of queries.

A fact captured in the database, may be associated with many valid time intervals and can be overlapped, and all intervals may be closed. In the example presented in section 2.2, records 2 and 3 represent the same fact "customer C102 is renting Tape T1245". On the other hand, transaction time intervals of this fact do not overlap, and its last interval is usually not closed. This is clearly shown in the same example where record 2 and record 3 have non-overlapping transaction time intervals with record 3 having an open interval. Both properties present unique problems to the design of temporal indexes.

A wealth of indices for temporal data exists; references [B+97] [ST99] provide comprehensive surveys. Here, we focus on the indexing of bitemporal data.

2.5.1 Access Methods for Bitemporal Data

There are two approaches used to index bitemporal data:

2.5.1.1 Partial-persistent methodology

This approach does not treat valid and transaction times symmetrically, but obtain index by making a valid-time index partially persistent [DRI89]. Such indices do not efficiently support range queries on transaction time.

The M-IVTT (Multiple Incremental Valid Time Trees) [NDE96], is a two level hierarchical index based on B-trees. Figure 1 shows the structure of the M-IVTT. In the upper level, one tree indexes the transaction time of events, having its leaves pointing to valid time trees. Underneath this transaction time tree there is a forest of Valid Time Trees (VTTs.) Each VTT indexes the valid time of all records existing at that point in (transaction) time. Due to potentially large demand for space, only some VTTs are kept

full, along with sufficient information (patches) on how to reconstruct any of the other ones.

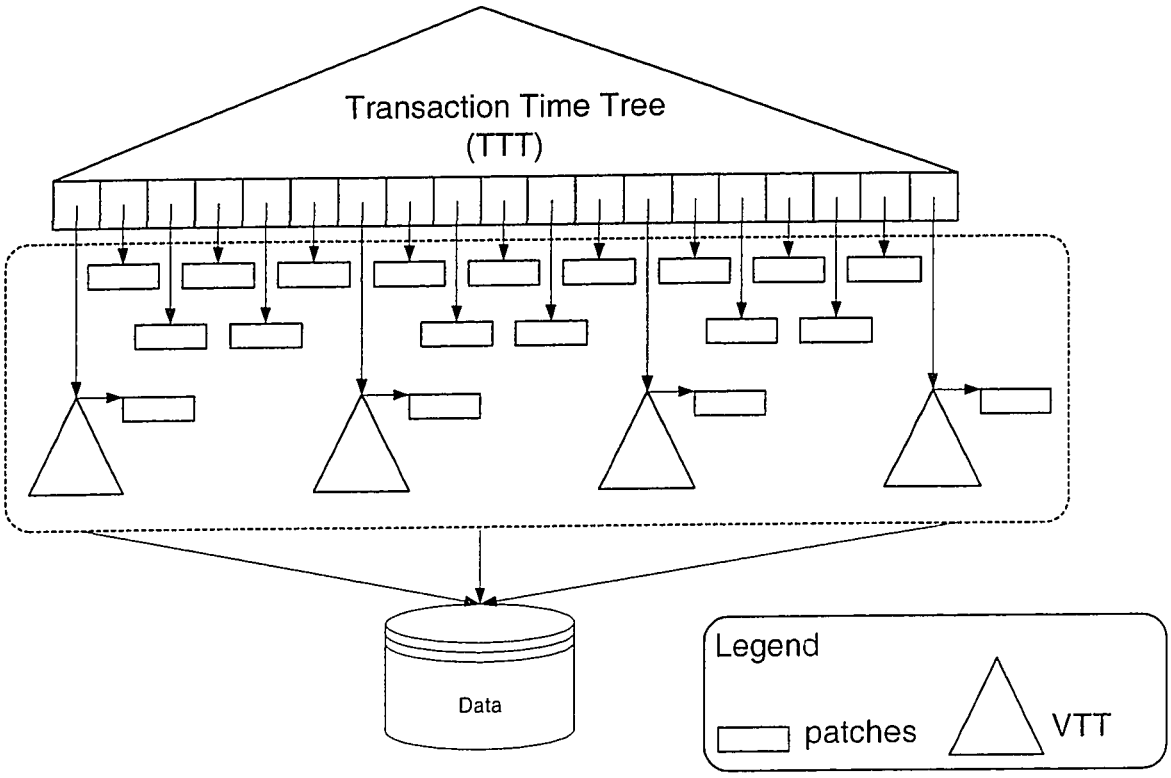
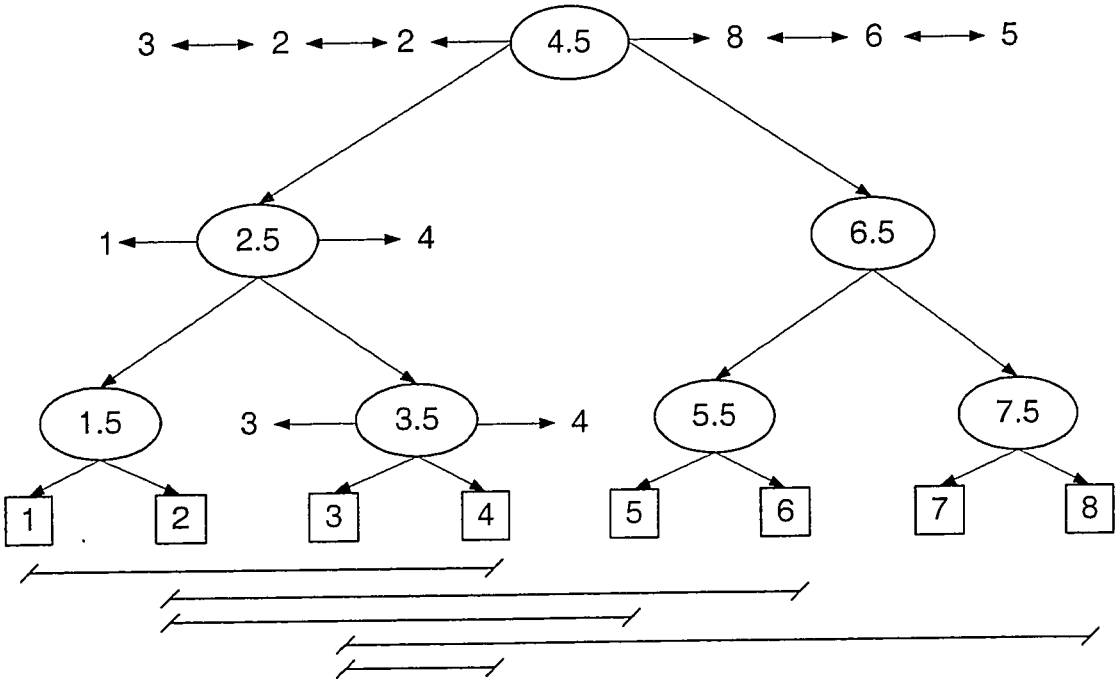


Figure 1 – The M-IVTT indexing approach (adapted from [NDE96])

The Bitemporal Interval Tree (BIT) [KTF95] is a modification of the Interval Tree [E83], which is a transient memory based structure with good worst-case performance. The BIT is disk based, partially persistent and well paginated. As summarized in [KTF98], BIT is more efficient for bitemporal queries whose predicate valid time VT (or valid interval $[VT^s, VT^e]$) satisfies VT ε $\mathcal{U}$ (respectively $[VT^s, VT^e] \subseteq \mathcal{U}$). For simplicity, they assumed $\mathcal{U} = \{1, 2, \ldots, V\}$ and S to be a set of n intervals with endpoints from $\mathcal{U}$. An Interval Tree for set S, with respect to $\mathcal{U}$, consists of a (backbone) full binary tree T with V leaves and a number of lists as shown in Figure 2. Each leaf is labeled with one element from $\mathcal{U}$. Each

nonleaf node u is assigned a value $val(u)$ that serves in directing the search from node u to its subtrees. Every interval $[l, r)$ from S is associated with a single nonleaf node u of T, where u is the node that contains l and r in its left and right subtrees, respectively. Intervals associated with some node u are kept in two doubly linked lists: $L(u)$ and $R(u)$. $L(u)$ (respectively $R(u)$) keeps the intervals in increasing (decreasing) order of their left (right) endpoint. Inserting an interval $[l, r)$ in the Interval Tree is easy: starting from the root of T, find the first node u such that $l < val(u) < r$. Then, l is inserted in $L(u)$ and r in $R(u)$. Searching for u would at most need to go down a path of the interval tree, thus it takes $O(\log_2 V)$. For fast insertion/deletion, each list is implemented using a balanced binary tree (not shown in Figure 2).



An Interval Tree with $U = \{1, ..., 8\}$ and $n = 5$ Intervals. The backbone binary tree T and the doubly linked lists are shown. The value for each node appears inside the node. The left/right lists for the root node contain the end-points of (2, 6), (2, 5), and (3, 8).

Figure 2 – A simple example of a BIT [KTF98]

The Bitemporal R-tree (BRT) makes an R-tree [BKSS90] partially persistent, following the approach of the MVB [B93] and MVAS [VV97]. Like those structures, the BRT is a directed acyclic graph of pages. The structure is then formed by several logical R-trees, representing the evolution of objects in the transaction time sense.

2.5.1.2 Spatial Indices

Another approach is to view bitemporal data as a special case of spatial data and to adapt spatial indices to bitemporal data. Due to the similarities between bitemporal and spatial data: the combined valid and transaction time of a fact can be treated as a region in two-

dimensional space. Many indices have been developed for spatial data [SAM90]. One of the most robust indices for spatial data with extent (i.e. non-point data) is the R-tree [G84] in its different variants (the R+-tree [SRF87], the R*-tree [BKSS90], and the Hilbert R-tree [KF94]). All variants of the R-tree try to minimize the overlap between the minimal bounding rectangles of the nodes at each level of the tree and to minimize the dead space in the bounding rectangle of each node (dead space is the space in the minimum-bounding rectangle not occupied by any enclosed rectangle. Minimizing overlap reduces the I/O-incurring branching of search into several subtrees. Minimizing dead spaces reduces the probability that queries unnecessarily access disk pages, eventually finding no qualifying data. Detailed description of both R-Tree and R*-Tree is described in the next subsections.

R-Trees

The R-trees are hierarchical data structures, meant for efficient indexing of multidimensional objects with spatial extent. R-trees are used to store, instead of the original space objects, their *minimum boundary boxes* (MBBs). The MBB of a n-dimensional object is defined to be the minimum n-dimensional rectangle that contains the original object. Similar to B-trees, the R-trees are balanced and they ensure efficient storage utilization.

The R-trees manage MBBs and not real objects, thus they cannot fully answer a query, unless the objects in the database are equal to their MBBs. In general, they are used to efficiently solve the *filter* step of a query, that is finding the database objects whose MBB intersects with the MBB of the query object.

I. The R-Tree

The R-tree[G84] is the father of all R-tree variants. Each R-tree node corresponds to a disk page and a n-dimensional rectangle. Each non-leaf node contains entries of the form *(ref, rect)*, where *ref* is the address of a child node and *rect* is the MBB of all entries in that child node. Leaves contain entries of the same format, where *ref* points to a database object, and *rect* is the MBB of that object.

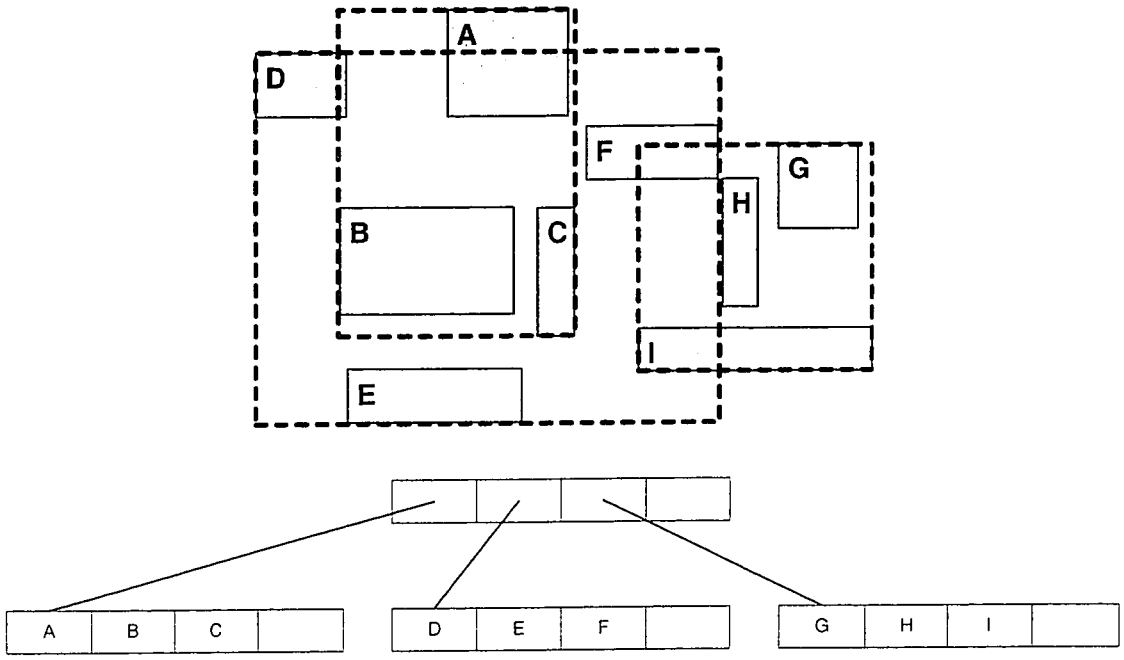


Figure 3- An R-tree for a set of 2-d rectangles

The rest of the R-tree properties include:

- Let M be the number of entries that can fit in a node, and let m be the minimum number of entries per node, $2 \leq m \leq \lceil M/2 \rceil$. Every node contains between m and M nodes, unless it is the root. If the number of entries in a node falls under the m bound after an entry deletion, the node is deleted, and the rest of its entries are distributed among the sibling nodes.

- The root contains at least 2 entries, unless it is a leaf.
- The tree is height-balanced; every leaf node has the same distance from the root. The height of the tree is at most $\lceil \log_m N \rceil$ for n index records ($N > 1$).

Figure 3 illustrates a set of 2-d objects in the plane, stored in an R-tree. The dotted rectangles are the MBBs of the root entries, and the solid rectangles are the MBBs of the objects stored at leaves. Note that the MBBs of entries at the same node may intersect one another.

Searching in an R-tree is done in a similar way as in a B-tree. For both point and region queries, the paths where *rect* intersects with the query object are followed. In contrast to the B-tree, the R-tree does not guarantee that traversing one path of the tree is enough when searching for an object, as the MBBs of entries in the same nodes may overlap one another. In the worst case, the search algorithm may have to visit all index pages, in order to answer a query.

Inserting an object to the R-tree, includes inserting its MBB to the R-tree along with a reference of the object to the *ref* field of the new entry. Only one path of the tree is traversed and the new entry is inserted to a leaf node. If the MBB of the object intersects many entries of an intermediate node, we follow the child whose MBB is less enlarged after the insertion. In case of a tie, we apply other criteria, such as the node's cardinality, or MBB area size. The object is inserted only at one leaf and if it causes the leaf page to overflow, we split the page in two, again after applying several splitting criteria. The split can be propagated to the ancestor nodes. If an insertion causes enlargement of the leaf page's MBB, we adjust it properly and propagate the change upwards.

Deletion in an R-tree requires an exact match query for the object, at first. If the object is found in a leaf, it is deleted. Again the deletion may cause a modification in the tree's structure, as it can cause the leaf page where from it is deleted, to underflow (the number of entries may fall under m). In the case of an underflow, the whole node is deleted, and all its entries are stored in a temporary buffer, and reinserted in the tree. As for insertion, deletion may affect the MBB of the page. In that case, we propagate the change up along the search path.

Minimizing the overlap between sibling nodes is an important issue, concerning the searching performance in an R-tree [RL85]. Guttman [G84] suggests various policies to minimize overlap during insertion. Roussopoulos et al. [RL85] introduce a *packing* technique, which builds optimal R-tree, provided all inserted data is known a-priori. The variant of R-tree that is considered to best handle dynamic object insertion is the R*-tree [BKSS90], discussed next.

II. The R-Tree*

Several weaknesses of the original R-tree insertion algorithms stimulated Beckmann et al. [BKSS90] to work on an improved version of the R-tree, the R*-tree. This version introduces a new insertion policy that significantly improves the performance of the tree. The main objective of this policy is to minimize the overlap region between sibling nodes in the tree. A straightforward advantage of this is the minimization of the tree paths that are traversed at an object search. The advantages of the new insertion algorithm over its R-tree respective can be summarized as follows:

- While traversing the insertion path, the insertion algorithm follows the nodes, whose MBB has the minimum increase of overlap. Thus, the search performance is improved [RL85].
- Whenever a new entry has to be stored into a full node, the node is not necessarily split, but some entries are deleted, and re-inserted to sibling nodes. The entries for re-insertion are chosen to be those with maximum distance from the center of the node's MBB. This feature of the algorithm increases storage utilization, and improves the quality of the partition, making it almost independent of the sequence of insertions.
- The algorithm for splitting a node is totally different from its R-tree equivalent. First, the algorithm decides the axis with respect to which the split will take place. Then, the projections of the MBBs over the split-axis are sorted according to the value of their left end point. This sequence can be divided to two sub-sequences, in M-2m+1 ways. Among these splits, the algorithm chooses the one that results in a minimum overlap between the MBBs. This algorithm is proved to achieve better quality of the MBBs partition over the tree.

Figure 4 shows an R*-tree for a set of 2-d rectangles.

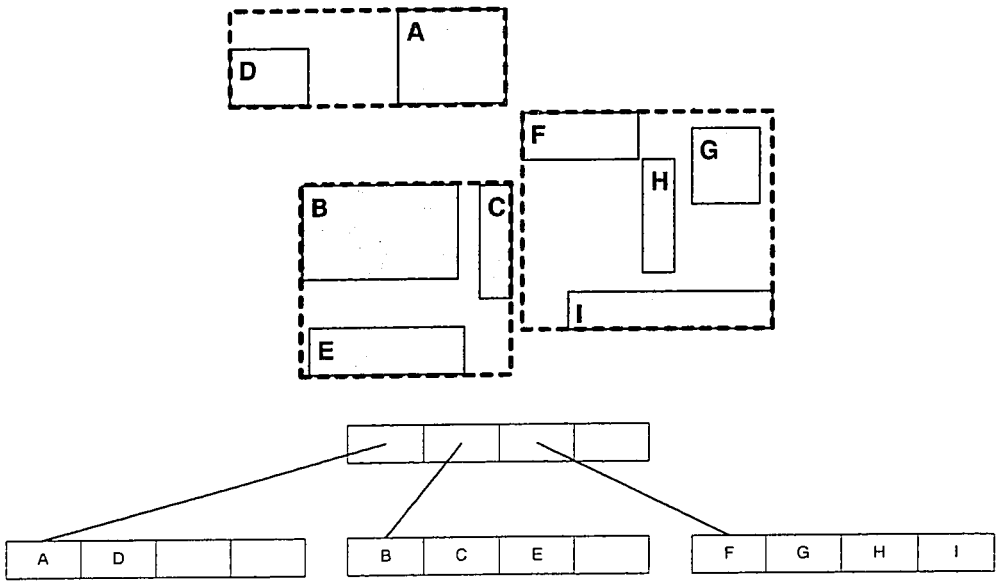


Figure 4 - An R*-tree for a set of 2-d rectangle

Example for an R*-tree:

Consider the rectangles in Figure 5. Suppose the rectangles with solid edges are the object rectangles, i.e., they are the basic MBB of two-dimensional spatial objects in the database. The bottom of Figure 5 shows the corresponding R*-Tree with M =3 and m =2. Each rectangle with dotted edges represents a directory rectangle. Directory rectangles are numbered in boldfaced numbers of the form R*i*. Note that rectangles R2 and R3 both contain rectangle F. However, only rectangle R3 is the parent of rectangle F.

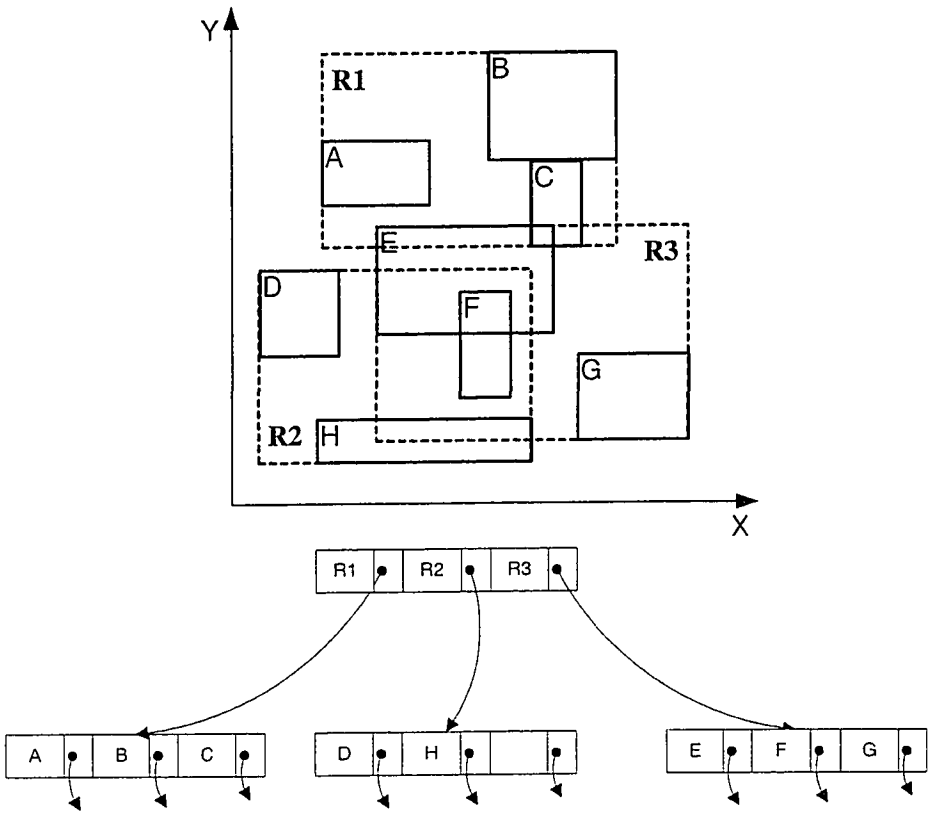


Figure 5 – Example of R*-Tree

Let's now assume that we want to find all object rectangles in the R*-Tree that overlap with rectangle S shown in Figure 6 with bold dashed edges. When the root node of the R*-Tree is searched, MMBs R1 and R3 are found to overlap with S. Then there is a possibility that an object rectangle contained in these MBBs will overlap with S. This means that pointer R1 and pointer R3 should be followed so that the child rectangles can be examined against S. Since there is no order among the cells in the node, we have to compare S with the rectangle of every cell in the node in order to avoid missing any qualified object rectangle. Among the child rectangles of MBB R1, rectangle C overlaps with S. Among the child rectangles of MBB R3, rectangles E and G overlap with S.

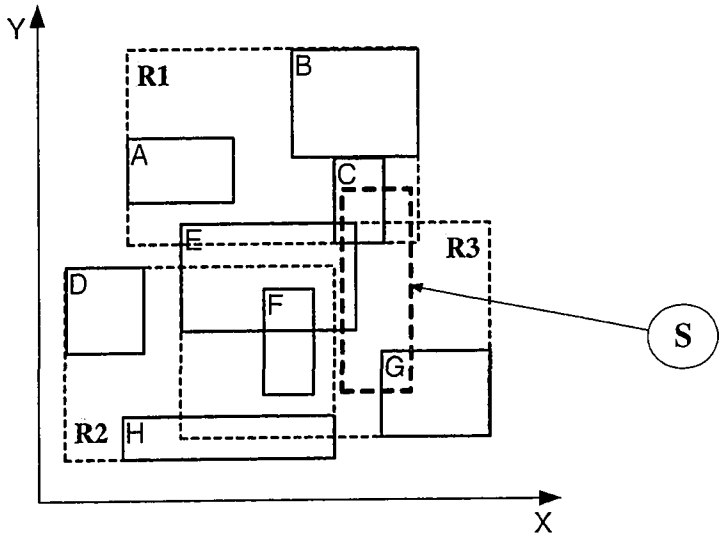


Figure 6 – Search for Overlapping Object Rectangles with S

Let's now consider the insertion of an object rectangle into the R*-Tree shown in Figure 5. Suppose we have inserted the object rectangle K shown with the boldface dotted edges in Figure 7.

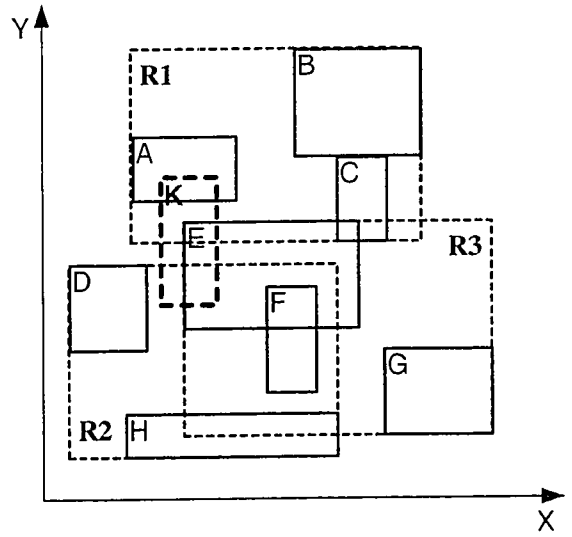


Figure 7 – Inserting rectangle K in the R*-Tree

We start from the root node of the R*-Tree. Since it is the parent of leaf nodes, we find the MBB in the root node that needs the least overlap enlargement to contain rectangle K. Suppose rectangle R1 is the MBB found. Since this node does not have enough room (recall that each node is assumed to accommodate at most three cells) to accommodate the new cell, an overflow occurs. Since this is the first overflow at the leaf level, a reinsertion is invoked. The reinsertion algorithm, proposed by Beckmann et al., sorts the entries in decreasing order of the distance between the centroids of the rectangle object and the MBB and reinserts the first p (variable for tuning) entries. Suppose one cell is to be reinserted. Suppose further that object rectangle A is selected for reinsertion. MBB R1 needs to be adjusted as a result (see Figure 8).

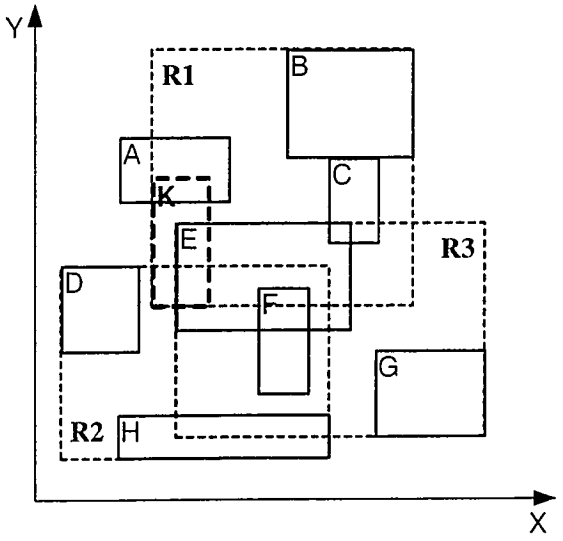


Figure 8 – Selecting A to be reinserted as a result of overf low in R1

For this example, the reinsertion brings us back to the same leaf node and causes another overflow. This time, we need to apply the split process to partition the four cells (three originally in the node plus the new cell) into two nodes.

A sort of the four cells along the X-axis produces {A, K, B, C}, and a sort along the Y-axis produces {K, C, A, B}. For each axis, there is only one way to partition the four cells into two sets since m = 2 for our example. For the sort along the X-axis, the partition is {(A, K), (B, C). For the sort along the Y-axis, the partition is {(K, C), (A, B)}. Clearly, the sum of the perimeters of the MBB for (A, K) and (B, C) is smaller than that for (K, C) and (A, B). Thus the X-axis is chosen as the split axis. Consequently, two new nodes containing {A, K} and {B, C} will be created to replace the original node.

Let the MBB for (A, K) be denoted R4 and that for (B, C) as R5. Then two new cells will be created for the current root node to replace R1 (see Figure 9). Since there is not enough space in the root node to hold the four cells, an overflow occurs at the root level. Since this is the root level, the reinsertion process is not invoked. Instead a split is performed.

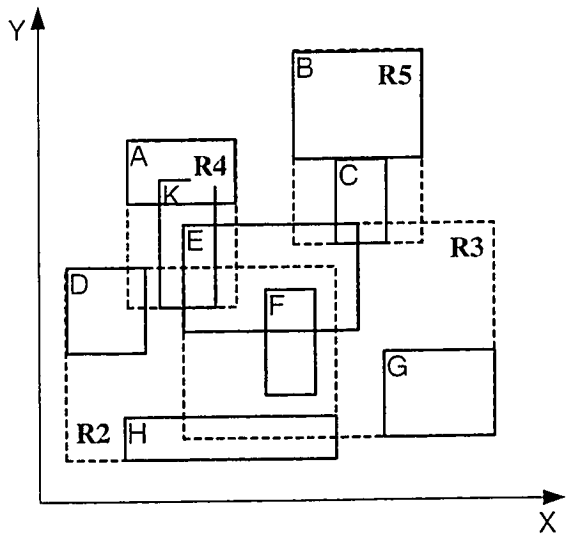


Figure 9 – R1 was split into R4 and R5

By using the same method, a split along the Y-axis will be performed. As a result, two new nodes replace the original root node, one containing (R4, R5) and the other

containing (R2, R3). Let the MBB for (R4, R5) be denoted R6 and that for (R2, R3) as R7. A new root node with two cells corresponding to R6 and R7 will be generated as a result. The final result is shown in Figure 10.

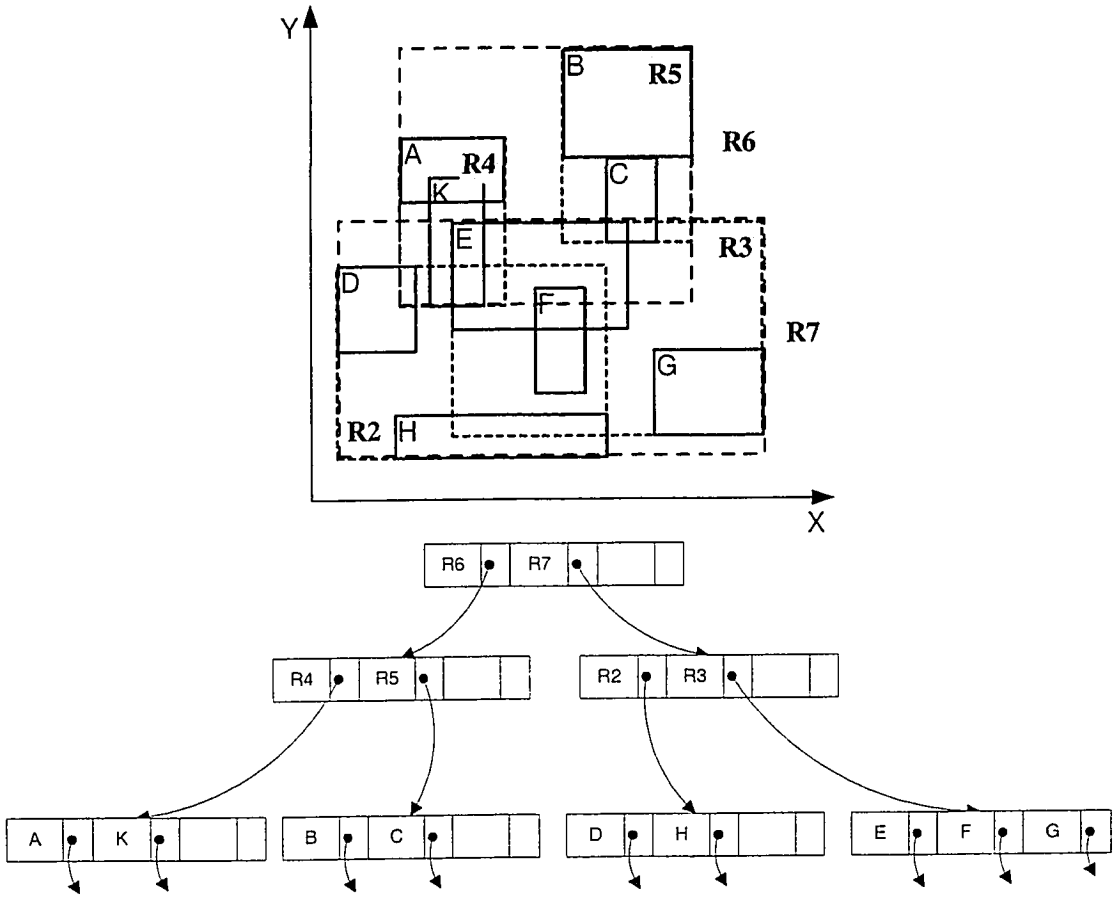


Figure 10 – Final result after inserting rectangle object K

Finally we consider the case of deleting the rectangle object D from the R*-Tree shown Figure 10. We first locate the node that should contain the cell corresponding to rectangle D. Starting from the root, we find that it is fully contained in directory rectangle R7 (note that D only overlaps with R6). Now we check containment in both directory rectangles R2 and R3. Rectangle D is fully contained in R2. Since we reach the leaf node, we check

for the exact match with rectangle D, and we allocate the cell to be deleted. Deleting cell D from this node causes an underflow to occur (i.e. the number of remaining cells in the node is less than m). Rather than merging the underflow node with its sibling node as in a B^+-Tree, this node is removed from the R*-Tree and the remaining cell H is reinserted into the R*-Tree.

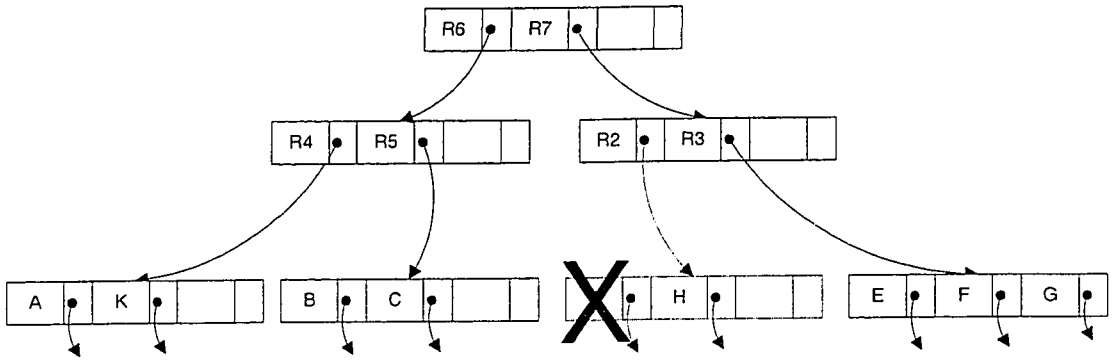


Figure 11 - Removing R2 node because of underflow

But removing the leaf node (H) means that we remove the MBB R2 from its parent node which results in another underflow in the parent node (see Figure 12). This means that we need to remove the parent node and reinsert its rectangle objects E, F, and G.

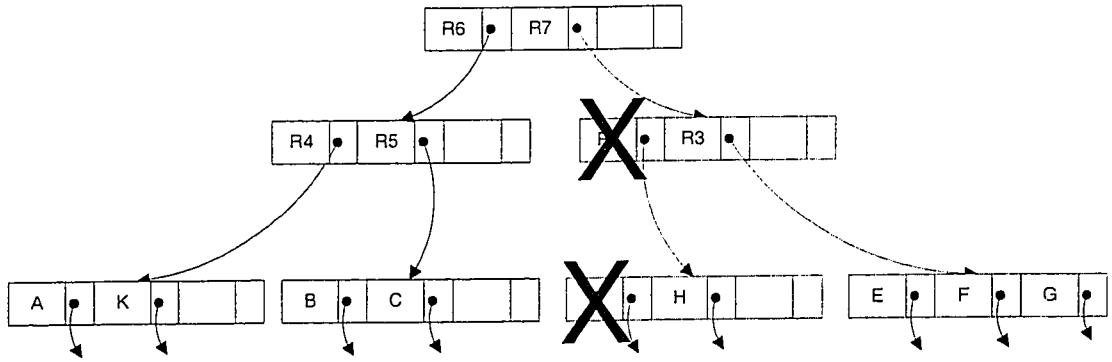


Figure 12 - Removing R7 node because of underflow

The underflow propagates all the way to the root node as shown in Figure 13.

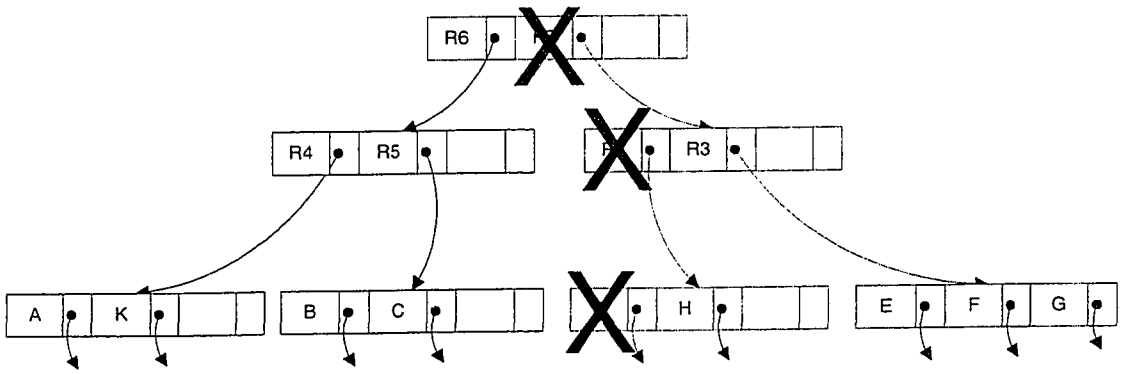


Figure 13 – Removing the R*-Tree root because of underflow

Now, we make the child node containing (R4, R5), the new root node and start adding the rectangle objects (H, E, F, G) back into the R*-Tree. We sort the entries in increasing order of the distance between the centroids of the rectangle objects and the MBBs and then we reinsert the entries. Assume, in our example, that we reinsert entries in following order (E, F, H, G). E will be inserted in R5 as shown in Figure 14.

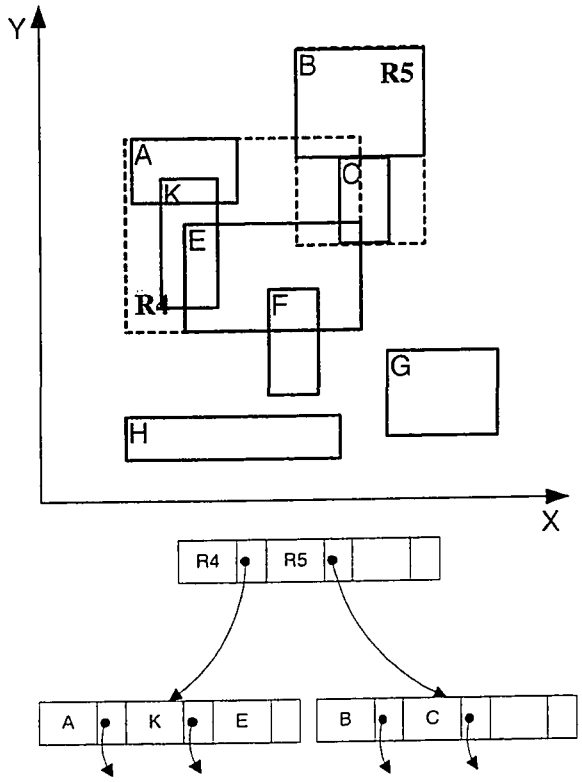


Figure 14 – Reinserting E in R4

Now inserting F in R4, will result in an overflow. A sort of the four cells along the X-axis produces { A, K, E, F}, and a sort along the Y-axis produces {F, E, K, A}. Both results in the same partition (A, K) and (E, F). Thus, R4 is split into R8 and R9 as shown in Figure 15.

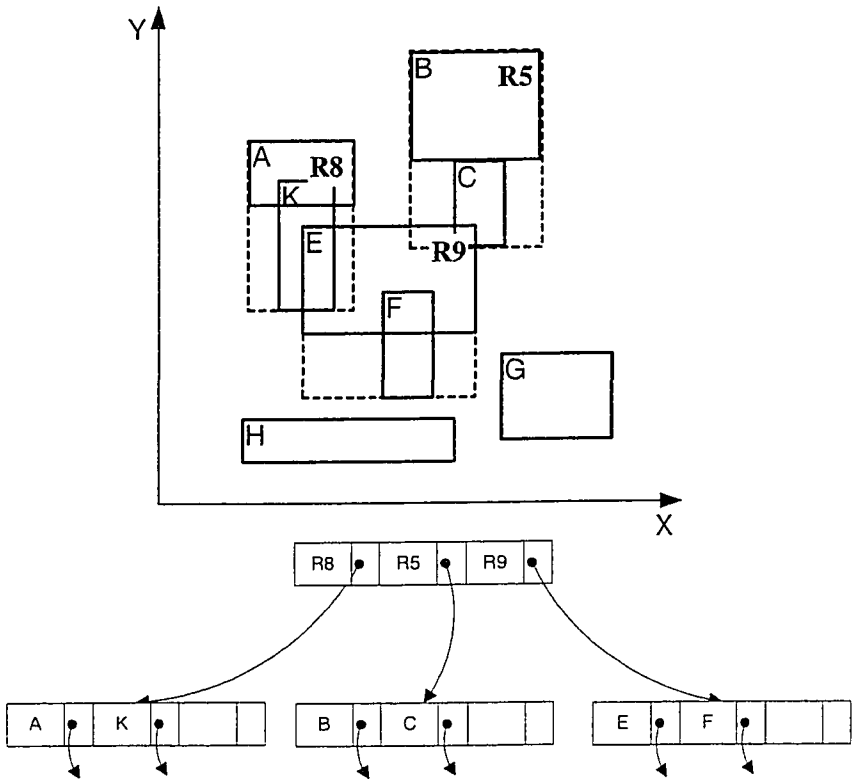


Figure 15 – Splitting R4 into R8 and R9 after reinserting F

To reinsert object rectangle H, we start from the root node of the R*-Tree. Since it is the parent of leaf nodes, we find the MBB in the root node that needs the least overlap enlargement to contain rectangle H. Suppose rectangle R9 is the MBB found. This time, R9 has enough room for E and the R*-Tree now looks as shown in Figure 16.

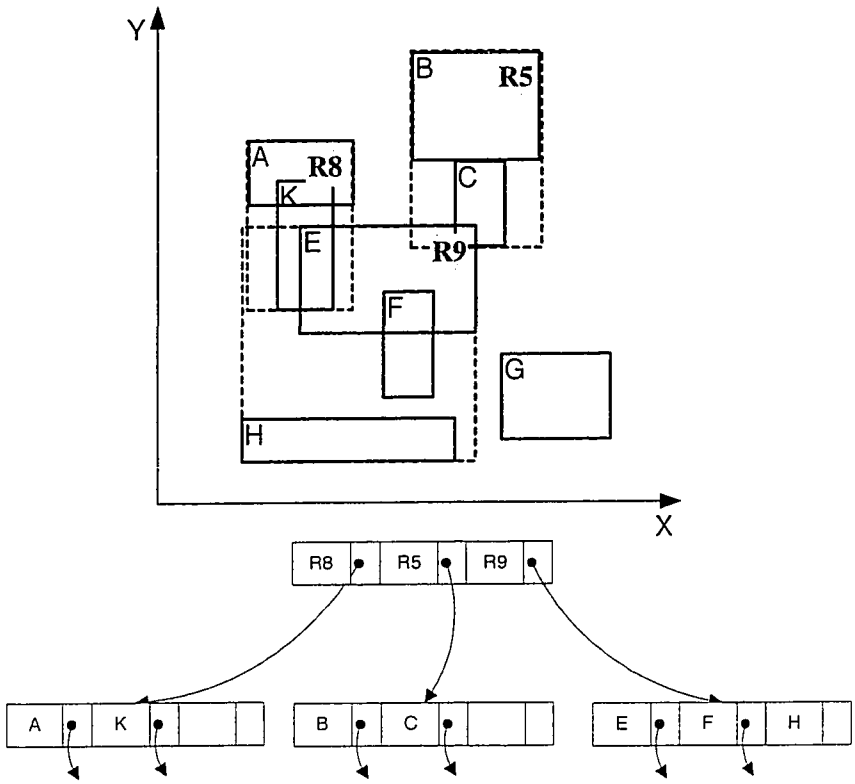


Figure 16 – Reinserting H in R9

Finally we insert object rectangle G. We start from the root node of the R*-Tree. Since it is the parent of leaf nodes, we find the MBB in the root node that needs the least overlap enlargement to contain rectangle G. Suppose rectangle R9 is the MBB found. R9 is full and applying reinsertion does not solve the problem. We have to apply the split algorithm that results into new MBBs R10 that includes (E, F) and R11 that includes (G, H). The root node will also need to be split resulting in the R*-Tree shown in Figure 17.

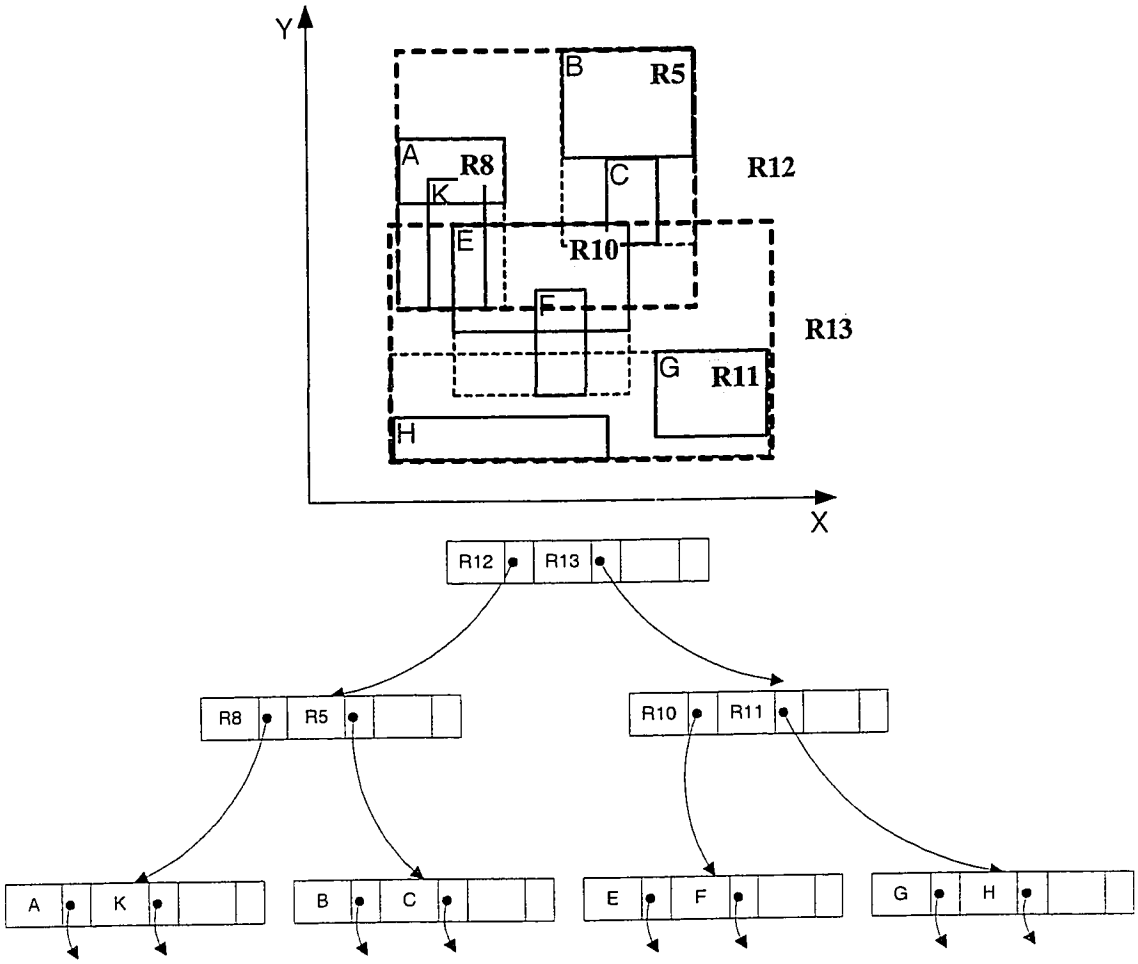


Figure 17 – Final R*-Tree structure after reinserting all rectangle objects

2.5.2 Access Methods for Now-relative Bitemporal Data

The bitemporal indices generally fall short in efficiently supporting now-relative data [CLI97], data for which the end of the valid time or/and transaction time tracks the progressing current time. Now-relative data occurs naturally in most real-world databases. For example, consider the recording of a new employee in a company's database. The time when the employee starts working (valid-time interval begin) is known, but it is unknown when the employee will leave. Letting the valid-time end extend to the progressing current time captures this. The same applies to transaction time.

The begin of transaction-time interval of a tuple is the time when it is inserted into the database. Since we do not know when the tuple will cease being current, the transaction-time end extends to the current time.

The 2-R index and the Bitemporal R-tree [KTF98], efficiently support now-relative transaction time, but not now-relative valid time.

The 2R-tree uses two R-trees (named *front* and *back* R-trees) to index bitemporal data. The bitemporal domain is mapped to a two-dimensional space (valid time × transaction time) as follows: An object with an unknown transaction end time is stored in the front R-tree as a line. Recall that in this approach the valid time ranges are bounded and, naturally, the transaction start time is always known. Once this object is updated, it is removed from the front R-tree and is inserted into the back R-tree as a rectangle. The interval based 2R approach is described visually in Figure 18, while Figure 19 describes the point based 2R approach. The front R-tree indexes two objects, inserted at (transaction) time t_s and t_s' and which are still current in the database, i.e., bear an open transaction end time. The back R-tree, on the other hand, indexes two other objects which were current in the database during $[t_s, t_f]$ and $[t_{s'}, t_f']$ respectively.

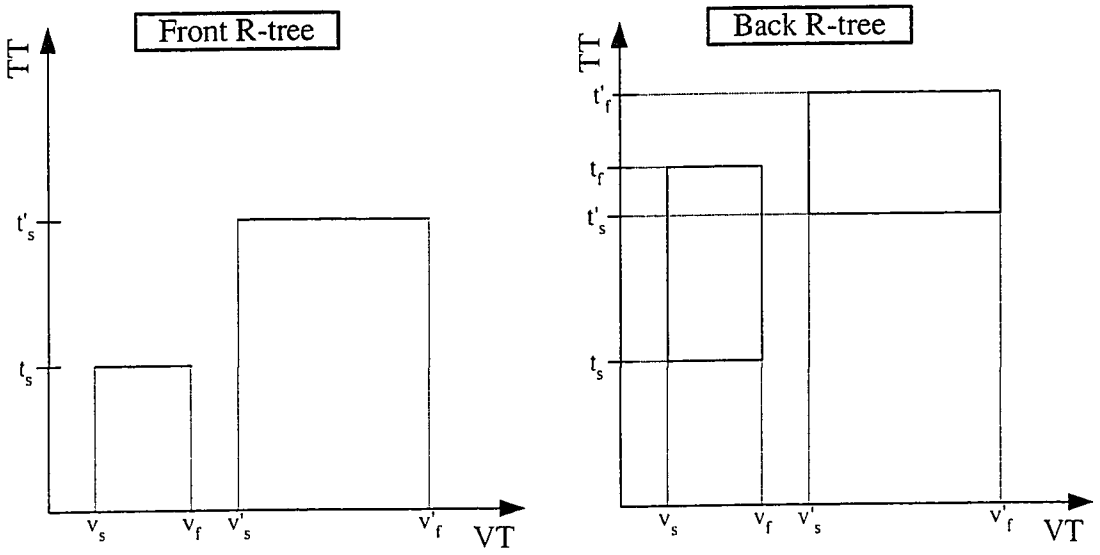


Figure 18 - The interval based 2R approach (2Ri)

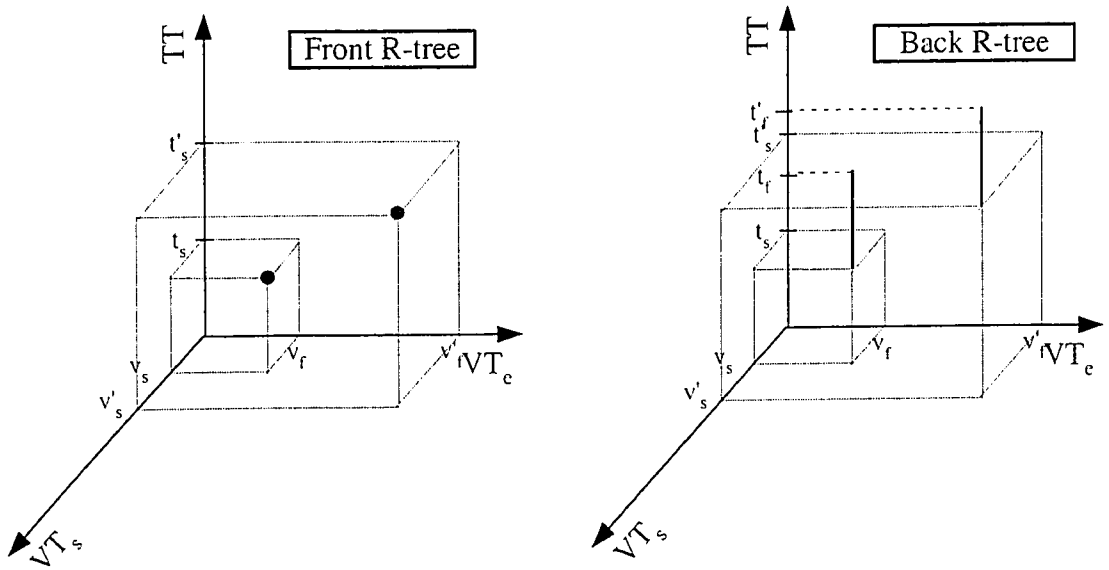


Figure 19 - The point based 2R approach (2Rp)

The GR-tree [BJSS98] and 4R-tree [BJSS00] index both support now-relative valid and transaction time. The GR-tree (shown in Figure 20) extends the R-tree to store both static

tuples (with closed valid and transaction time ranges) and growing objects (with either valid or transaction end time unknown). In this new tree, the indexed objects in its nodes can be either a growing rectangle or a growing stair-shape object, in addition to the standard MBBs supported by the R-tree. By storing such growing objects, the dead space among objects in the GR-tree is decreased when compared to using the R-tree and hence it becomes much more efficient.

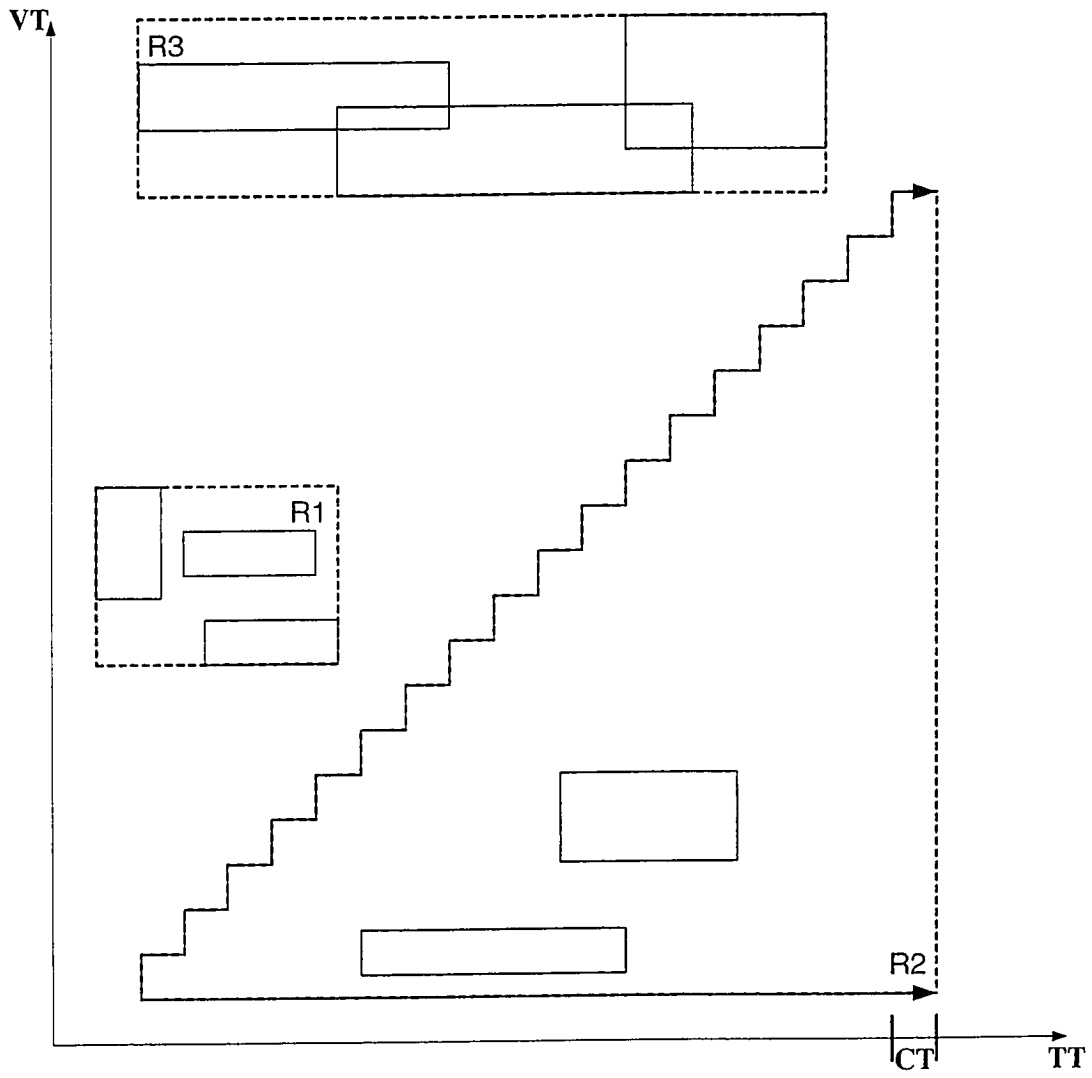


Figure 20 – GR-Tree with Minimum Bounding Region of Node 2

To reduce dead space, the 4R-tree maps a growing rectangle into a closed line and a growing stair-shape object to a point. Using such a transformation the proposed approach is able to use "off-the-shelf" R-trees (which is the main goal of the proposal). Indeed, the objects are indexed in four R-trees, depending on whether their valid and transaction end time are open or not. As objects are updated they may move between such R-trees like in the 2R-tree approach. In fact, it is interesting to note that in the case of bitemporal data with no now-relative valid time the 4R-tree reduces to the 2R-tree.

2.6 Motivation for our proposed access method for Now-relative Bitemporal Data

The 2-R index is efficient and easy to implement because it is using the off-the-shelf R-Trees. However, it does not support now-relative valid-time.

While the GR-Tree solves the main drawback of the 2-R index, its implementation requires the development of a new access method, which means it cannot be simply added to existing DBMS.

Finally the 4R-Tree possesses the advantages of both the 2-R index and the GR-Tree. That is it is using off-the-shelf access method, the R*-Tree, and provides now-relative support for both the valid- and transaction-time.

[BJSS00] has reported possible enhancements in the query performance of the 4R-Tree and [KTF98] has proved that the point-based 2-R performance is much better than its corresponding interval-based. Since the 4R-Tree is rectangle-based (interval-based), we anticipate that a point-based version of the 4R-Tree will have much better performance.

Both versions of the 4R-tree are described in details in Chapter 3.

Chapter 3 Point-based 4-R Index for Now-Relative Bitemporal Data

3.1 Now-Relative Bitemporal Data

Two temporal aspects of database tuples, termed valid and transaction time, have proven to be of interest in a wide range of database applications. Valid and transaction time are two orthogonal dimensions in semantics, i.e. the valid time when the data is true in the modeled reality is independent of the transaction time during which the data is current in the database [C+98]. The valid time of a tuple can be in the past or in the future (allowing a database to store information about the past and the future) and can be changed freely. In contrast, the transaction time of a tuple cannot extend beyond the current time and cannot be changed. Data having associated both valid and transaction time is termed bitemporal data. Bitemporal data is now-relative if the end of valid time or the end of transaction time is not fixed, but instead tracks the current time and continuously extends as time passes.

Now is a distinguished timestamp value in many temporal data model proposals. In the next subsection, we discuss the informal semantics of this familiar term and its representation.

3.1.1 Now in Valid Time

A common use of *now* is to indicate that a fact is valid until the current time. For example, suppose that Empl began working as a 'S/W Development Director' on Jan. 1, 2000. Table 4 shows the relevant tuple from the **EmpTitle** Table.

Table EmpTitle		
EmpID	**Title**	**Valid Time**
Empl	S/W Development Dir.	Jan. 2000 -> Now

Table 4 - Empl's employment tuple

Empl started working as a 'S/W Development Director' on Jan. 2000, as indicated by the "valid time" attribute (for the examples in this chapter we assume a timestamp granularity of one month). The variable *now*, appearing as the terminating datetime in the valid-time period for Empl's employment tuple, represents a currently unknown future time when Empl will stop working for the company or will have a different tittle. The result of a query that requests the current employees will include Empl.

The informal semantics of this value is that Empl is a 'S/W Development Director' until we learn otherwise. As the current time inevitably advances, the interpretation of *now* also changes to reflect the new current time. Some authors have called this concept "until changed" instead of "now" [WJL91][WJL93], but the semantics is the same.

Other data models use *forever* or ∞ as the terminating period datetime, as shown in Table 5. Forever is the largest representable timestamp value, that is, the one furthest into the future. This value admits that we do not know when Empl will depart the company, and so assumes that she will be working forever.

Table EmpTitle2		
EmpID	**Title**	**Valid Time**
Emp1	S/W Development Dir.	Jan. 2000 -> *forever*

Table 5 - Emp1's employment tuple using *forever* as an upper bound for valid time

One limitation of using *forever* is that it is overly optimistic: forever is a long time into the future! In SQL and in IBM's DB2, forever is about 8,000 years from the present [DW90][MS93]; in TSQL2's more liberal design, it is approximately 18 *billion* years from the present time [DS93]. Hence, to assert that Emp1 will be employed until forever is most surely incorrect. A related implementation limitation is that, bitemporal regions will be represented in R-Trees using static rectangles that extend to the maximum possible transaction- and valid-time values. As a consequence, the MBBs in internal tree nodes also extend to the maximum values, resulting in excessive dead space and overlap as shown in Figure 21.

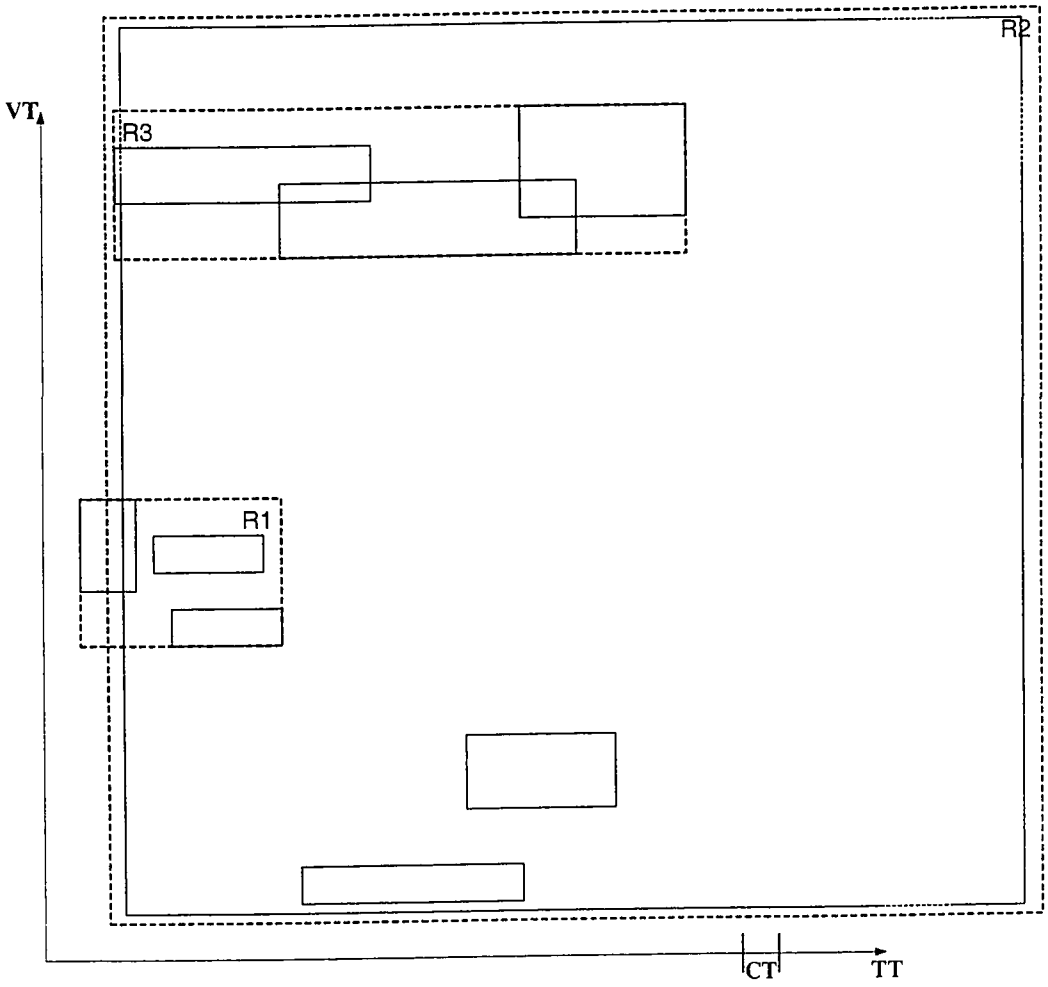


Figure 21 - Indexing Growing Bitemporal Regions Using Maximum Timestamp (forever) Values

An alternative way to view this problem is that there is a difference between the *actual* and *expected* times of a fact. On a day-to-day basis, we expect Empl to remain employed. A database that uses *forever* as the terminating time of her employment tuple (very optimistically) records her expected employment, while a database that uses *now* records only her actual employment, the time she has worked to the current time.

3.1.2 Now in Transaction Time

Transaction time denotes the time period between a fact being stored in the database and the fact being (logically) deleted from the database [SA85]. It is an orthogonal concept to valid time, in that it concerns the history of the database, as opposed to the history of the enterprise being modeled.

Transaction-time timestamps are supplied automatically by the DBMS during updates (valid-time timestamps are generally supplied by the user). Specifically, insertions initialize the starting transaction time to the "current time" and the terminating transaction time to *now*. (There is an additional requirement that the transaction time be consistent with the transaction serialization order.) Updates change the terminating time of *now* to the value of the current transaction time. Hence, in transaction-time tables, deletion is logical. The information is not physically removed from the table; rather it is tagged as no longer current by having a terminating time different from *now*. Physical deletion never occurs in a transaction-time table.

A more precise label than "now" or "forever" for the transaction-time concept of "not yet logically deleted" is "until changed." The most recent transaction for a fact is considered the current state of that fact, *until changed* by a later transaction.

Querying the current state, i.e., in a rollback operation, considers all tuples with a terminating time of *until changed*, and no other tuples. In [BJSS98], Jensen & als used the label "*until changed*" instead of the label "*now*" in transaction time to make clear the special, transaction-time specific meaning of *now*, and to ensure that updates are consistent with, and in fact a refinement of, currently stored information. In the remainder of this thesis report, we will follow the same notion.

3.1.3 Two-dimensional Bitemporal Regions

This section describes the characteristics of the different kinds of two-dimensional bitemporal data regions. Using TQuel's four-timestamp format [SNO87], each tuple has a number of non-temporal attributes and four time attributes: VT^s and VT^e— the times when the tuple's information became and ceased to be true in the modeled reality; TT^s and TT^e— the times when the tuple became and ceased to be current in the database. A tuple is now-relative if its information is valid until the current time or if the tuple is part of the current database state. This is represented in the 4TS format by the use of the variable ***NOW*** for VT^e, and the variable ***UC*** ("until changed") for TT^e.

In the example in Table 6, the time granularity is a month, and the current time is July 2000.

Rec #	EmpID	Salary	Valid Time		Transaction Time	
			VT^s	VT^e	TT^s	TT^e
1	Emp1	70,000	Mar. 2000	*NOW*	Mar. 2000	*UC*
2	Emp2	50,000	Jan. 2000	Mar. 2000	Jan. 2000	*UC*
3	Emp3	40,000	Apr. 2000	Jun. 2000	Jan. 2000	May 2000
4	Emp4	60,000	Jan. 2000	*NOW*	Jan. 2000	May 2000
5	Emp4	60,000	Jan. 2000	May 2000	Jun. 2000	*UC*
6	Emp5	30,000	Jan. 2000	*NOW*	Mar. 2000	*UC*

Table 6 - EmpSalary Bitemporal Relation

- Record #1 records the fact that "Emp1's salary is 70,000 starting from Mar. 2000 until the current time", it was recorded in the database in Mar. 2000 and remains as part of the current database state until now.
- Record #2 records the fact that "Emp2's salary is 50,000 starting from Jan. 2000 till Mar. 2000", it was recorded in the database in Jan. 2000 and is still current.
- Record #3 records the fact that "Emp3's salary was 40,000 between Apr. 2000 and Jun. 2000", it was recorded in the database in Jan. 2000 and logically deleted in May 2000.
- Record #4 records the fact that "Emp4's salary is 60,000 starting from Jan. 2000 until the current time", it was recorded in the database in Jan. 2000 and logically deleted in May 2000.
- Record #5 is a modification of the fact stored in record #4. It corrects the fact to read as follows "Emp4's salary is 60,000 starting Jan. 2000 till May 2000", this modification was recorded in the database in May. 2000 and is still current.
- Record #6 records the fact that "Emp5's salary is 30,000 starting Jan. 2000 until the current time", it was recorded in the database in Mar. 2000 and is still current.

Specific constraints apply to insertion, deletion, and modification of tuples.

The constraints for insertions are:

- Valid Time: $VT^s \leq VT^e$ and $VT^s \leq$ 'current time' if VT^e is equal to *NOW*; and
- Transaction Time: TT^s = 'current time' and $TT^e = UC$.

These constraints partition the data space and force the corresponding data points to be allocated in a subspace bounded by one or more of the lines $VT^s = VT^e$, $TT^s = TT^e$, and $VT^s = TT^s$ (see Figure 32 and Figure 33).

Any *current* database tuple can be logically deleted or modified. To delete a tuple, the TT^e value *UC* is changed to the fixed value 'current time' - 1, making the tuple not current anymore (for example, Tuple 3). It is important to note that tuples are not physically deleted. A modification is modeled as a deletion followed by an insertion (for example, an update led to Tuple 4 and Tuple 5).

The temporal aspect of a tuple can be represented graphically by a two-dimensional ("bitemporal") region in the space spanned by valid and transaction time [JS96]. Cases 1–4 in Figure 22 illustrate the *bitemporal regions* of Tuples 1–4, respectively.

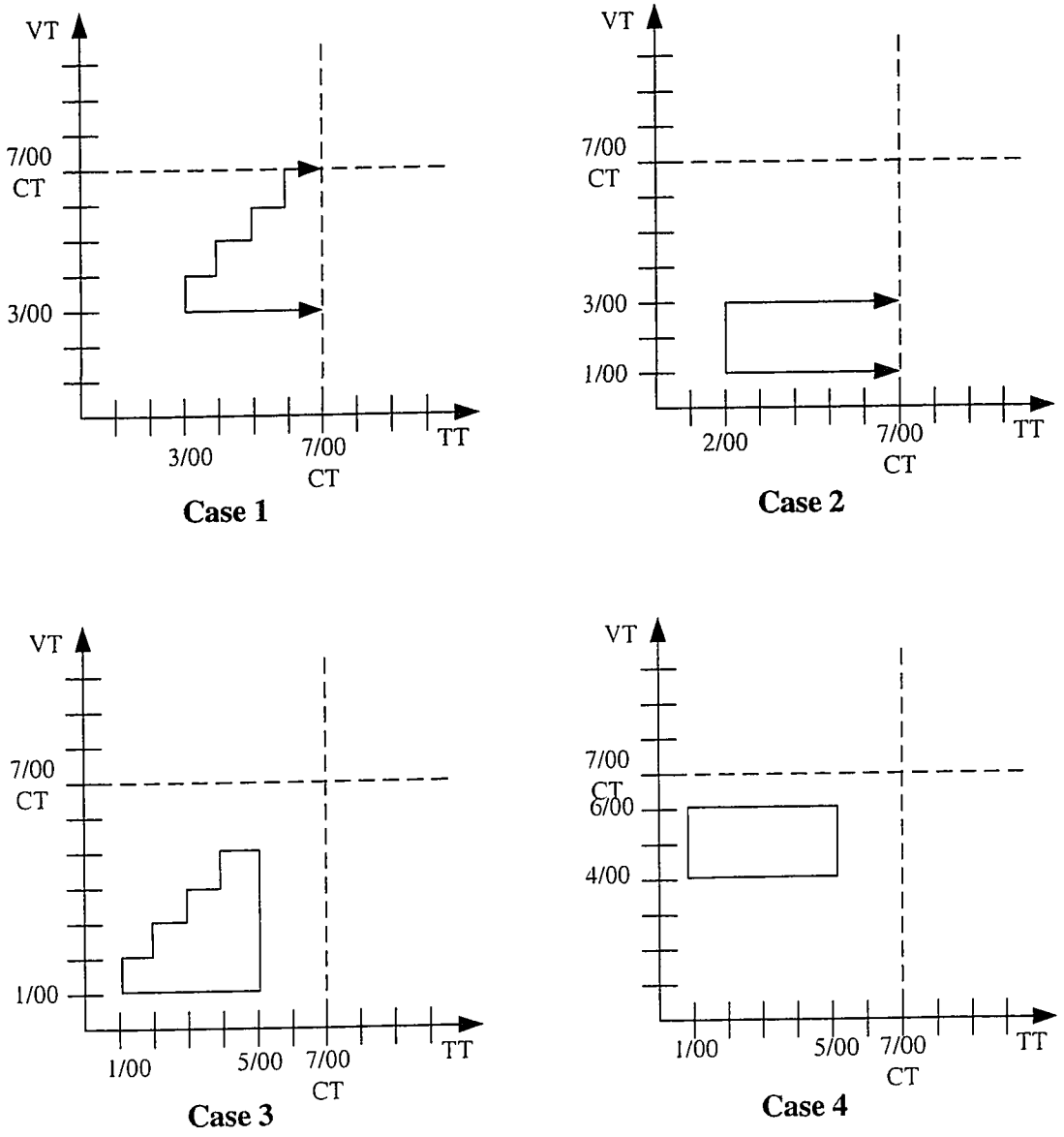


Figure 22 - Two-dimensional Bitemporal Regions (adapted from [BJSS00])

Generally, there are four combinations of time attributes for which bitemporal regions are qualitatively different. The four cases are represented graphically in Figure 22 and summarized in Table 7, where tt_1, tt_2, vt_1 and vt_2 are fixed timestamps that conform to the constraints given above.

	VT^s	VT^e	TT^s	TT^e	constraints
Case 1	vt_1	*NOW*	tt_1	*UC*	$tt_1 \geq vt_1$
Case 2	vt_1	vt_2	tt_1	*UC*	
Case 3	vt_1	*NOW*	tt_1	tt_2	$tt_1 \geq vt_1$
Case 4	vt_1	vt_2	tt_1	tt_2	

Table 7 - Four possible combinations of time attributes

3.1.3.1 Case 1

A fact that is now-relative for both its valid- and transction-time. A stair-shaped region growing in both the valid- and transaction-time represents this.

In the case that the fact was stored retro-actively (i.e. start of valid time < start of transaction time), then the stair-shaped region will have a high first step (for example, record # 6 in Table 6).

3.1.3.2 Case 2

A fact that both its start and end valid-time are known at the time when it was recorded in the database and still current. This is represented by a growing rectangle in the transaction-time.

3.1.3.3 Case 3

Case 3 is the logical deletion of case 1. Specifying the end for transaction-time represents a logical deletion. This means that the growth of the stair-shaped region of case 1 will stop and a static stair-shaped region represents the bi-temporal information.

3.1.3.4 Case 4

Case 4 is the logical deletion of case 2. Specifying the end for transaction-time represents a logical deletion. This means that the growth of the rectangle of case 2 will stop and the bi-temporal information is represented by a static rectangle.

3.2 The Rectangle-based 4R-tree

This section provides a detailed description of the 4-R indexing technique. The idea behind the technique is to apply data transformations that render the continuously growing (now-relative) bitemporal data regions static. To answer queries using this technique, the queries must go through a series of transformations described in the following subsections.

3.2.1 Data Transformation

In this section, we will describe how we:

- divide bitemporal data regions into four classes corresponding to the 4 cases described above,
- perform transformations of the regions in each of the classes, thus eliminating any variables, and
- use separate R-trees for indexing the transformed data regions of each class.

During the bitemporal data transformation, we also use the property of the stair-shaped region where these regions are always bounded by the line VT = TT.

3.2.1.1 Tree R1 UcNow

Tree R1 indexes case 1 where both VT^e and TT^e are variables. The transformed region is represented by 2-dimensional points $<VT^s, VT^s, TT^s, TT^s>$. The actual region is the region bounded by the transformed point, the line VT = TT, and the line TT = *CT*.

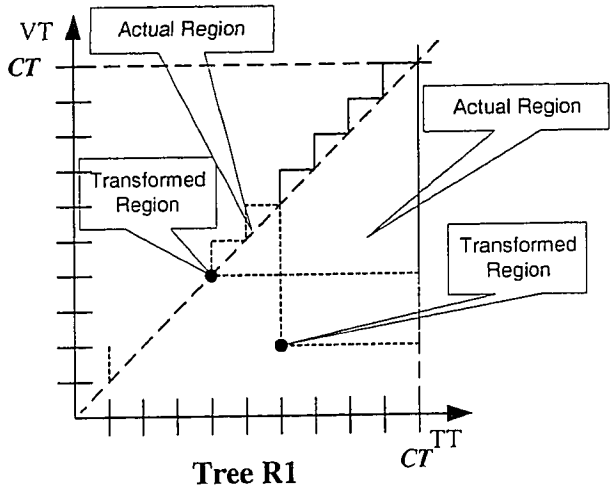


Figure 23 – Tree R1: VT^e = NOW and TT^e = UC

3.2.1.2 Tree R2 Uc

Tree R2 indexes case 2 where the only variable is TT^e. The transformed region is represented by 2-dimensional intervals $<VT^s, VT^e, TT^s, TT^s>$. The actual region is the region bounded by the transformed interval and the line TT = *CT*.

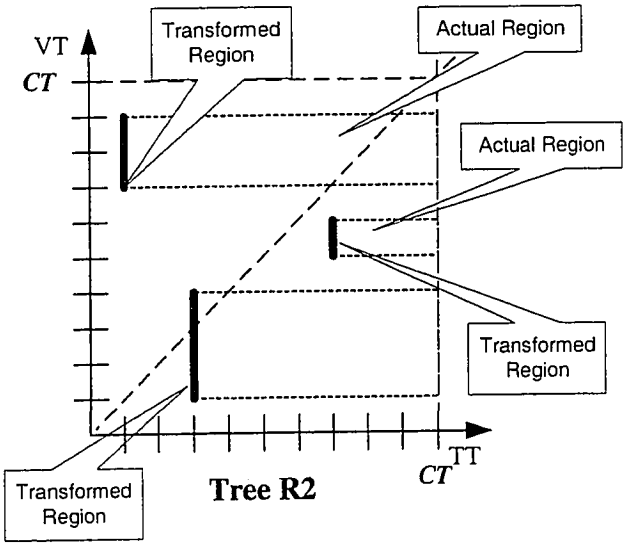


Figure 24 – Tree R2: $VT^e \neq$ NOW and TT^e = UC

3.2.1.3 Tree R3 Now

Tree R3 indexes case 3 where the only variable is VT^e. The transformed region is represented by 2-dimensional intervals < VT^s, VT^s, TT^s, TT^e>. The actual region is the region bounded by the transformed interval and the line VT = TT.

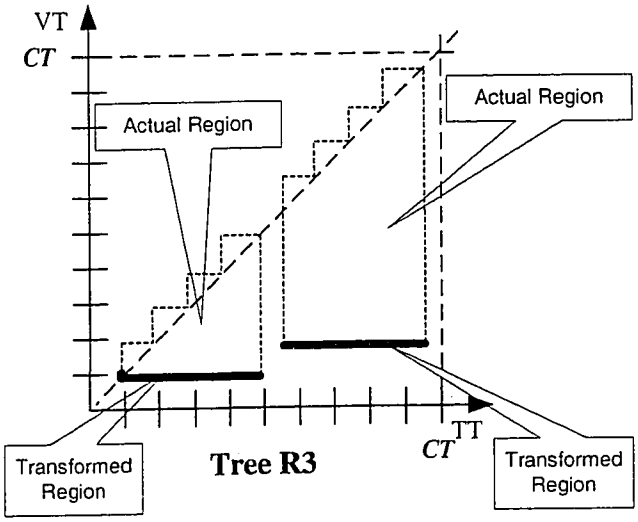


Figure 25 – Tree R3: VT^e = NOW and $TT^e \neq$ UC

3.2.1.4 Tree R4 NoVar

Tree R4 indexes case 4 where all temporal values are ground values and hence no transformation is required.

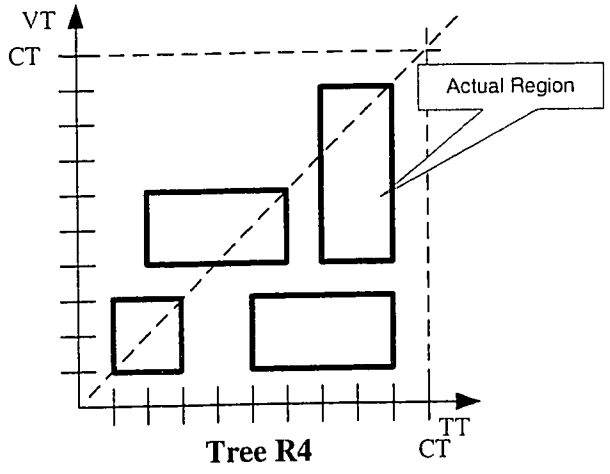


Figure 26 – Tree R4: $VT^e \neq$ NOW and $TT^e \neq$ UC

3.2.2 Query Transaformation

[BJSS00] investigates the most common type of index query, namely the intersection query.

Let $<VT^{l}_{q}, VT^{u}_{q}, TT^{l}_{q}, TT^{u}_{q}> \in T \times T \times T \times T$ denote the argument rectangle of an intersection query, where:

- T is the domain of timestamps,
- $VT^{l}_{q} \leq VT^{u}_{q}$
- $TT^{l}_{q} \leq TT^{u}_{q}$
- $TT^{u}_{q} \leq \boldsymbol{CT}$ (where $\boldsymbol{CT}$ is the value of the current time).

The search spreads to two or four trees and is performed differently in each tree. The following subsections discuss the search in each of the four trees.

Several noteworthy special cases occur when querying the four trees. In trees R1 and R3, the indexed points and intervals reside only on or below the line VT = TT, and the regions encoded by these points or intervals also do not extend above this line. Thus, search in these trees is only performed when at least a part of the search rectangle goes below VT = TT.

Another special case is the current-time transaction-timeslice query ($TT^{l}_{q} = TT^{u}_{q} = \boldsymbol{CT}$), which is expected to occur frequently in practice. Current data resides only in trees R1 and R2, so this timeslice query may be restricted to these two trees, ignoring trees R3 and R4. If, in addition, a current time transaction-timeslice query is above the line VT = TT, the only tree to be searched is R2. As a final special case, if such a current-time transaction timeslice has no constraints on valid time, all bitemporal tuples indexed by trees R1 and R2 should simply be returned, and no search is required.

3.2.2.1 Search in Tree R1

Searching tree R1, the argument search rectangle is enlarged to cover the space spanned from the origin of the transaction and valid time to the argument rectangle's top-right corner. Tree R1 contains no data points above the line VT = TT because the original regions encoded by the points in this tree extend only up to that line. Thus, the transformed search rectangle could also be reduced to not extend above the line VT = TT without affecting correctness. But this reduction of the search rectangle also does not improve performance because that additional area is empty, so for simplicity, the unreduced rectangle is used.

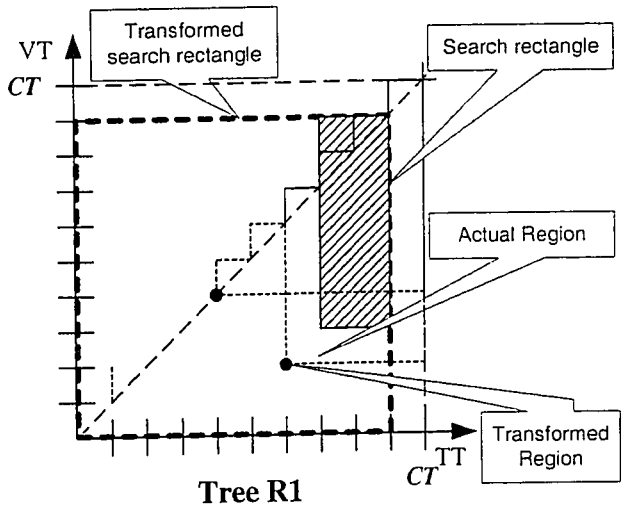


Figure 27 – Search in Tree R1: VTe = NOW and TTe = UC

3.2.2.2 Search in Tree R2

When searching tree R2, the search rectangle is extended to the left to cover all intervals whose actual regions extend to the right along the transaction time.

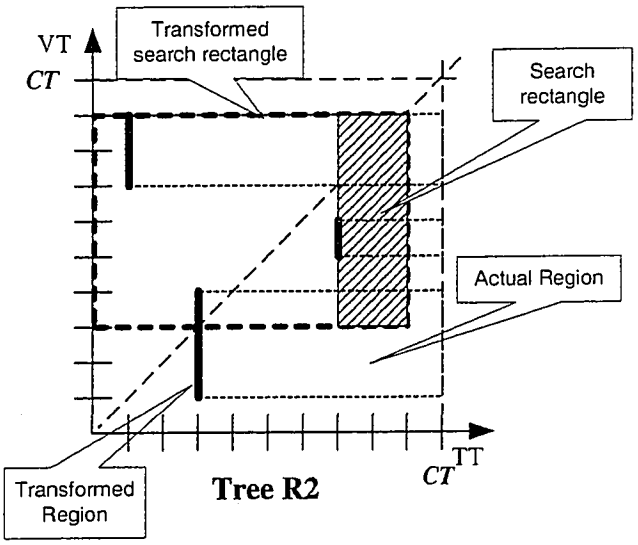


Figure 28 – Search in Tree R2: $VT^e \neq$ NOW and TT^e = UC

3.2.2.3 Search in Tree R3

When searching tree R3, the search rectangle is extended downward to cover all intervals whose actual regions extend to the top along the valid time.

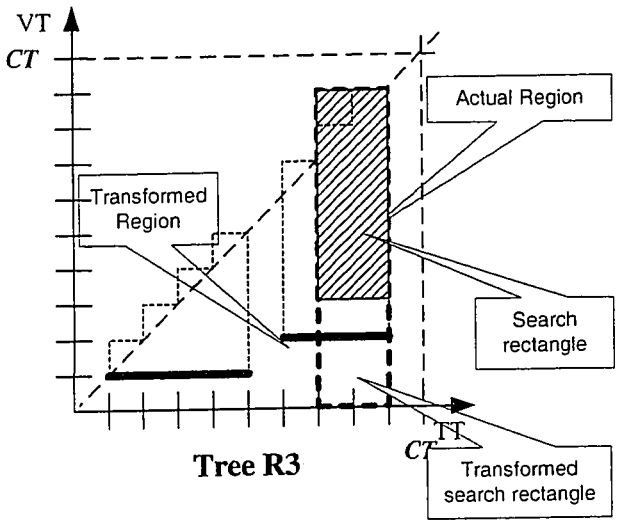


Figure 29 – Search in Tree R3: VT^e = NOW and $TT^e \neq$ UC

However, this transformation is a little bit tricky. When a part of the search rectangle is below the line VT = TT and another part is above it, only the lower part should be extended downwards as shown in Figure 30.

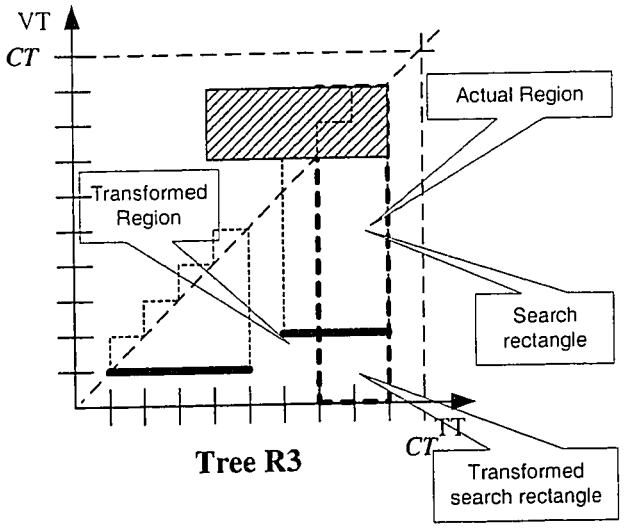


Figure 30 – Special case for search in Tree R3

3.2.2.4 Search in Tree R4

Since Tree R4 indexes untransformed rectangles, there is no need to transform the query rectangle.

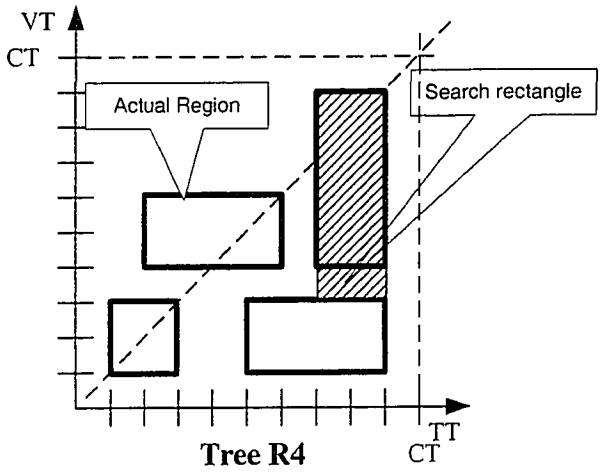


Figure 31 – Search in Tree R4: $VT^e \neq NOW$ and $TT^e \neq UC$

3.3 The Point-based 4R-tree

The performance experiments reported in [BJSS00] have shown some weakness in the 4-R index. Mainly, the unproportionally long segments negatively affected the performance. In tree R2, the MBB have long extents in the valid-time direction, while the transformed queries in R2 have relatively long transaction-time extents. This results in accessing many MBBs in R2. The same problem is also manifested in tree R3.

We address this problem by using a point-based representation instead of the region/interval-based representation used in the original 4-R tree.

3.3.1 Data Transformation

The proposed transformation will transform temporal regions described in section 3.1.3 into variable-free data points. In tree R1, two-dimensional points represent temporal attributes. In both tree R2 and tree R3, temporal attributes are represented by a 3-dimensional points. Finally Tree R4 represents the original temporal region using 4-dimensional points.

3.3.1.1 Point-based Tree R1

Since Tree R1 indexes tuples having both Valid Time and Transaction Time as now-relative, bitemporal values are already represented by a 2-dimensional points (VT^s, TT^s) and there is no need to change this index.

3.3.1.2 Point-based Tree R2

Tree R2 indexes tuples that have now-relative Transaction Time, while Valid Time is a fixed interval with fixed values for both the start and end. Bitemporal values are represented by a 3-dimensional points (VT^s, VT^e, TT^s).

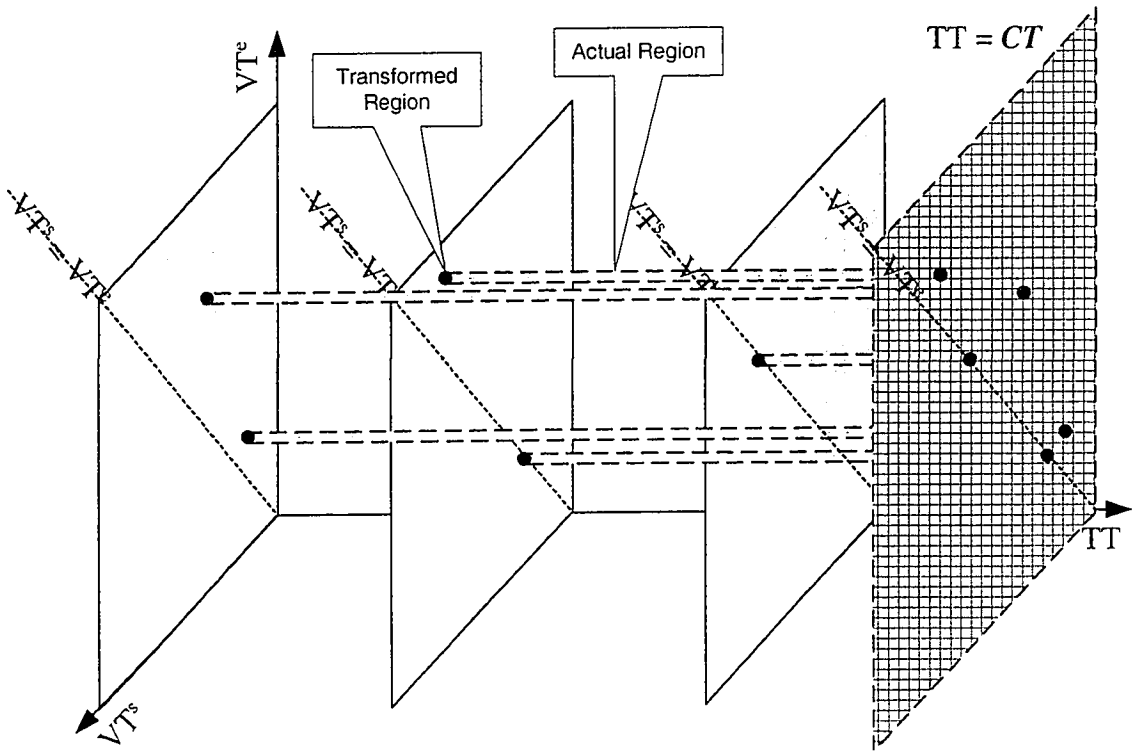


Figure 32 - Point-based Tree R2: $VT^e \neq$ NOW and TT^e = UC

In this tree, all points are on or below the plane $VT^s = VT^e$ ($VT^s \leq VT^e$). Bitemporal data grows in the Transaction Time dimension and is bounded by the plane TT= *CT*.

3.3.1.3 Point-based Tree R3

Tree R3 indexes tuples that have now-relative Valid Time, while Transaction Time is a fixed interval with fixed values for both the start and end. Bitemporal values are represented by a 3-dimensional point (TT^s, TT^e, VT^s).

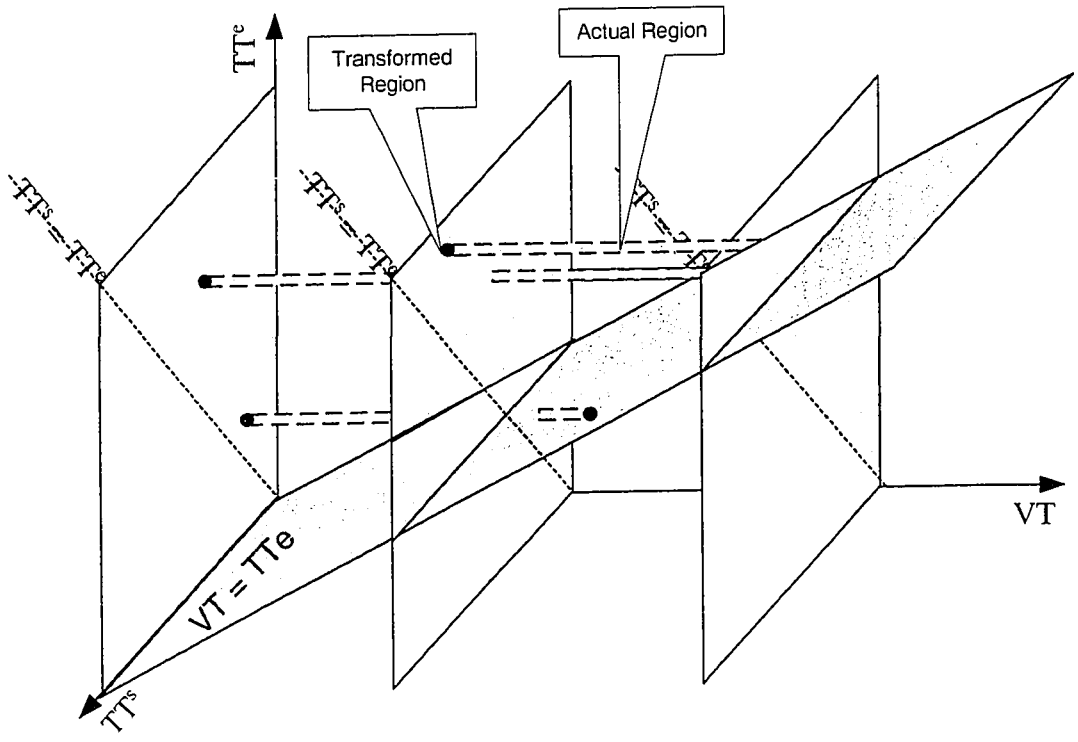


Figure 33 - Point-based Tree R3: VT^e = NOW and $TT^e \neq$ UC

In this tree, bitemporal data grows in the Valid Time dimension and is bounded by the plane $TT^e = VT$. Also, all points are below the plane $TT^s = TT^e$ ($TT^s \leq TT^e$).

3.3.1.4 Point-based Tree R4

Tree R4 indexes tuples that do not have any now-relative data. Bitemporal values are represented by a 4-dimensional points (VT^s, VT^e, TT^s, TT^e).

3.3.2 Query Transformation

In this section, we also investigate the intersect query as described in section 3.2.2. In trees R2 and R3, where the temporal regions are mapped to 3-dimensional points, the

argument rectangle $<VT^l_q, VT^u_q, TT^l_q, TT^u_q>$ is mapped to the hyper-rectangle whose coordinates are:

- $(VT^s = VT^l_q, VT^e = VT^l_q, TT = TT^l_q)$
- $(VT^s = VT^l_q, VT^e = VT^u_q, TT = TT^l_q)$
- $(VT^s = VT^u_q, VT^e = VT^l_q, TT = TT^l_q)$
- $(VT^s = VT^u_q, VT^e = VT^u_q, TT = TT^l_q)$
- $(VT^s = VT^l_q, VT^e = VT^l_q, TT = TT^u_q)$
- $(VT^s = VT^l_q, VT^e = VT^u_q, TT = TT^u_q)$
- $(VT^s = VT^u_q, VT^e = VT^l_q, TT = TT^u_q)$
- $(VT^s = VT^u_q, VT^e = VT^u_q, TT = TT^u_q)$

3.3.2.1 Search in point-based Tree R1

Since point-based Tree R1 is exactly the same as the original tree R1, the search continues to be the same. All points satisfying the condition

$$VT^s \leq VT^u_q \text{ AND } TT^s \leq TT^u_q$$

intersects the query rectangle.

3.3.2.2 Search in point-based Tree R2

Figure 34 focuses on the Valid Time plane and describes those areas that intersect with the original query rectangle. Note that any point that has $VT^s > VT^u_q$ or $VT^e < VT^l_q$ are not considered to intersect with the valid time range $[VT^l_q, VT^u_q]$.

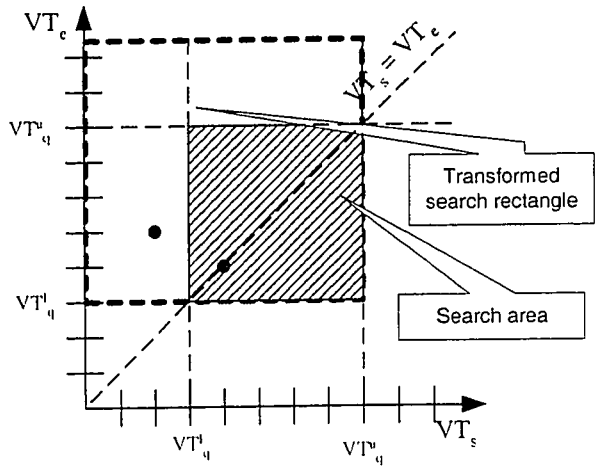


Figure 34 – Valid time points intersecting $[VT^l_q, VT^u_q]$

Now combining the valid time query range with the transaction-time range will result in a hyper rectangle extending from the origin of the transaction time upto TT^u_q. Any point that have $TT^s > TT^u_q$ will not intersect the transaction time range $[TT^l_q, TT^u_q]$.

The search cube is extended to the origin of both the VT^s and TT dimensions as well as the maximum timestamp for the VT^e dimension. This cover all the points whose actual intervals extend to the right in the transaction-time dimension and satisfies the condition

$$VT^s \leq VT^u_q \text{ AND } VT^e \geq VT^l_q \text{ AND } TT^s \leq TT^u_q.$$

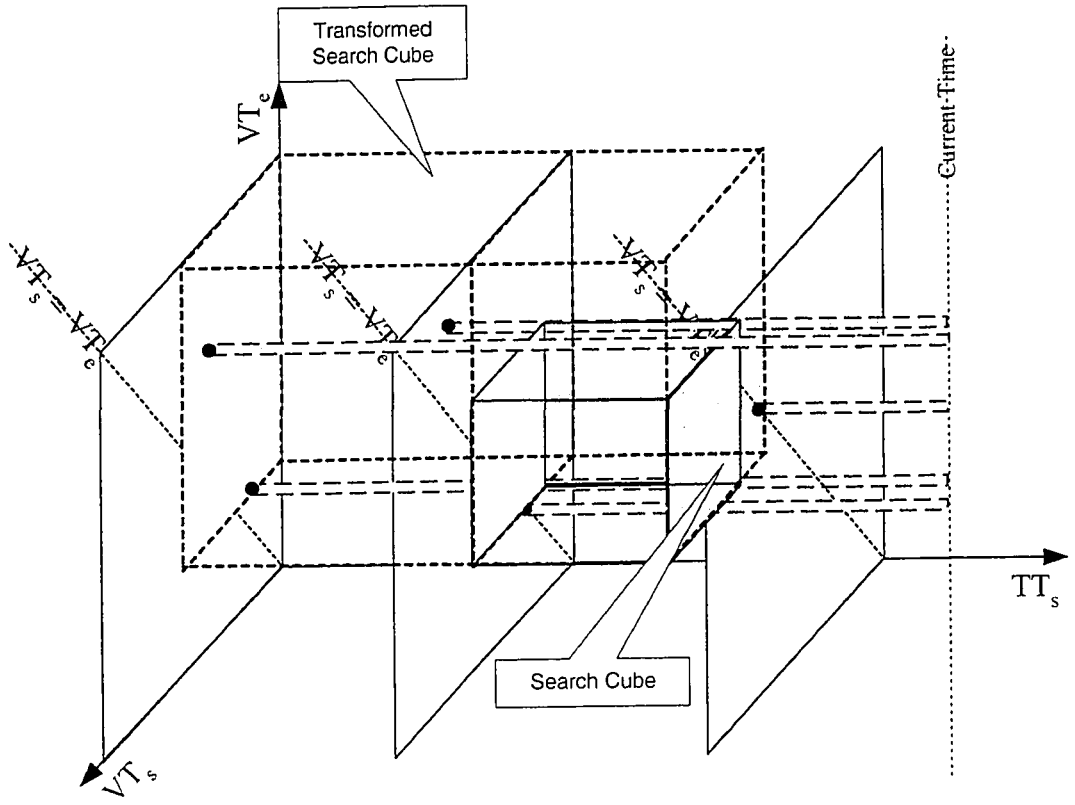


Figure 35 – Search in point-based Tree R2: $VT^e \neq$ NOW and TT^e = UC

3.3.2.3 Search in point-based Tree R3

The transformation is very similar to the transformation done for tree R2 with the following differences:

- The plane of transaction time is considered instead because this tree represents points whose start and end transaction times are known.
- Only points below the plane VT = TT^e are considered.

The search cube is extended to the origin of both the TT^s and VT dimensions as well as the maximum timestamp for the TT^e dimension. This covers all the points whose actual intervals extend to the right in the valid-time dimension and satisfies the condition

$$TT^s \le TT^u_q \text{ AND } TT^e \ge TT^l_q \text{ AND } VT^s \le \max(VT^u_q, TT^u_q).$$

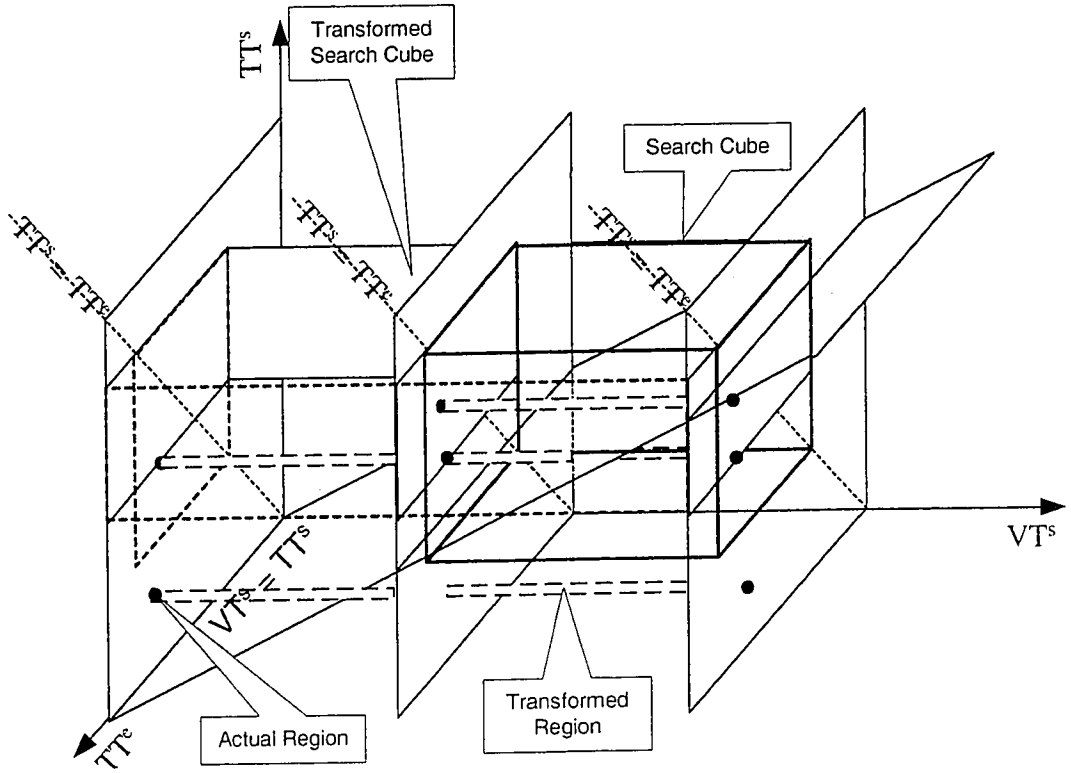


Figure 36 – Search in point-based Tree R3: VT^e = NOW and $TT^e \neq$ UC

3.3.2.4 Search in point-based Tree R4

This tree does not have any transformation. All points satisfying the condition

$$VT^s \le VT^u_q \text{ AND } VT^e \ge VT^l_q \text{ AND } TT^s \le TT^u_q \text{ AND } TT^e \ge TT^l_q$$

intersects the query rectangle.

Chapter 4 Performance Analysis

We need to compare the performance of the proposed index with the other indices that we have described in section "2.5.2 Access Methods for Now-relative Bitemporal Data". Since the implementation of the GR-Tree is quite complex and its implementation is not available, we decided to restrict the comparison with the other indices namely called 1R, 2R and 4R trees. All these indices are using on-the-shelf spatial index implementation, namely the R*-Tree, as the basis of their implementation. In [BJSS00], the comparison included only the rectangle-based implementation of these temporal indices. We have decided to extend the comparison to include both the rectangle and point based implementation. Thus, our experiments will cover six temporal indices:

1. R1_rect index:

   All keys are stored in one R*-tree where temporal keys are represented as rectangles in the transaction time and valid time dimensions. Now relative temporal info is represented using the maximum timestamp technique.

2. R1_point index:

   Keys are stored in one R*-tree where temporal indexes are represented as 4-dimensional point in the transaction time and valid time dimensions. Now relative temporal info is still represented using the maximum timestamp technique.

3. R2_rect index:

   All keys are stored in two R*-trees: the Front R-Tree and the Back R-tree. The Front R-Tree (which we call R2_rect_Uc) contains those keys that have a TT^e = *UC*. While the Back R-Tree (which we call R2_rect_NoVar) contains those keys that have a fixed TT^e. Note that the original 2R-Tree [KTF98], was assuming that

VT^e is always a fixed timestamp, which is a wrong assumption. In each subtree, keys are represented as rectangles in the transaction time and valid time dimensions. Now relative valid time info is still represented using the maximum timestamp technique.

4. R2_point index:

   It is very similar to the R2_rect index except that keys are represented as points instead of rectangles. In the R2_point_Uc, keys are represented as 3 dimensional points for the TT^s, VT^s and VT^e. While keys in the R2_point_NoVar are represented as 4 dimensional points for the TT^s, TT^e, VT^s and VT^e.

5. R4_rect index:

   Keys are stored in four R*-trees as described in section "3.2 The Rectangle-based 4R-tree". We named the 4 trees according to the eliminated variable:

   - R4_rect_UcNow contains those keys that have $TT^e = \textit{UC}$ and $VT^e = \textit{Now}$.
   - R4_rect_Uc contains those keys that have $TT^e = \textit{UC}$.
   - R4_rect_Now contains those keys that have $VT^e = \textit{Now}$.
   - R4_rect_NoVar contains those keys that have both TT^e and VT^e as fixed timestamps.

6. R4_point index (our proposed index):

   It is very similar to the R4_rect index except that keys are represented as points instead of rectangles.

4.1 Experimental Design

To fairly compare the performance of the indices, we need to apply the same workload and queries to the six indices. The workload needs to be as realistic as possible and

represent history for objects stored in the index. Also queries need to cover the majority of queries which are current time slice queries, as well a percentage of point and range queries. The workload also needs to simulate the life of the index over a fairly long period of time and not only a bulk load and a set of queries. Details of the workload generation and characteristics are detailed later in section "4.1.1 Workload Generation". Performance of the update (insert and delete) and search operations is measured in actual I/Os. For each operation, we reset the counters of the actual read/write of pages to/from the disk drives, and the actual count of I/Os is the total of both counters at the end of the operation.

4.1.1 Workload Generation

We have not been able to find any study that presents empirical data distribution for temporal data application. Both [BJSS00] and [SJ99] used a workload generator that simulates the evolving temporal aspects of a set of objects. To simulate the index usage over a period of time, the generated workload is an intermix of:

- index operations (insertions and logical deletions), and
- queries.

In the following subsections, we describe the generation of each along with the parameters that controls their generation.

4.1.1.1 Generating index operations

This workload generator uses a set of parameters that account for the spatial and valid-time aspects of objects and are adapted from the GSTD algorithm [TSN99]. In [SJ99], the spatial and valid-time values were augmented with transaction times that represent

the timestamp when the records are stored in the database. The enhanced workload generator generates both retroactive and predictive insertions. A retroactive insertion occurs when $VT^s < TT^s$ for the inserted record. While a predictive insertion occurs when $VT^s > TT^s$ for the inserted record.

The generator adopts the concept of an object history which captures the evolution of a single object. Each object's history has a number of triples of a data value (non-temporal attribute), a valid time interval and a transaction time interval. The valid time intervals within a history do not overlap. This guarantees at most one non-temporal attribute value at each point in time. For each history, there may be at most one triple which is current (i.e. $TT^e = \textit{UC}$).

The history of an object is active if it has an entry with a valid time that includes the current time. For active histories, the interval having the largest VT^s is made now-relative. Histories that are not active contain no entries with $VT^e = \textit{Now}$. ***VL*** controls the maximum valid-time interval length. Valid-time intervals are uniformly distributed between 0 and ***VL***. For now-relative intervals, the $\triangle$–offset is normally distributed with mean 0 and deviation ***DevDelta***. While its VT^s is normally distributed with mean equal to TT^s and deviation ***Dev***. Parameter ***VL*** captures how often objects change, and parameter ***Dev*** controls the correlation between transaction and valid time.

The transaction-time intervals in the triples are obtained by simulating database modifications. To accomplish this, the workload generator maintains a list of the currently active histories. Each history has associated the time when the next insertion for the history will occur, along with the valid-time interval to be inserted. The list is always ordered on TT^s.

Figure 37 describes how the workload generator updates a history. An update to a history is simulated by updating VT^e of the most recent entry in the history from being ***Now*** to the static value of VT^s of the new entry. This guarantees that the valid-time intervals of the entries within the history always meets and do not overlap. After the insertion, a new, planned entry is generated together with its insertion time. The new entry's VT^s is normally distributed with deviation ***Dev*** and mean equal to its planned TT^s. This planned TT^s is generated such that, after inserting the new entry, the length of the previous valid time interval will not exceed ***VL***.

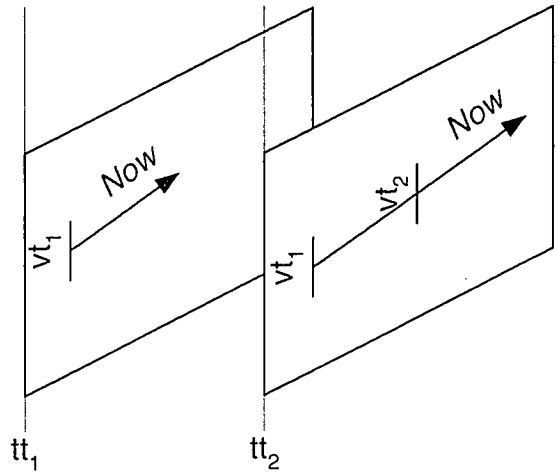


Figure 37 – updating a history within the workload

At each time point, the workload generator generates one of the following index operations:

- Inserting a new object history
- Terminating an object history
- Updating an object history.

At each time point, the ratio of Insertions, terminations and updates is controlled by ***ProbStart***, ***ProbEnd***, (1 - ***ProbStart*** – ***ProbEnd***). A history is updated by either:

- inserting a new valid-time interval somewhere between the beginning and the current end of the history, or
- deleting some existing valid-time interval from the history.

Two more parameters ***ProbUpdInsert*** and ***ProbUpdDelete*** controls the ratio of insert operations with respect to delete operations during the update of object histories. The workload generator ensures that the valid-time intervals in a history continue to meet after an update. Following the insertion of a new interval, all intervals fully covered by it are logically deleted and partially covered intervals are updated as shown in Figure 38. The "new inserted key" fully overlaps with k_2 valid-time interval, then it is logically deleted. While the "new inserted key" partially overlaps with k_1 and k_3 valid-time intervals, then they are both updated. First the original keys are logically deleted, then corresponding updated keys are inserted into the history. The updated k_1 valid-time interval is { VT^s of original k_1, VT^s of "new inserted key" }, and the updated k_3 valid-time interval is { VT^e of "new inserted key", VT^e of original k_3 }.

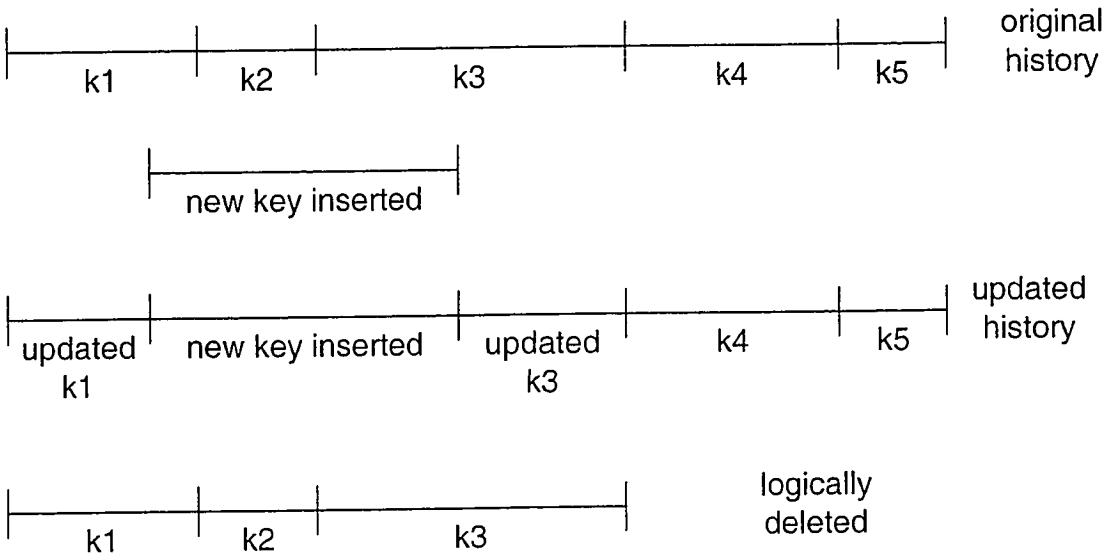


Figure 38 – Updating an object history by adding a new interval

Following the deletion of an interval, the valid-time end of the previous interval, if it exists, is set to the valid-time end of the deleted interval as shown in Figure 39.

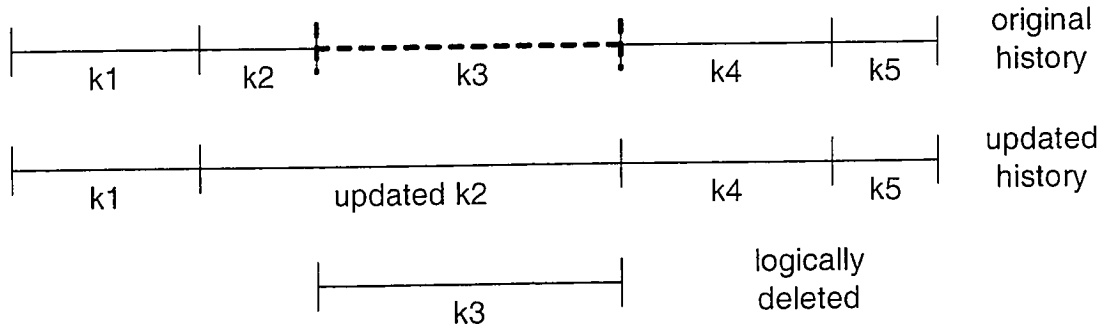


Figure 39 - Updating an object history by deleting an interval

The workload generator provides two more parameters to specify the beginning of time for the experiment ***Beginning*** as well as the number of initial objects created at the start

of the experiment ***InitialInserts***. For the first ***InitialInserts*** time units, at each time point, the workload generator will introduce a new object.

We have enhanced the generator by adding one more parameter that controls the percentage of data with $VT^e = Now$.

4.1.1.2 Generating queries

The workload generates ***QueryQty*** queries every ***Interval*** index operations. The queries are overlapping queries represented as rectangles in both the transaction- and valid-time dimensions. The valid-time begin VT^l of a query is uniformly distributed between 0 and the largest VT^s of an entry in the index. The parameter ***Current*** controls the percentage of the queries whose the transaction-time end is set to the current time (i.e. $TT^u = CT$). Then for the remaining '1- ***Current***' queries, they are uniformly distributed between ***Beginning*** and ***CT***.

The workload generates three types of queries:

- Range Queries (or windows), where TT^l not equal TT^u and VT^l not equal VT^u.
- Transaction Timeslice Queries, where $TT^l = TT^u$ and VT^l not equal VT^u.
- Point Queries, where $TT^l = TT^u$ and $VT^l = VT^u$.

The percentage of queries generated from each of these types is controlled through the parameters ***Windows***, ***Slices***, and ***Points*** respectively. The maximum query window transaction time extent and the maximum query window valid time extent are controlled by the parameters ***MaxQWindowTT*** and ***MaxQWindowVT***.

4.1.1.3 Workload parameters summary

Table 8 summarizes the workload generator parameters described in the previous sections.

Parameter	Short description	Values	"Average" value
Beginning	Beginning Time of the experiment	5000	5000
InitialInserts	Number of time points from the beginning where the workload creates a new object at each of them.	6000	6000
VL	Maximum valid time interval length	50, 100, 500, 1000, 3000, 5000	500
Dev	Deviation of VT^s	1000, 5000, 10000, 25000, 50000	5000
ProbStart	Probability of operations that start new histories	0.0 → 1.0	0.5
ProbEnd	Probability of operations that ends current active histories	0.0 → 1.0	0.1
ProbUpdInsert	Probability of insert operations on an active history	0.0 → 1.0	0.7
ProbUpdDelete	Probability of delete operations on an active history	0.0 → 1.0	0.3
pNow	Probability of data with $VT^e = Now$	0.0 → 1.0	0.6

Parameter	Short description	Values	"Average" value
Query Parameters			
Interval	Interval (in # of logical operations) when to generate queries	5000	5000
QueryQty	number of queries to generate at each time specified	50	50
Current	Probability of current-time query in query mix (orthogonal to other query types)	0.0, 0.25, 0.50, .75, 1.00	.65
Windows	Probability of range queries	0.0, .25, 1.0	0.25
Slices	Probability of time slice queries	0.0, .50, 1.0	0.50
Points	Probability of point queries	0.0, .25, 1.0	0.25
MaxQWindowTT	Maximum query window transaction time extent	1, 100, 300, 500, 1000, 3000	300
MaxQWindowVT	Maximum query window valid time extent	1, 100, 300, 500, 1000, 3000	300

Table 8 - Workload generator parameters

4.2 Implementation

4.2.1 GiST: Generalized Search Tree

The Generalized Search Tree (GiST) [HNP95], is a template indexing structure that allows domain experts (e.g in computer vision, bioinformatics, or remote sensing) to easily customize a database system to index their content. Its implementation **Libgist v2.0** was developed at University of California (Berkeley) and comes prepackaged with extensions for B-trees, R-trees, R*-trees, SS-trees and SR-trees. The GiST is also used in the implementation of the Postgres open source RDBMS.

We have enhanced the implementation of GiST to allow specification of the page and the buffer sizes used for any of the trees. As a default, we chose to use a page of size 1 KB such that one tree node is stored in one page. We have also selected a buffer size of 100 pages. For those indices that use more than one subtree, the size of the buffer of the subtree is equal to the total buffers divided by the number of the subtrees. For example, the R_4 index uses a buffer of 25 pages each such that the total is still 100 pages. Leutenegger and Lopez [LL98] have shown that omitting a buffer may lead to quantitatively and qualitatively incorrect conclusions. Nodes loaded into the buffer are replaced using the least recently used page replacement policy. As a result of insertions and/or deletions, node contents change and the corresponding page in the buffer is marked as ***dirty***. Dirty pages are written back to disk when they are removed by the replacement policy or when an explicit flush operation is requested.

4.2.1.1 Enhancements to Libgist v2.0

In order to perform the designed experiments, we had to make several enhancements to **Libgist v2.0** implementation:

1. Support of temporal intervals

   Add support to both the transaction and valid time temporal intervals. This includes support of special values *UC* and *Now*.

2. Implementation of 2-R and 4-R temporal indices.

   We have defined the six indices used in our study along with the support functions that allow to parse and print temporal intervals using both rectangle and point representations. We also created two subclasses of the ***gist*** class namely:

   - gist_2t, and
   - gist_4t

   That corresponds to the 2-R and 4-R temporal trees. These subclasses hide the fact that the index is using more than subtree and provide a unified interface similar to the original ***gist*** class.

3. Allow specification of the buffer size at run-time.

   Libgist v2.0 has hard coded value for the buffer size equal to 17 buffers. We allowed the specification of the buffer size to be used for a specific index file by specifying the value during creation or opening an index file. This will allow us to study the effect of the buffer size on the different indices covered by this study. Also, for indices that use more than one subtree, we will need to assign each subtree a buffer size such that total buffers of all subtrees is still equal to the buffer size used for indices using only one tree.

4. Allow specification of the page size when creating new index files.

   Libgist v2.0 has hard coded value for the page size equal to 1024 bytes. We allowed the specification of the page size to be used for a specific index file by specifying the value during creation of an index file. The page size is then stored within the header page of the file and reused each time the file is re-opened. This will allow us to study the effect of the page size on the different indices covered by this study.

5. Buffer replacement strategy.

   Libgist v2.0 uses a rather naïve buffer replacement strategy. The strategy tries first to use ***virgin*** buffers that have never used before. Once all buffers have been used, the strategy was to simply get the first buffer that is not pinned. This results in re-using over and over the first unpinned couple of pages. We have introduced a timestamp that tracks the last time the buffer was pinned. This allowed us to select the buffer least recently used for replacement.

6. Capture and report index runtime statistics.

   Added capabilities to capture actual I/Os statistics for each index operation and accumulate them at the index level. We also analyze the tree structure and capture tree properties as:

   - Internal node utilization,
   - Leaf node utilization,
   - Internal nodes count,
   - Leaf nodes count,
   - Tree height

- Tree size (in KB),
- Buffers count,
- Page size,
- Number of records in internal nodes,
- Number of records in leaf nodes,

And for each level, we calculate:

a. Dead space,

b. Overlap, and

c. Nodes count;

4.2.1.2 Libgist v2.0 Class Diagram

Figure 40 shows the main classes of the Libgist 2.0. The classes that we have added are highlighted. The R2 indices are instantiated from the class gis2t, while the R4 indices are instantiated from the gis4t class. Subtrees used within the gis2t and gis4t classes are instantiated from the class rstar_ext_t namely:

- temporal_rstar_point_ext
- temporal_rstar_rect_ext
- temporal_uc_rstar_point_ext
- temporal_uc_rstar_rect_ext
- temporal_now_rstar_point_ext
- temporal_now_rstar_rect_ext
- temporal_uc_now_rstar_point_ext
- temporal_uc_now_rstar_rect_ext

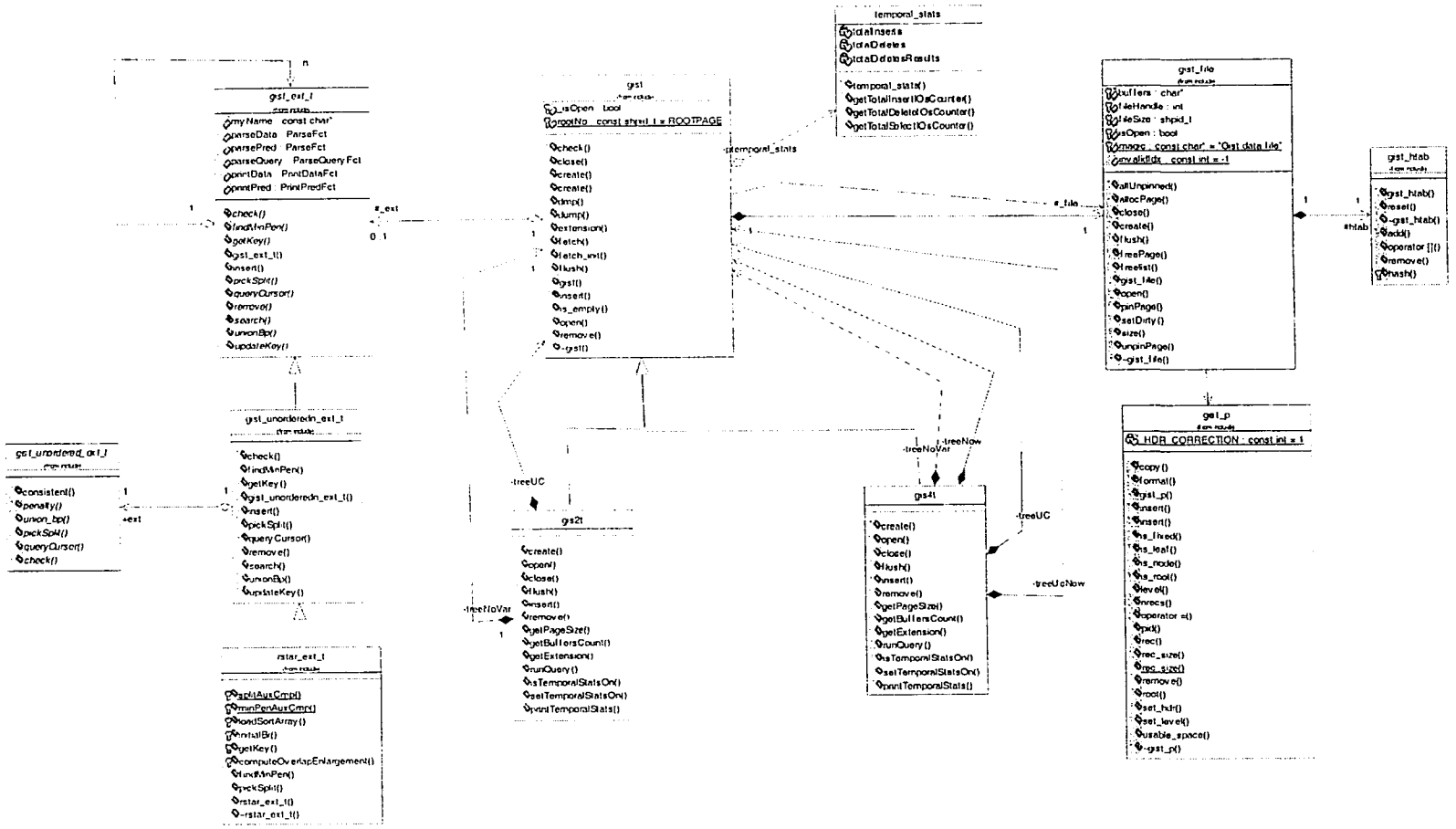


Figure 40 – Enhanced Libgist 2.0 Class Diagram

4.2.2 AMDB: Access Method Debugger

We have also used **amdb v.1.0** [SKH99] implemented at the University Of California (Berkeley). **amdb v.1.0** is a graphical tool to visualize the entire search tree, paths and subtrees within the tree, as well as data contained in each node of the tree. The tool also allows graphical animation of index search, insert and delete operations. We have used this tool to visualize the space partitioning for the different indexes under study. Appendix A contains many snapshots obtained from **amdb v.1.0**.

4.2.3 Index operations algorithm

4.2.3.1 Insert

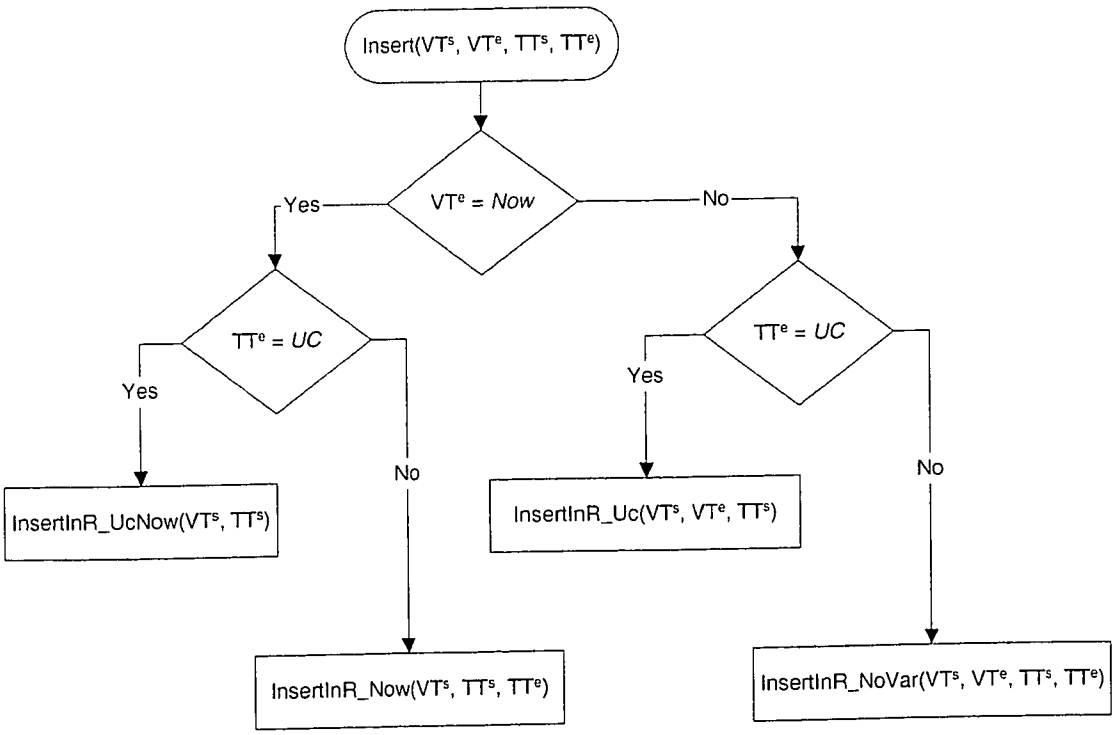


Figure 41 – Algorithm for inserting in the 4-R Index

4.2.3.2 Delete

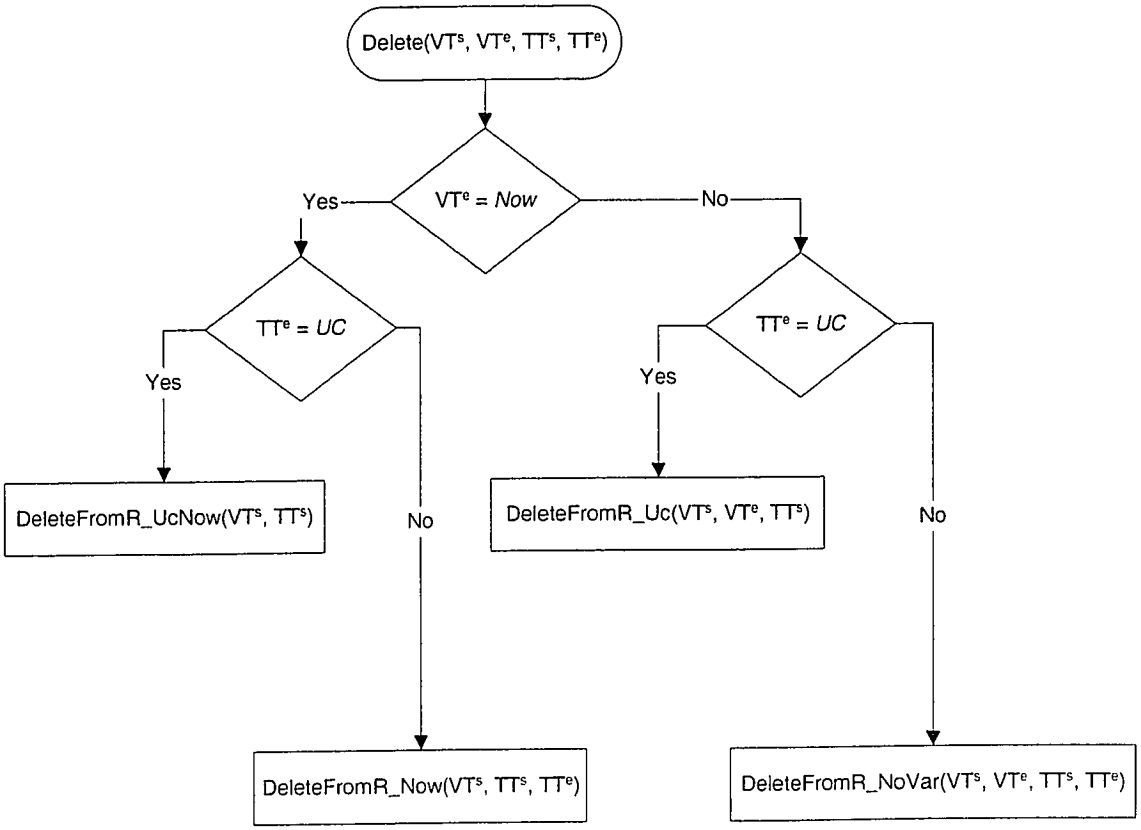


Figure 42 – Algorithm for deleting from the 4-R Index

4.2.3.3 Query

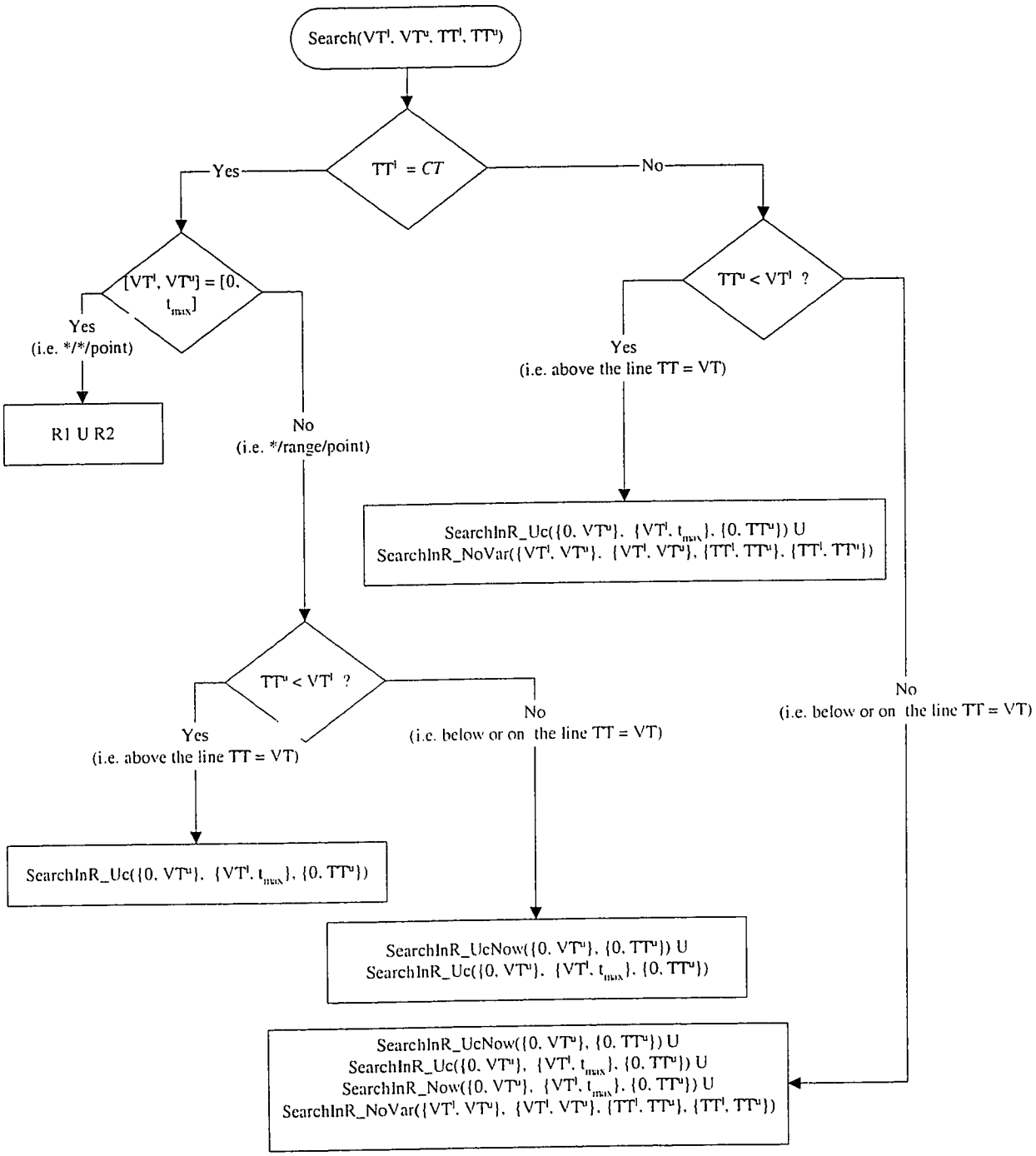


Figure 43 – Algorithm for searching in the 4-R Index

4.2.4 Output Matrix

4.3 Experiments & Results

In this section, we study the effect of various parameters on the performance of the six indices. In each subsection, we present the search and update performance results pertaining to the parameter under study. Additional results are presented in Appendix A.

For studying the effect of the size, we experimented with different workload sizes ranging from 10,000 and 100,000 logical operations. For all other experiments, each one was conducted using a workload of 100,000 logical operations. This is the largest size that we could experiment with within the available computing and time resources constraints. Each experiment involves building the six indices which needs about 700 MB and takes between 18 and 24 hours.

4.3.1 General observations

The results confirmed that the proposed temporal index R4_point performs consistently better than the other five indices in terms of query performance. However, it does not have the best update performance. It was noticed that the R1_rect has the worst update performance that is linearly increasing as the index size increases. Within the five other indices, the rectangle version of the index shows better update performance than the corresponding point version. Only the R4_point competes with the 2 rectangle versions and is very close to the R4_rect performance.

The internal nodes for the point indices stores fewer items because each MBB is larger. For example, the R2_point_NoVar has MBBs of size 64 (2 points representing interval in each temporal dimension), while the corresponding R2_rect_NoVar has MBBs of size

32. This means that internal nodes in R2_point_NoVar contains half the entries that may be contained in the R2_rect_NoVar.

4.3.2 Impact of Index Size

In this experiment, we study the performance of the six indices as the size of the index (# of logical operations) increases. Figure 44 shows that the number of I/Os increases as the size of the index increases. This is due to the corresponding increasing size of the search results.

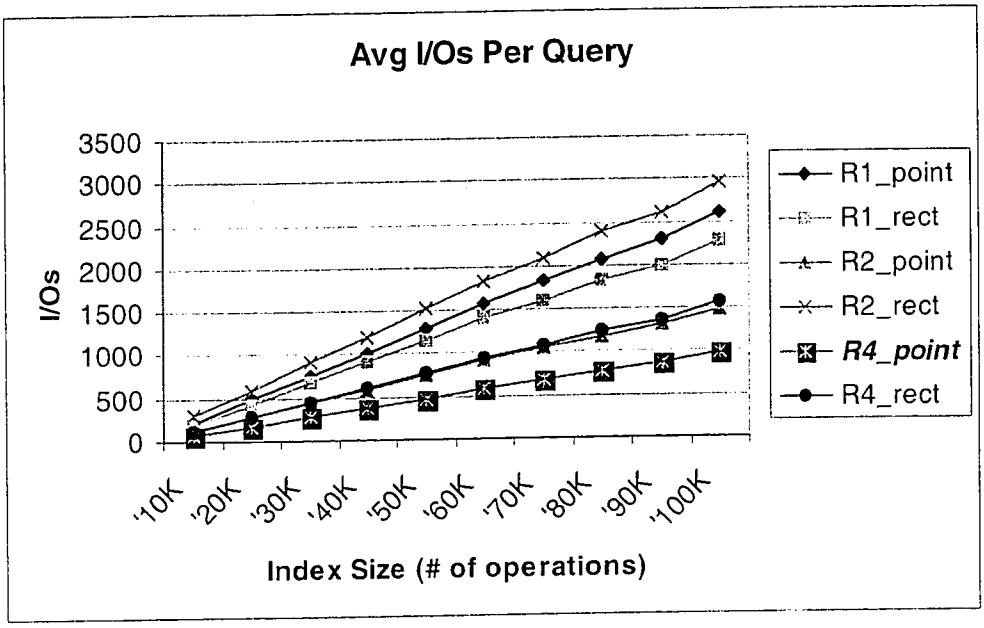


Figure 44 - Search performance with varying index size

In order to eliminate the effect of the query results' size, we have normalized the I/Os against the size of the search results. This is done by dividing the total query I/Os by the number of records returned in the corresponding search results. Figure 45 shows the average number of I/Os needed to return one record in the search result. It clearly shows that the R4_point index has the best performance. The R2_point also has a good

performance and even its performance surpasses the R4_rect performance especially as the size of the index increases.

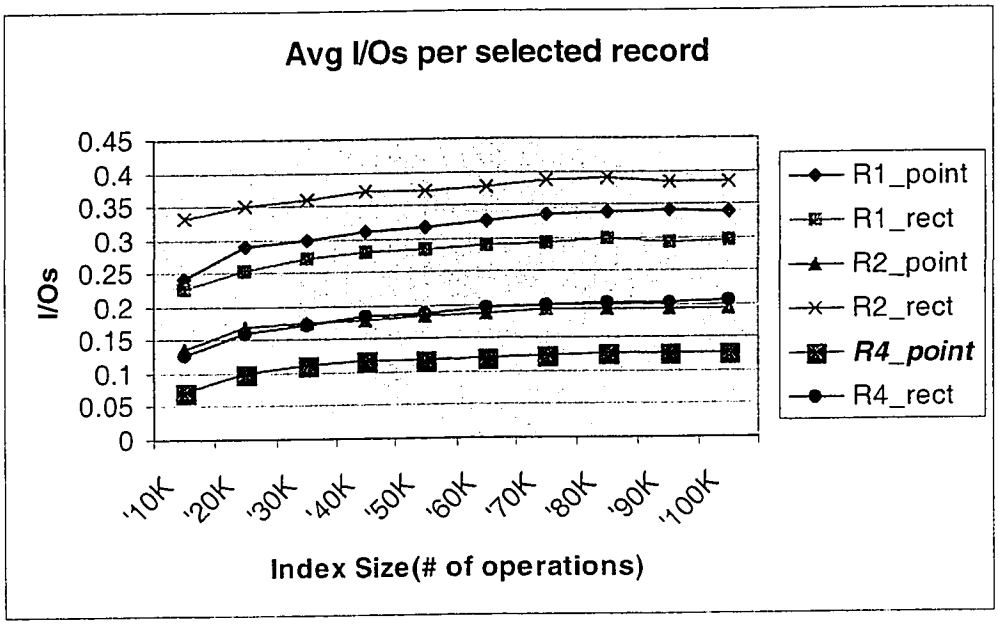


Figure 45 - Search performance normalized to the search result size

Although R2_rect has the worst search performance, it has the best update performance. Studying this phenomenon, we discovered that, due to the sequential nature of the transaction time, insertions always fall in the same bucket that gets split. This means that insertions that have growing TT^s always follow the same path from the root to the leaf. The pages of this path are always kept in the buffer and no I/Os are needed until the node gets full and the index needs to split it. In this case, the index needs only to write the dirty page to the disk and this I/O is probably counted during the search when the index t needs to bring into the buffer more pages. However, as shown in Figure 46, R2_rect has low utilization because the pages that get split are left half full and the index never inserts more keys into old pages.

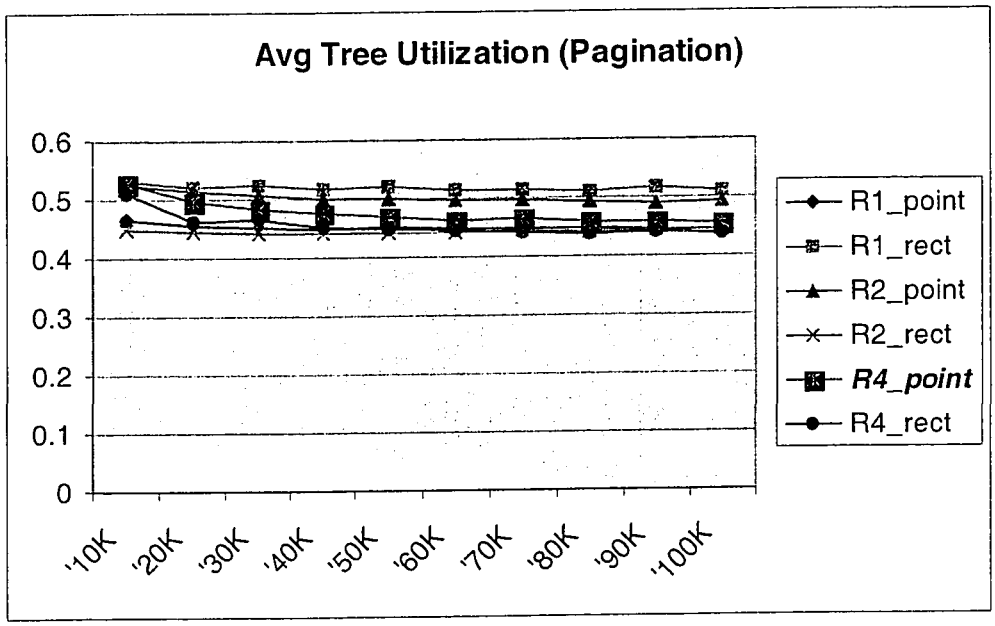


Figure 46 - Tree utilization for varying index size

The R1_rect has the worst update performance especially as the size of the index increases. This is mainly because of the fact that all internal nodes overlap and cover the whole temporal space. This makes the delete performance very bad, because almost all subtrees need to be searched to find the record to be deleted.

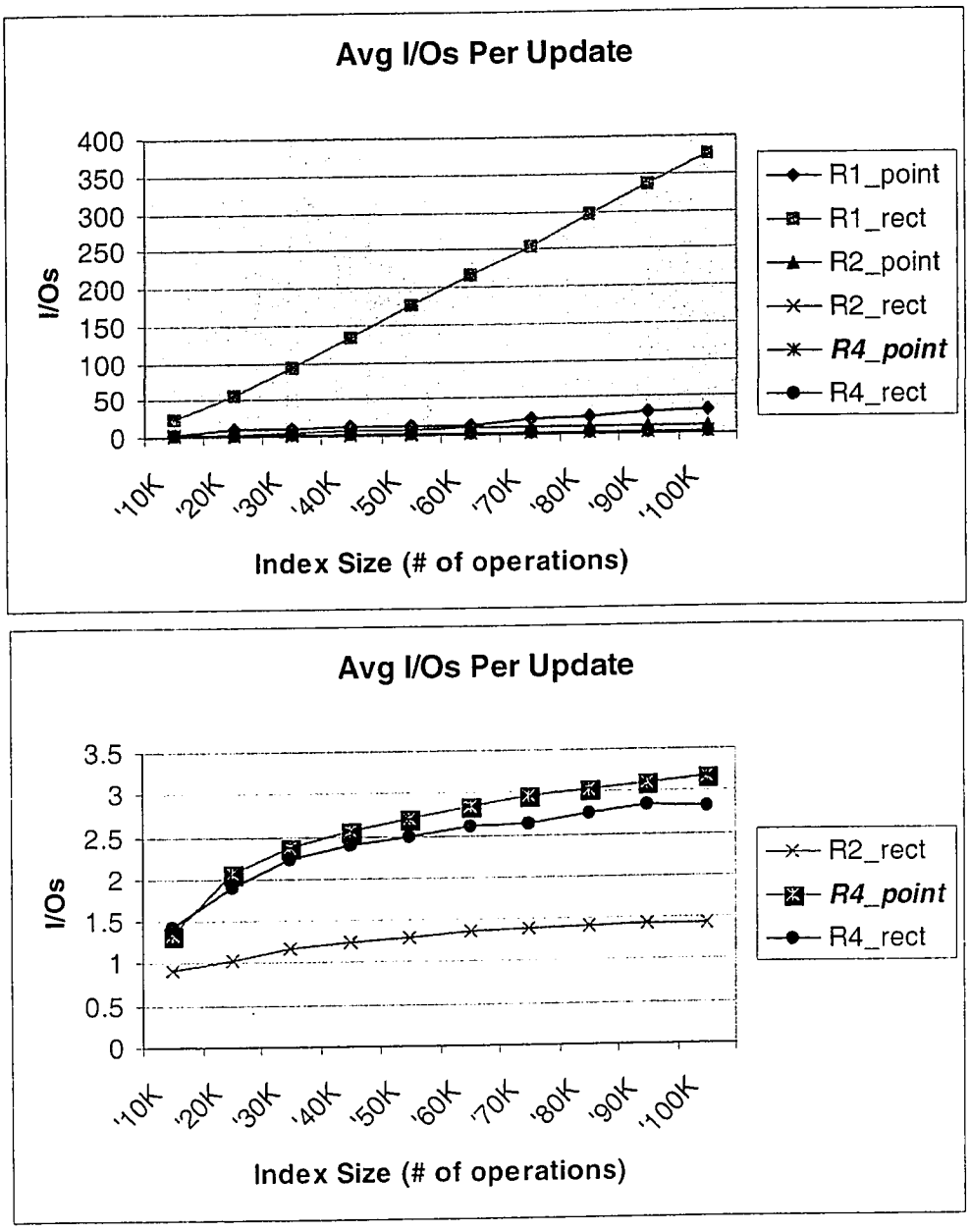


Figure 47 - Update performance with varying index size

We used the amdb tool to visually debug the index structure. Figure 48 shows the global view of the tree which consists of a root, 2 levels of internal nodes and the leaf nodes. Lighter nodes represent a better utilization of the node (more keys stored and less empty space).

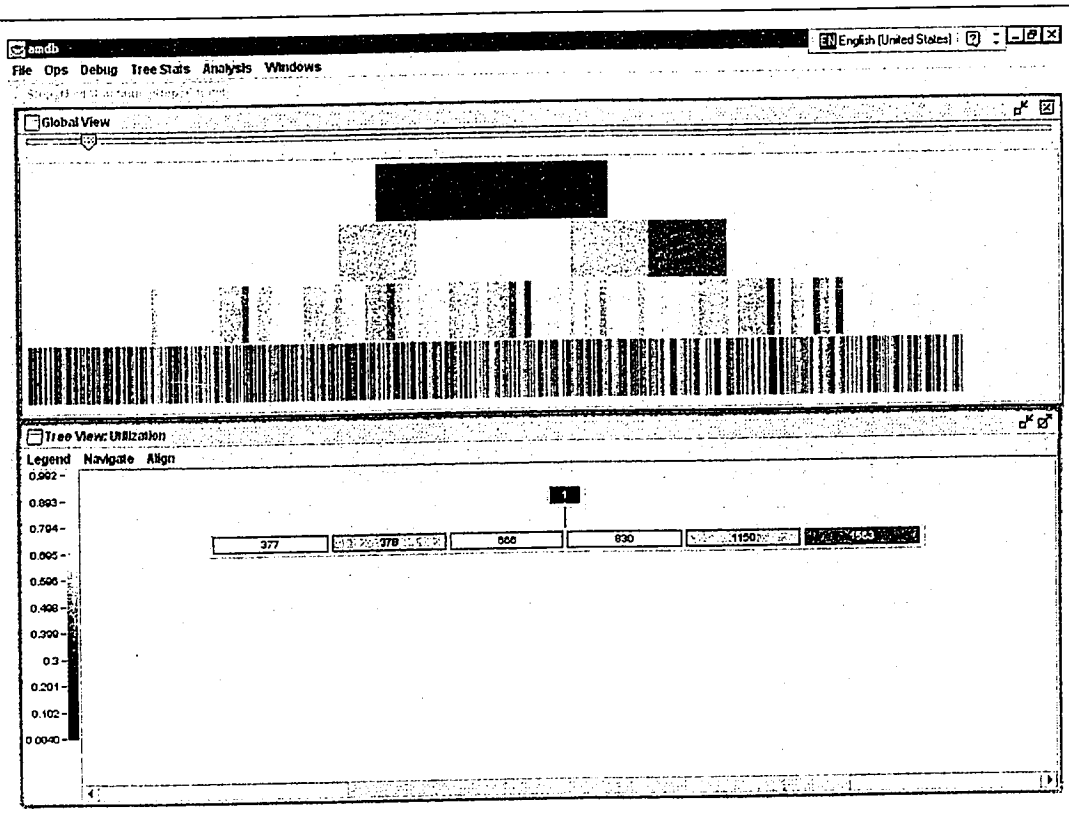


Figure 48 - R1_rect tree structure and utilization

Figure 49 shows the MBBs corresponding to the root's six child nodes shown in Figure 48. Figure 49 is not empty but all MBBs of the R1_rect are rectangles expanding from the root of the experiment space (point {5000, 0}) to the maximum time stamp for both TT and VT.

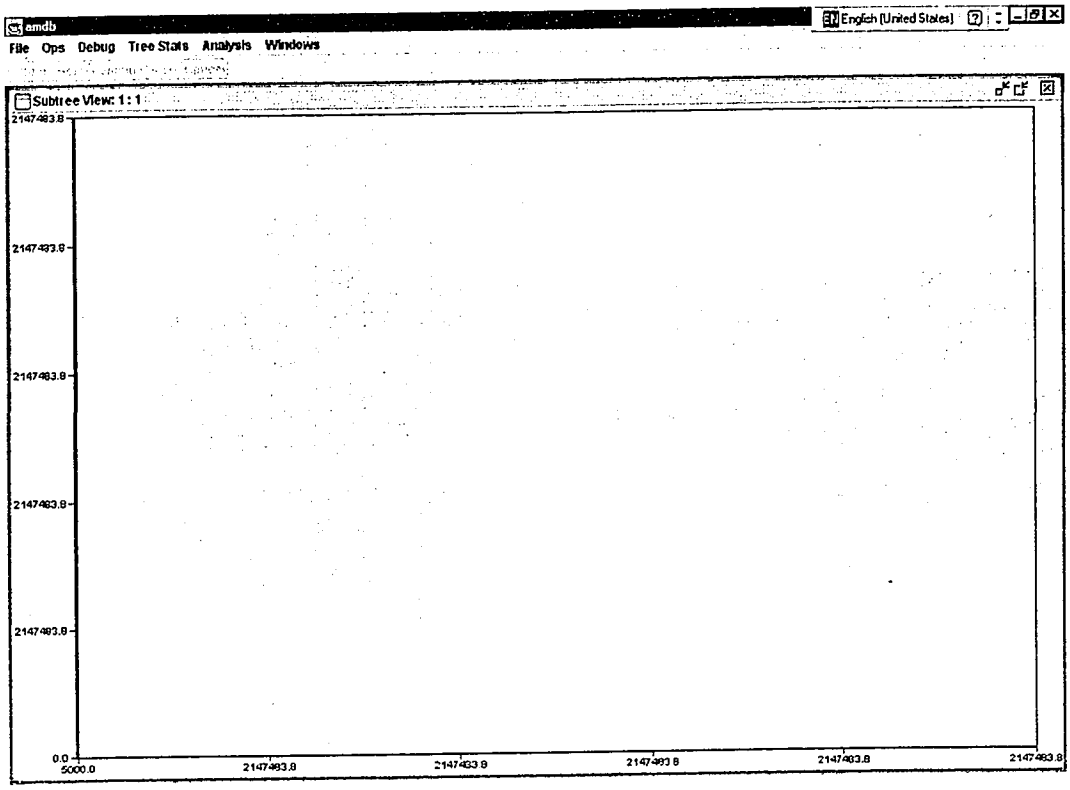


Figure 49 - R1_rect MBBs are fully overlapping

Our R4_point has the second worst update performance. However it exhibits the minimum number of splits and nodes count (see section A.1). Studying this behavior and comparing it to the R4_rect index update performance, we found that the bad performance is mainly a result of the subtree NoVar and the subtree UC. Almost each of these subtrees uses 4 times more I/Os. However, the performance of the subtree UcNow in the R4_point index is always better than the corresponding one in the R4_rect.

The point representation of the temporal dimension allows the R*-Tree algorithm to generate better splitting because it considers 4 axis in case of the NoVar subtree and 3 axis in the case of the Uc subtree. While in the rectangle representation, only 2 axis are considered in the splitting, where values on the TT axis are sequentially increasing so rectangles that lies completely to the left of current time will never grow and will never

split. This means that the page located in the buffer will continue to receive more inserts until it needs to get split. This can be visually depicted in Figure 50, Figure 51, Figure 52, and Figure 53.

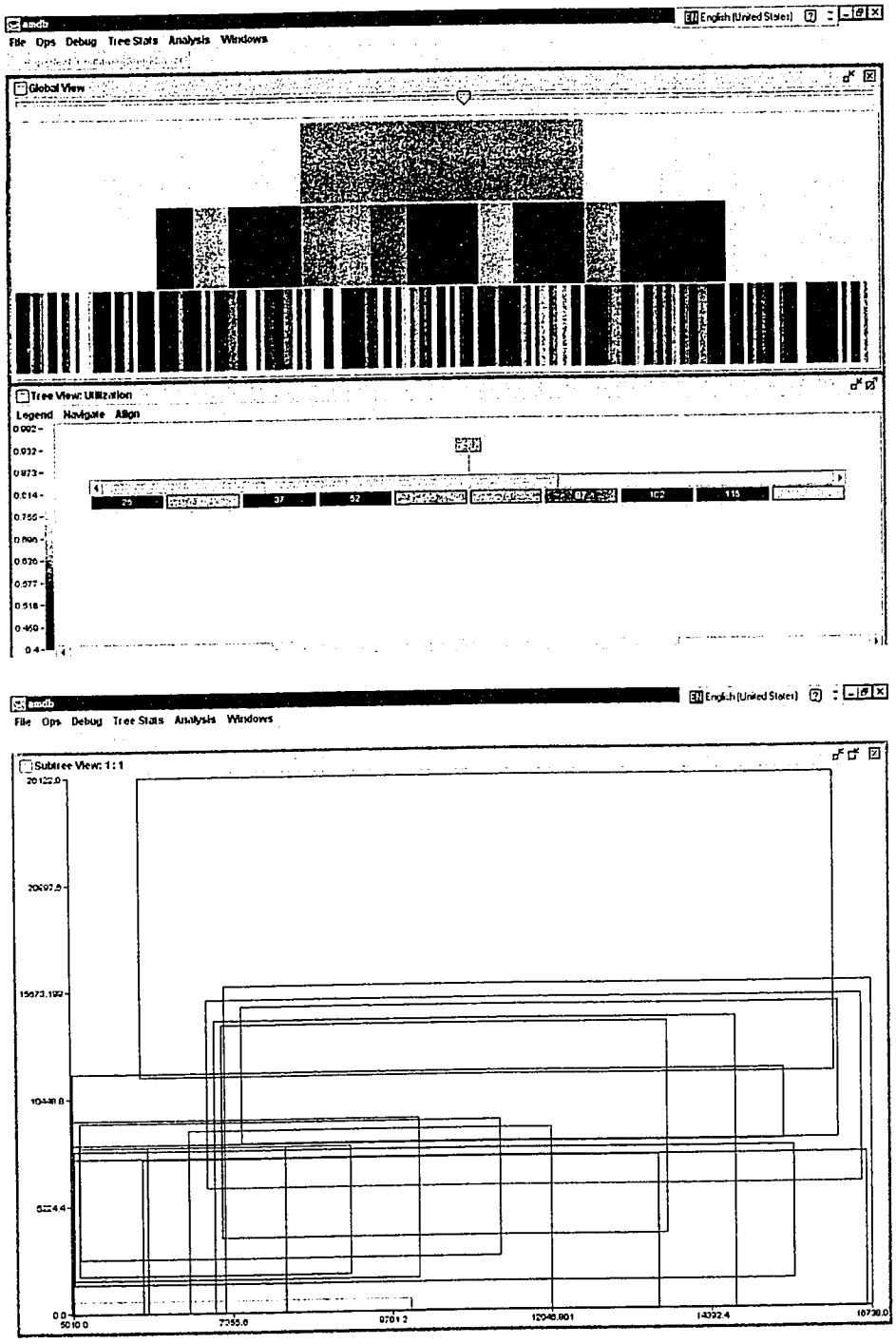


Figure 50 - R4_rect_NoVar

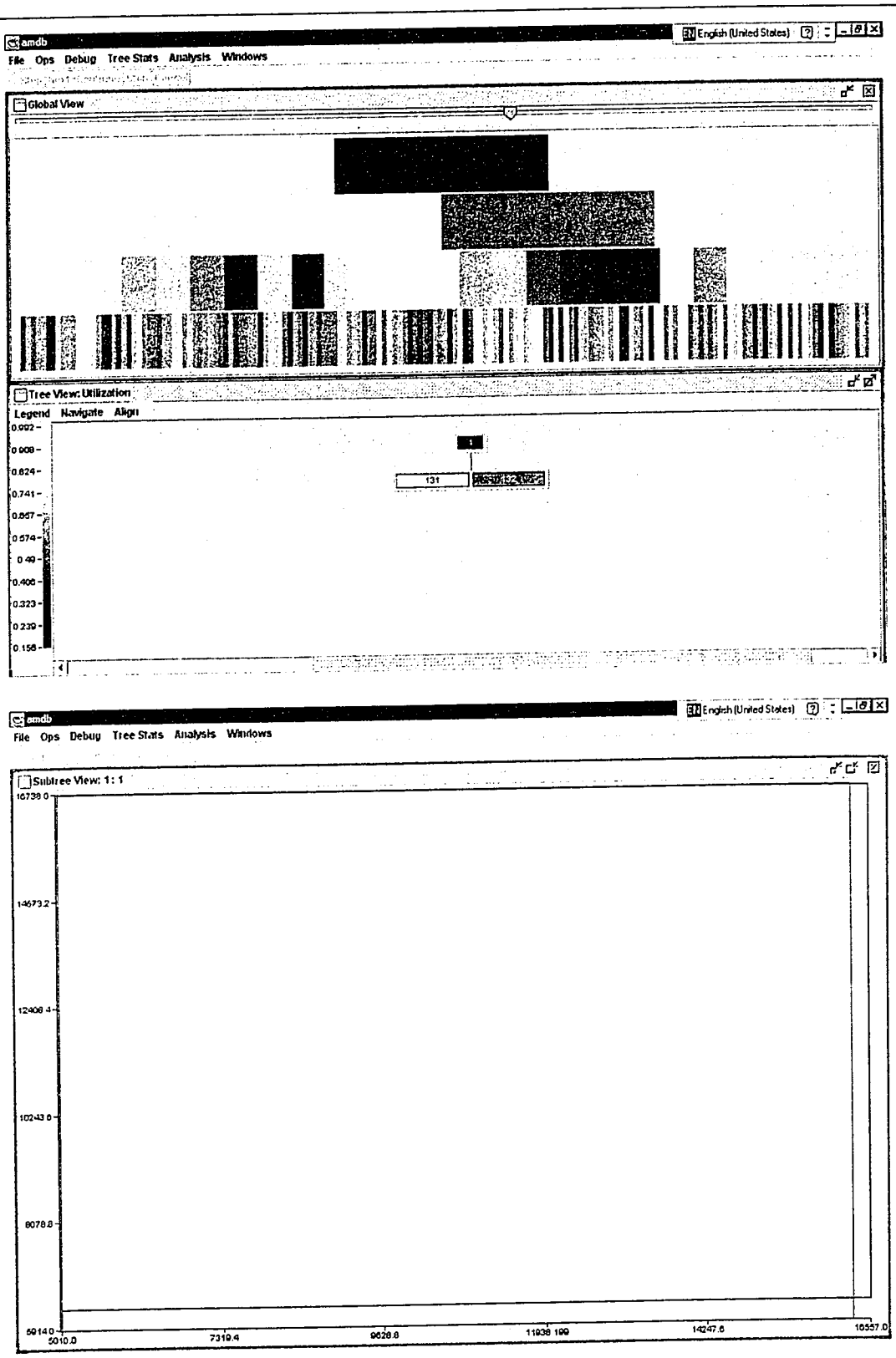


Figure 51 - R4_point_NoVar

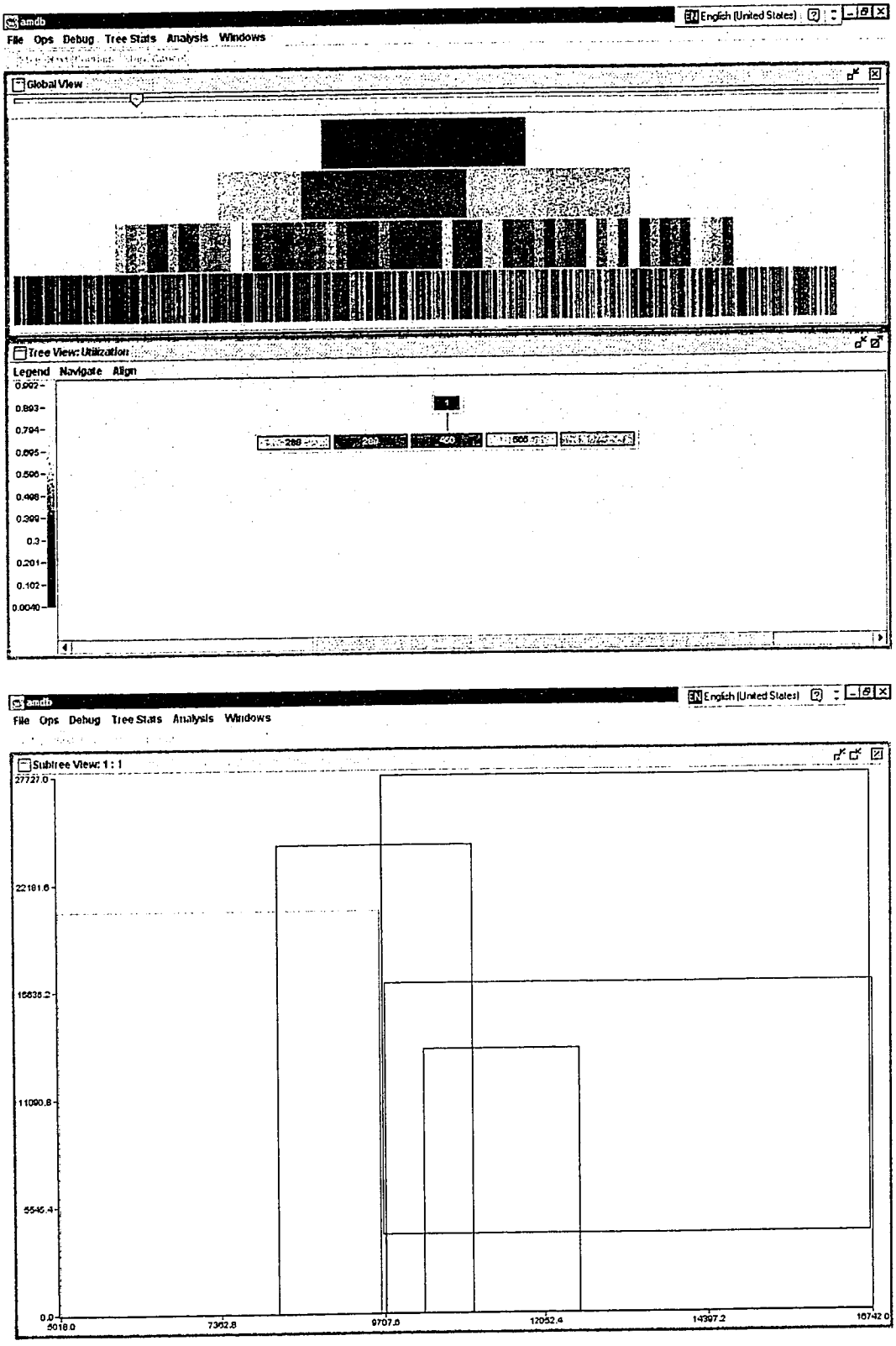


Figure 52 - R4_rect_UC

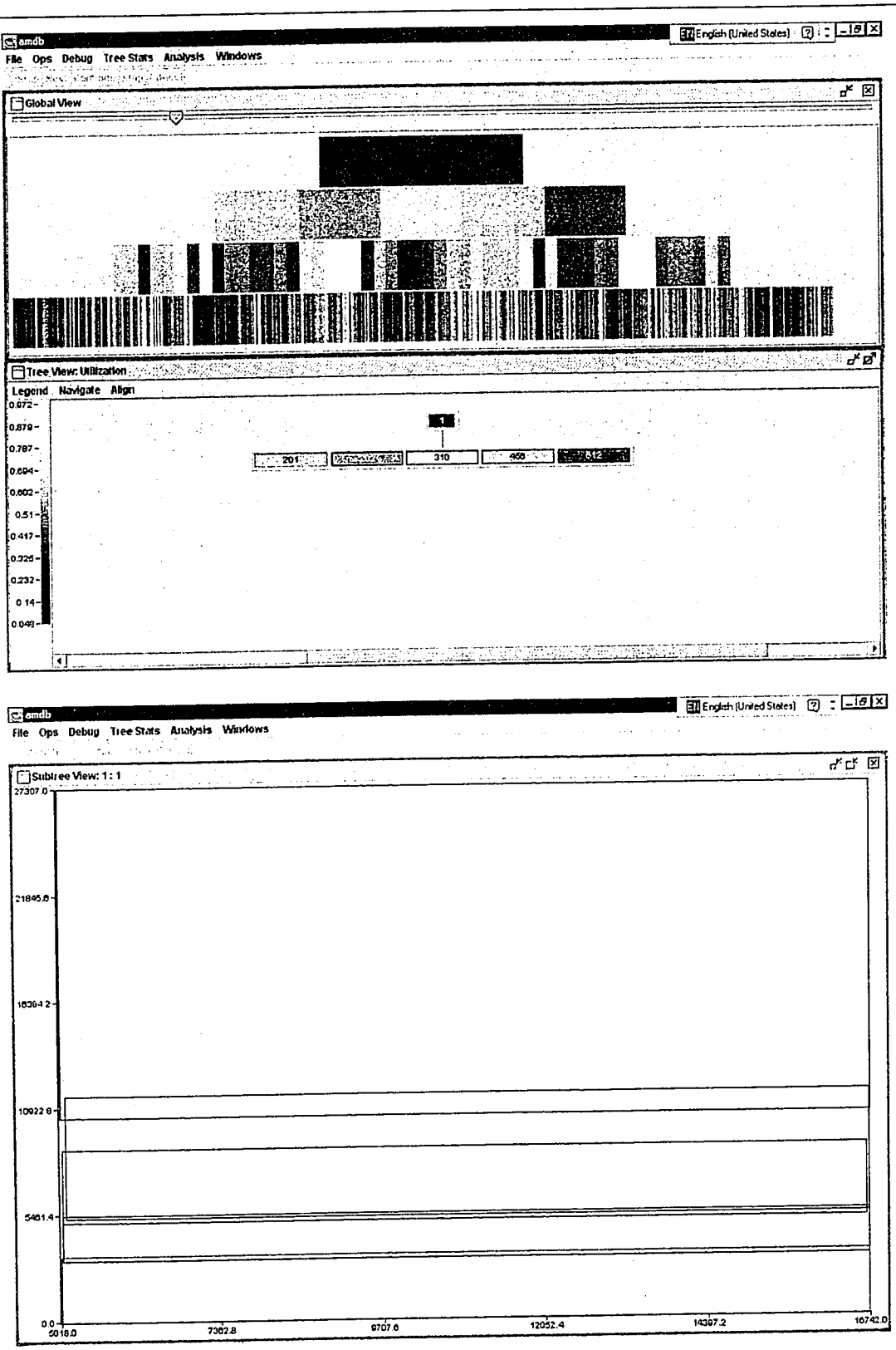


Figure 53 - R4_point_UC

Another result from this experiment is that the R4_point always requires the least amount of disk space. This is because the R4_point_UcNow leaves contains only two dimenensional points which allows leaves to hold twice as many keys. Figure 55 shows that the R4_point_UcNow subtree holds the largest number of keys (39%). Also the R4_point has the least total number of nodes (see appendix A.1).

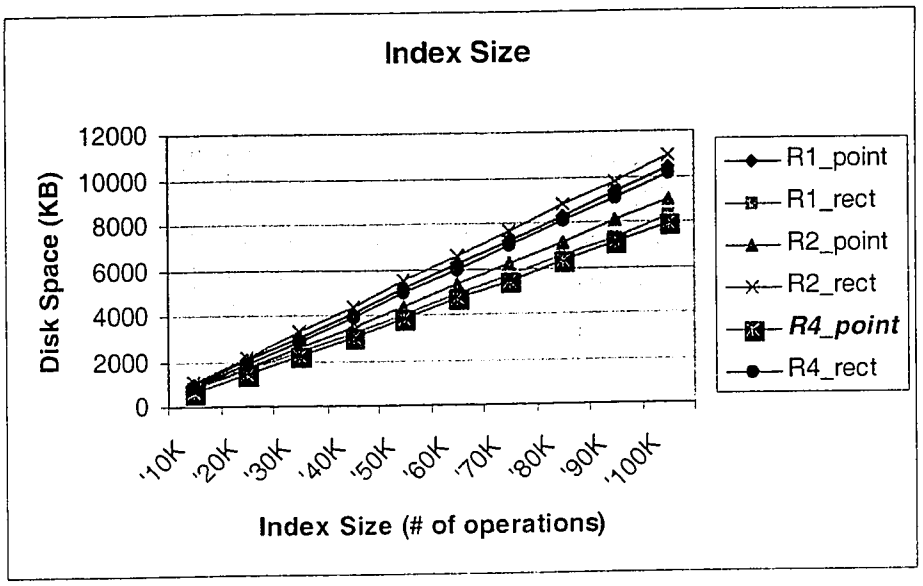


Figure 54 - Disk space used by each index

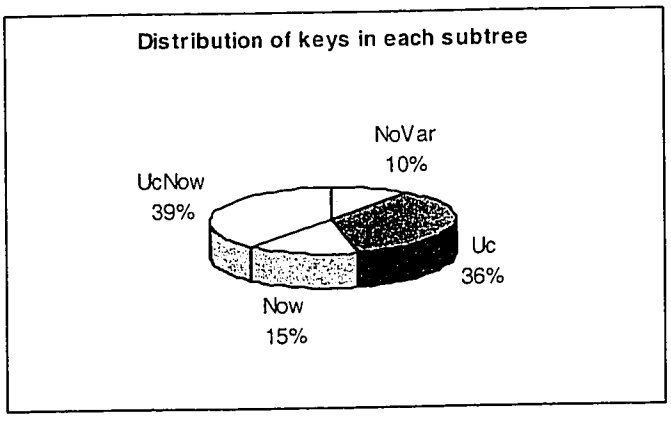


Figure 55 - Distribution of keys in each subtree

4.3.3 Effect of 'pNow'

In this experiment, we study the performance of the six indices as a function of pNow (i.e. Probability of data with VT^e = *Now*). Figure 56 shows that the number of I/Os increases as pNow increases. This because the size of the search results increases as most of the queries (Qcur = 65%).

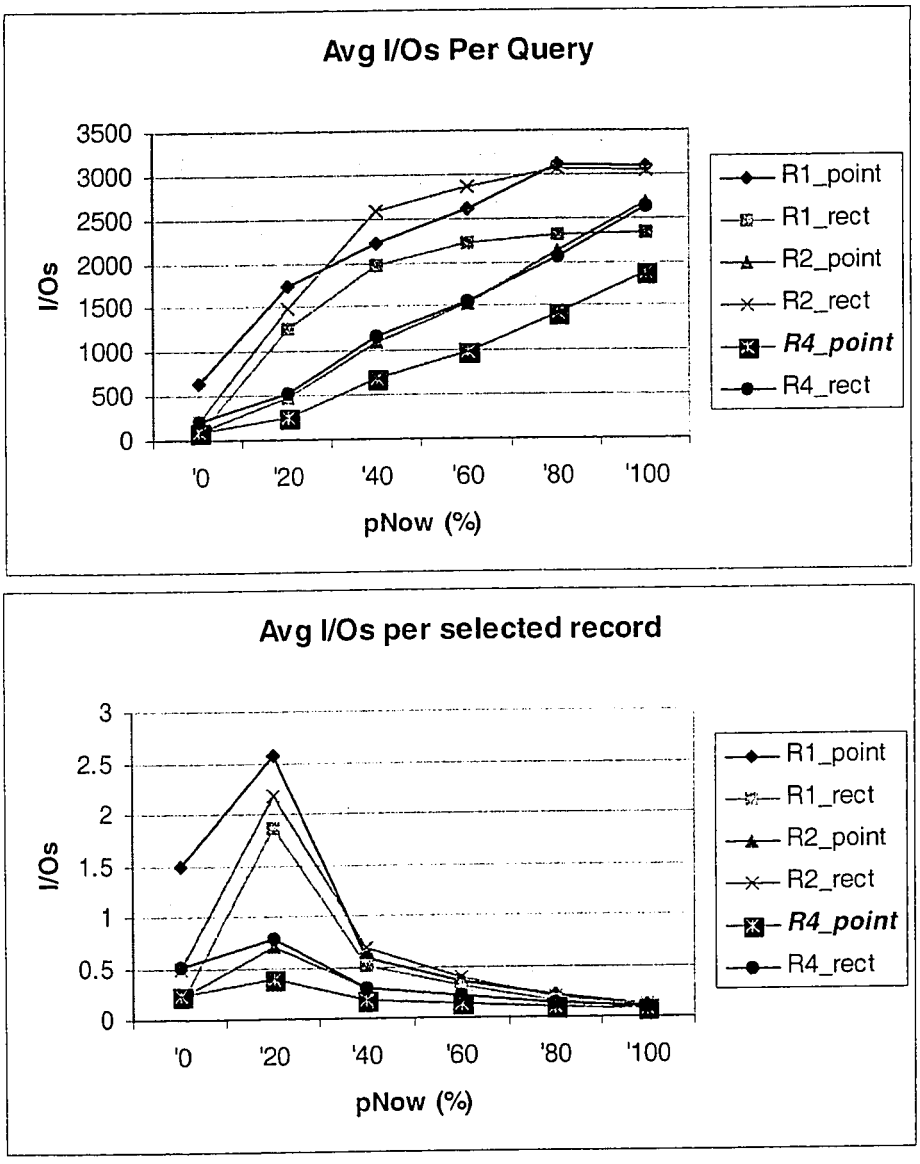


Figure 56 - Search performance for varying pNow

However, if we normalize the search I/Os against the size of the query results, the experiment shows that the worst search performance occurs when 20% of the load operations have VT^e = *Now*, while the other 80% have a defined VT^e. The R4_point is the least sensitive to changes in the mix of pNow insertions, while R1_point, R1_rect and R2_rect are more sensitive to the mix of pNow insertions. In all cases, the search performance of R4_point is better than other indices.

Now let's analyze why the search performance deteriorates in all indices when pNow is 20%.

First we will consider the distribution of the keys between the different subtrees to understand which subtree effects the performance the most.

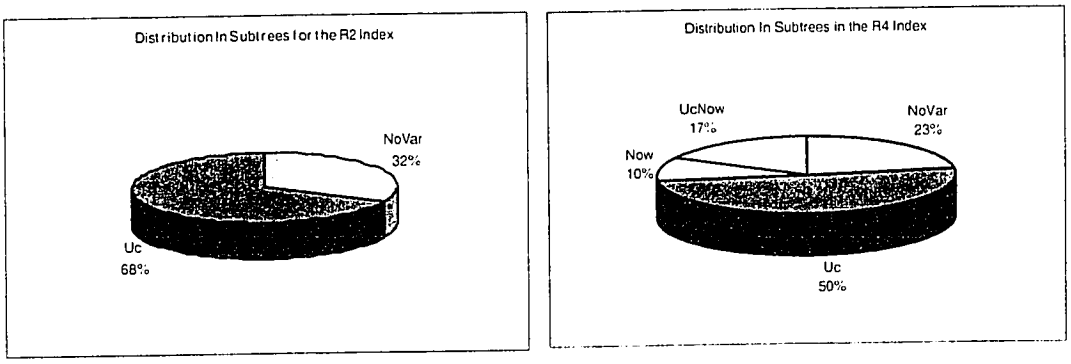


Figure 57 - Distribution of the keys in the subtrees of the R2 and R4 indices

As shown in Figure 57, the majority of the keys are stored in the UC subtree. Note that both subtrees in the R2 indices still contain keys with VT^e = *Now* which means that corresponding rectangles still expands to maximum value. In the point versions of the indices, the points will be located in the top of the space. This results in a very bad partitioning of the space. Figure 58 shows that all rectangles in the R1_rect are covering all the temporal space and are 100% overlapping.

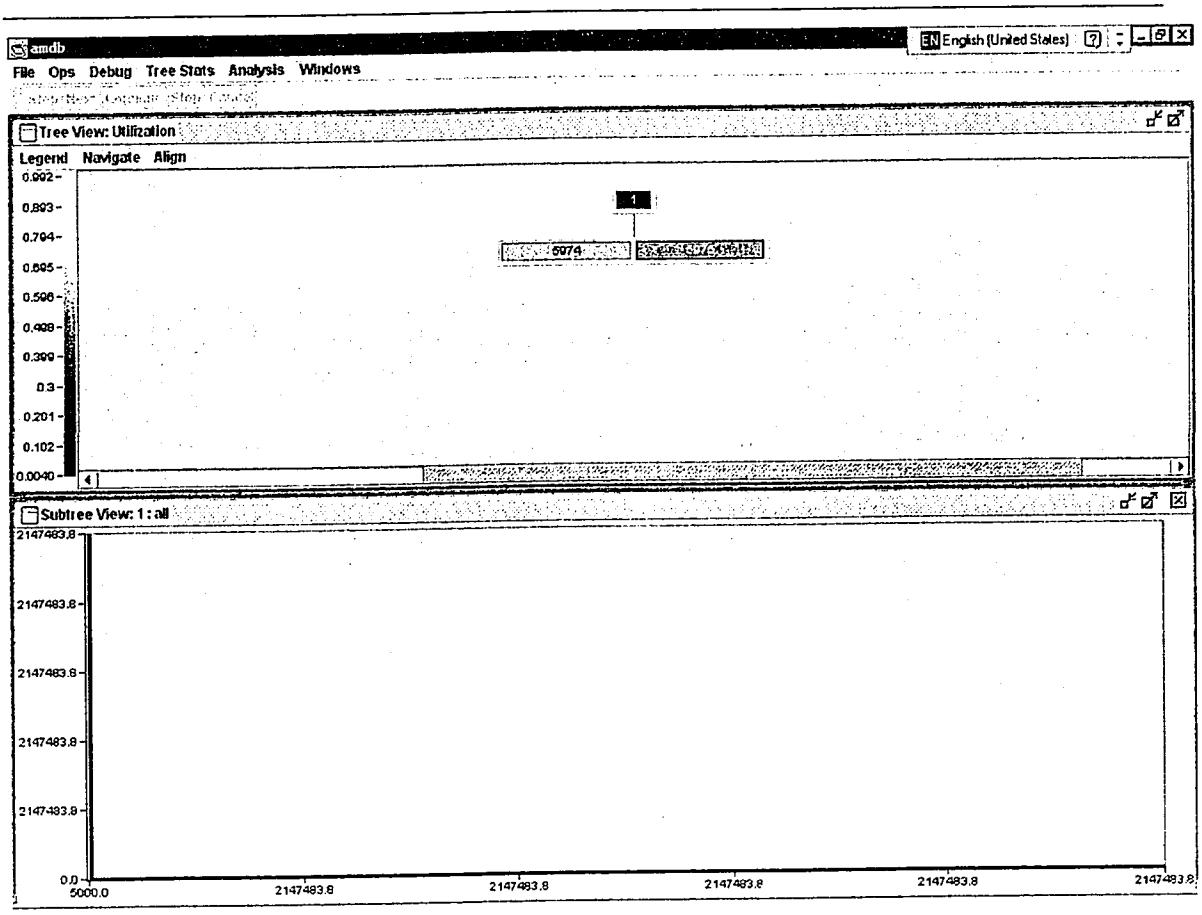


Figure 58 - R1_rect (pNow = 20%)

In the R2 indices (Figure 59 and Figure 60), the keys are first partitioned between the 2 subtrees allowing each subtree to further better partition its space.

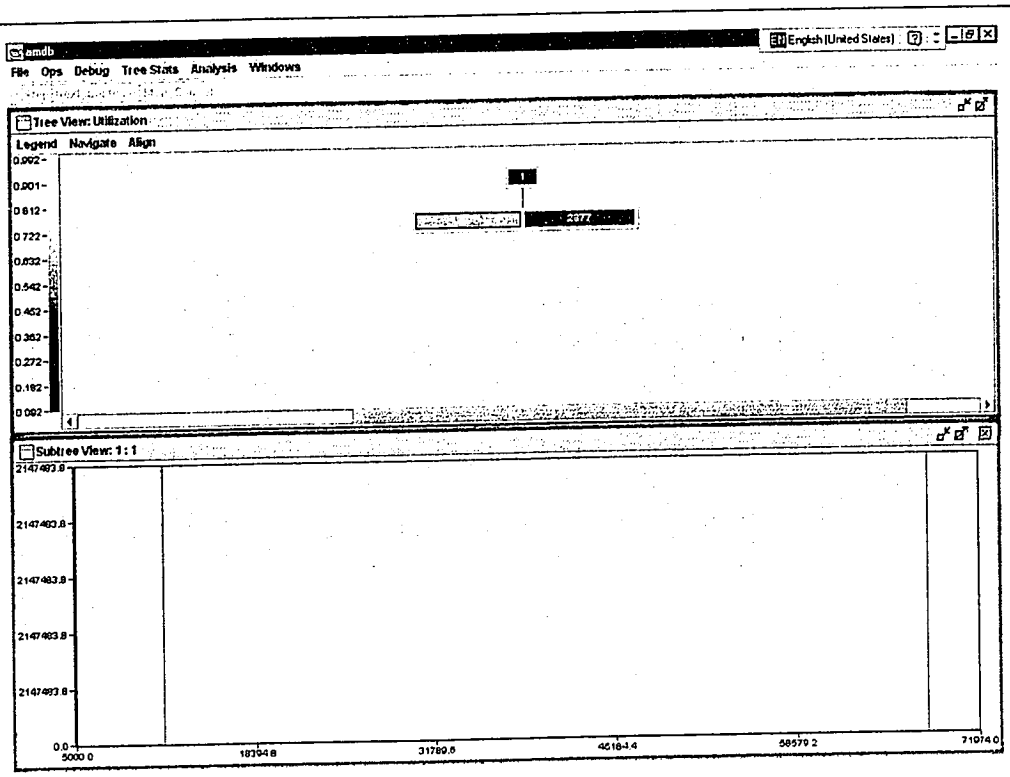


Figure 59 - R2_rect_NoVar (pNow = 20%)

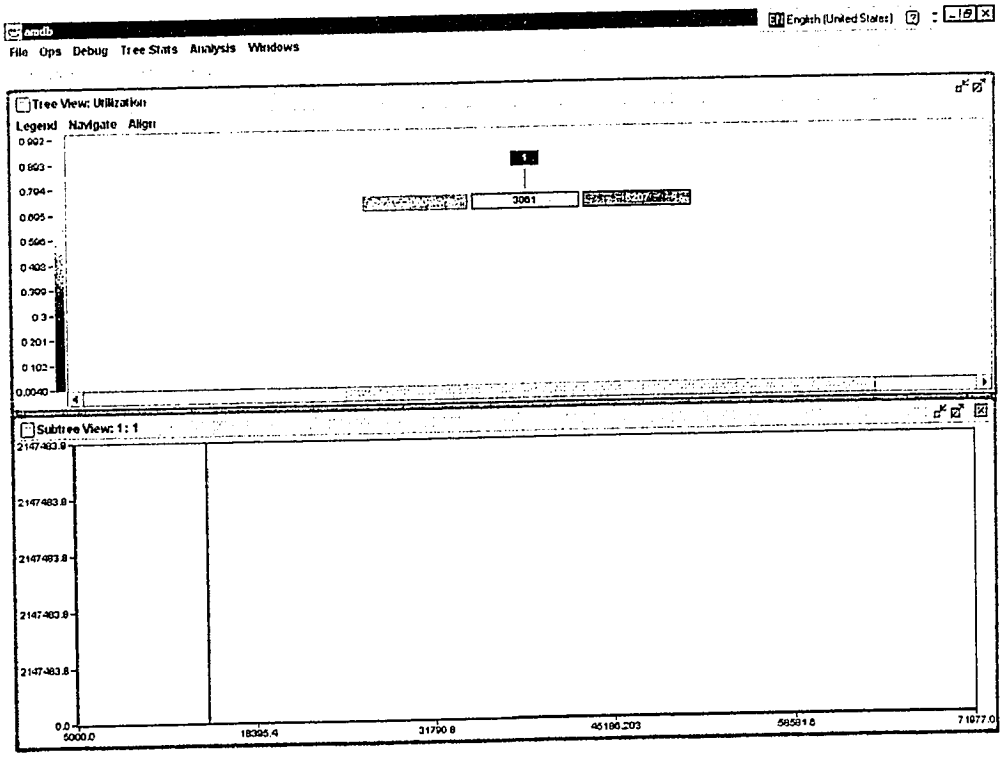


Figure 60 - R2_rect_UC (pNow = 20%)

In the R4 indices (Figure 61, Figure 62, Figure 63, and Figure 64), the keys are first partitioned between the 4 subtrees, then each subtree further partition its own space.

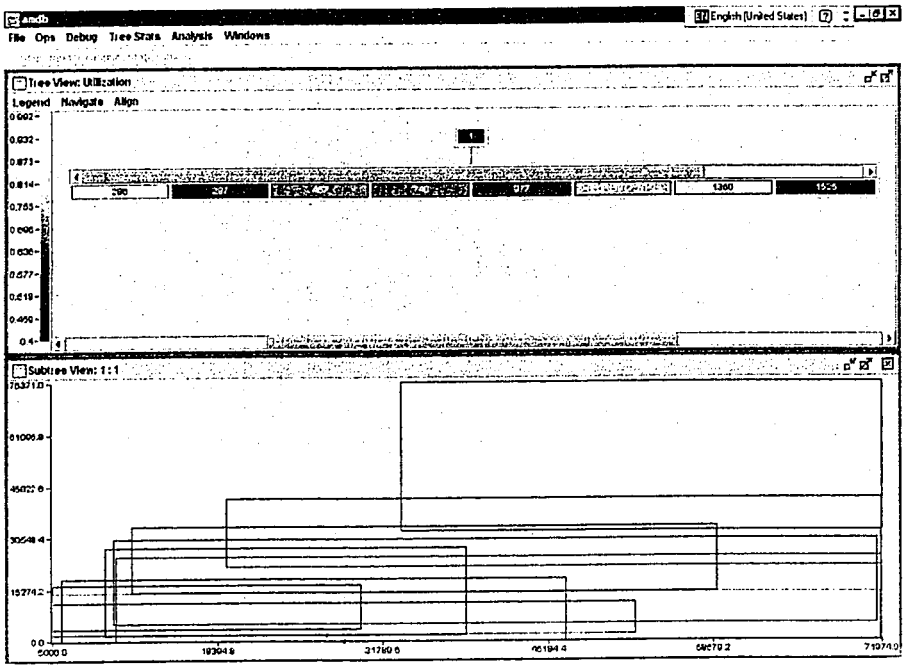


Figure 61 - R4_rect_NoVar (pNow = 20%)

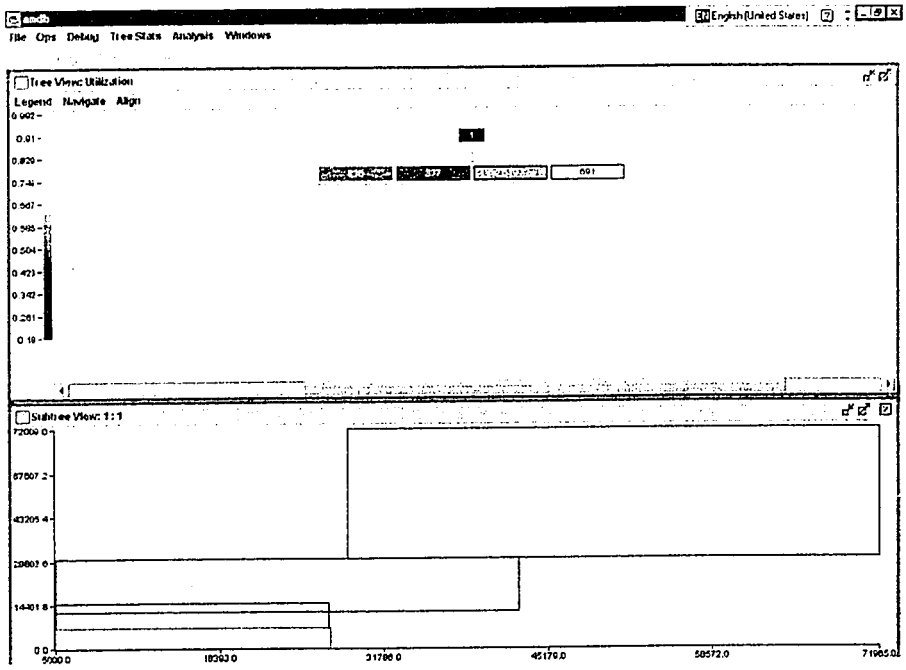


Figure 62 - R4_rect_Now (pNow = 20%)

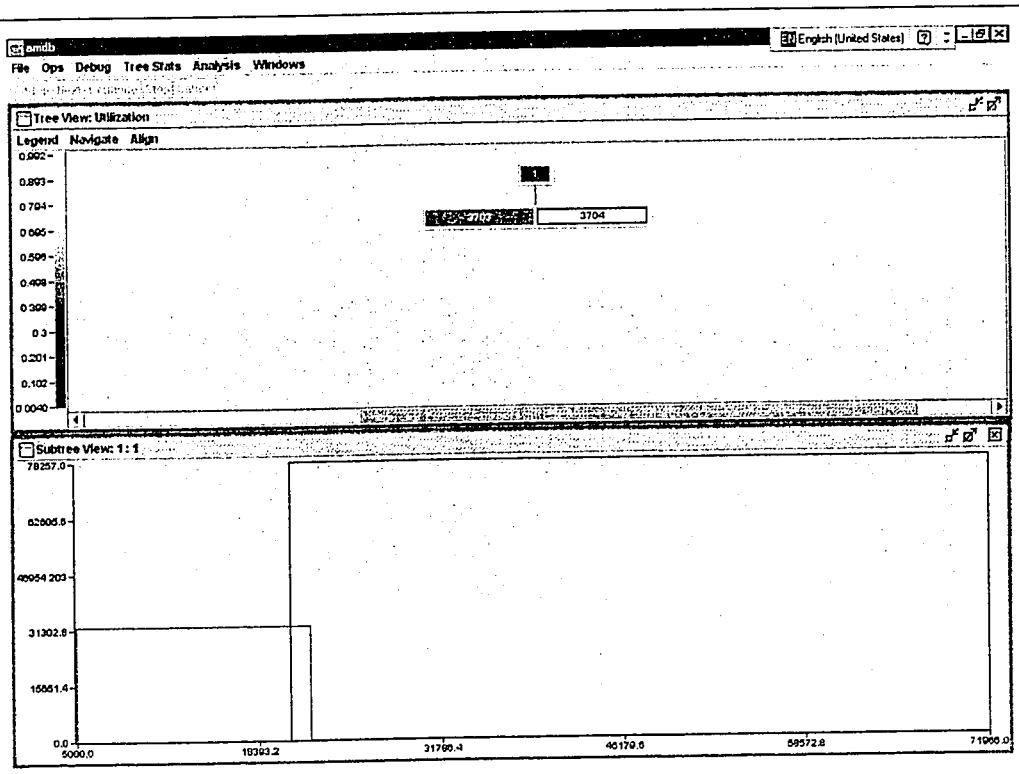


Figure 63 - R4_rect_UC (pNow = 20%)

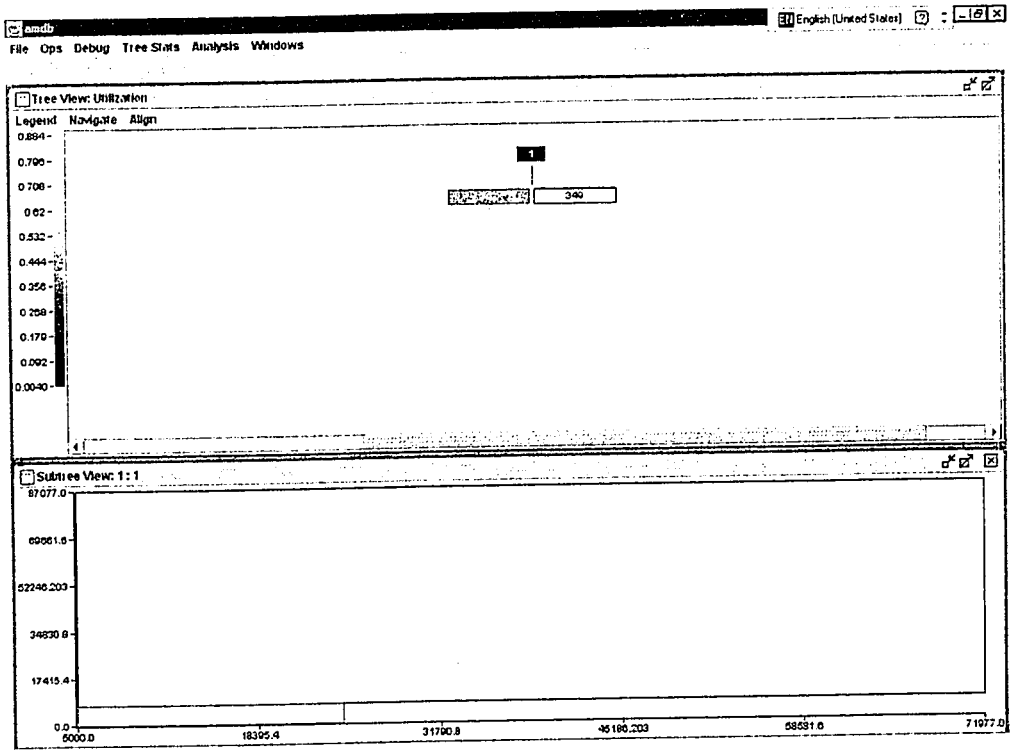


Figure 64 – R4_rect_UcNow (pNow = 20%)

Another factor that we will need to consider is the size of the query results. The number of I/Os shown in "Avg I/Os per selected record" in Figure 56 is calculated by dividing the number of I/Os used during the queries by the number of keys returned in the search results. Since pNow = 20%, then a very small number of keys is expected to be retrieved by the queries which 65% are current queries. Figure 65 clearly shows that the size of the search results in case of pNow = 20% is very small. However, the number of pages that needs to be searched is high because most pages contains keys with both VT^e = ***Now*** and VT^e not equal ***Now***. This explains why we have such bad performance with pNow = 20%.

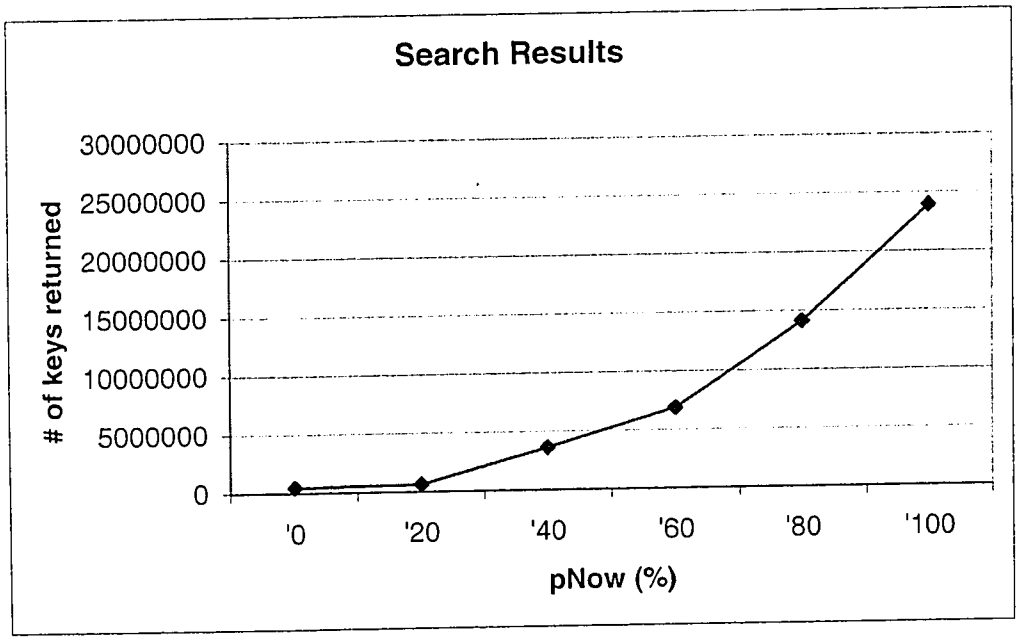


Figure 65 - Size of search results with varying pNow

So far, we have just considered the search performance. We now presents the update performance.

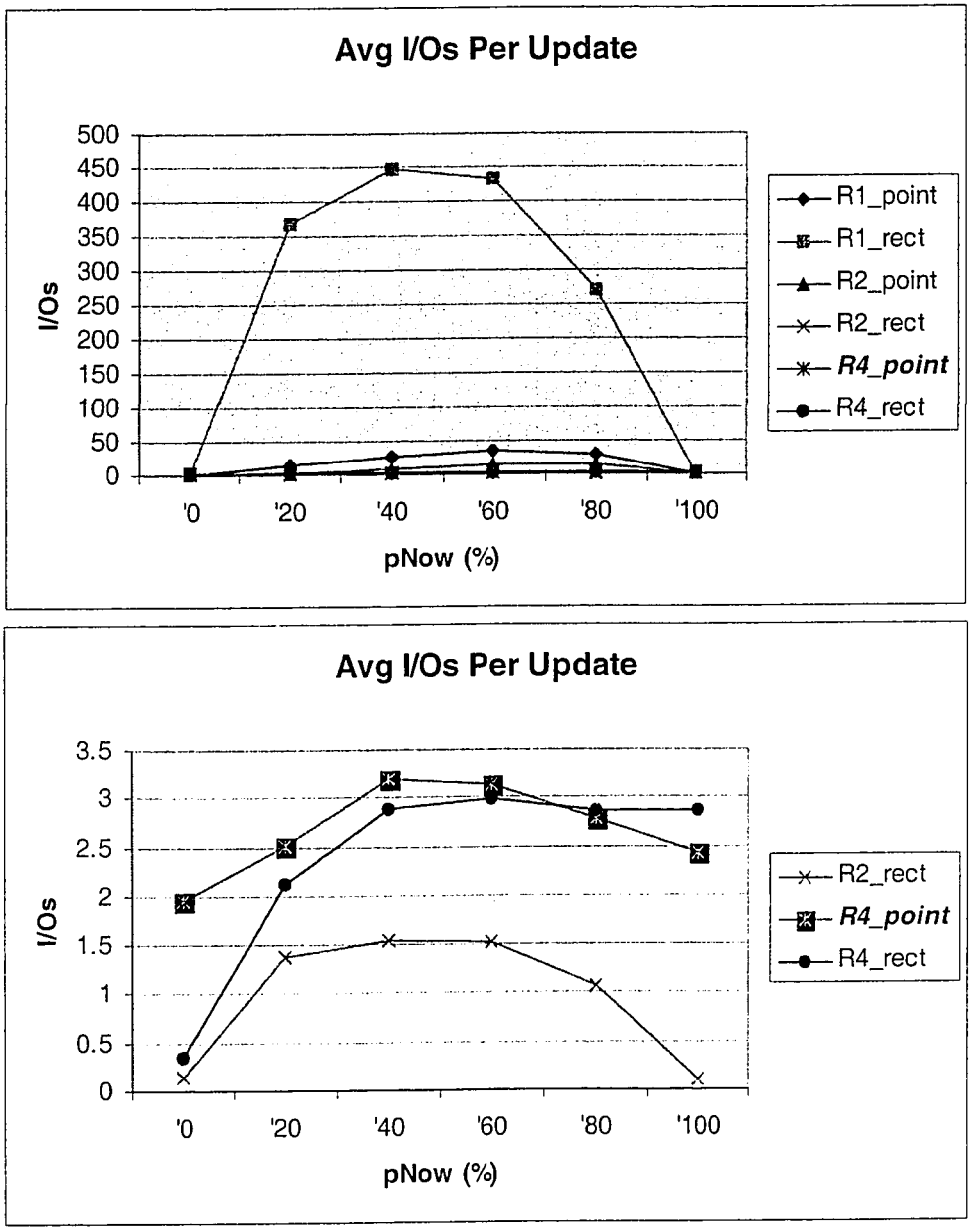


Figure 66 - Avg I/Os per Update for varying pNow

The R2_rect has the best update performance. The index partitions are long stripes extending to the maximum timestamp of VT. As we explained before, due to the sequential nature of transaction time, a node keeps growing in the TT dimension until it get split. No inserts are added to an old split node and no I/Os are really needed until a

new page is needed. One insertion path is always followed most of the time and is stored in the buffers. Figure 67 describes the R2_rect in case of pNow = 100%.

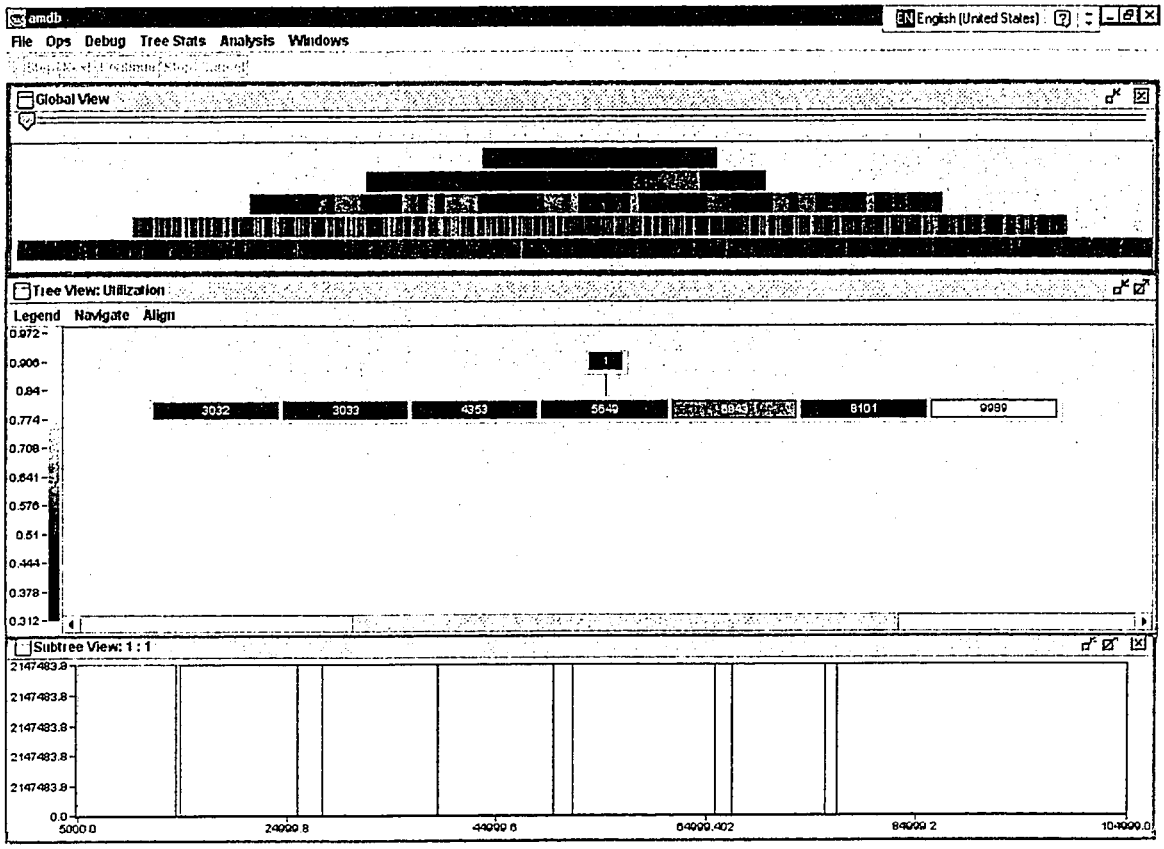


Figure 67 - R2_rect_Uc (pNow = 100%)

The R4 indices however have the second best update performance. The R4_point has even better performance than the R4_rect when pNow = 100%. This is mainly due to the fact that points in R4_point are actually represented by 2 dimensional points which results in storing larger numbers of keys in the leaves. In the R4_rect, points in the *UcNow* subtree are still stored as 2 dimensional rectangles (i.e. 4 points vertices) where $VT^s = VT^e$ and $TT^s = TT^e$.

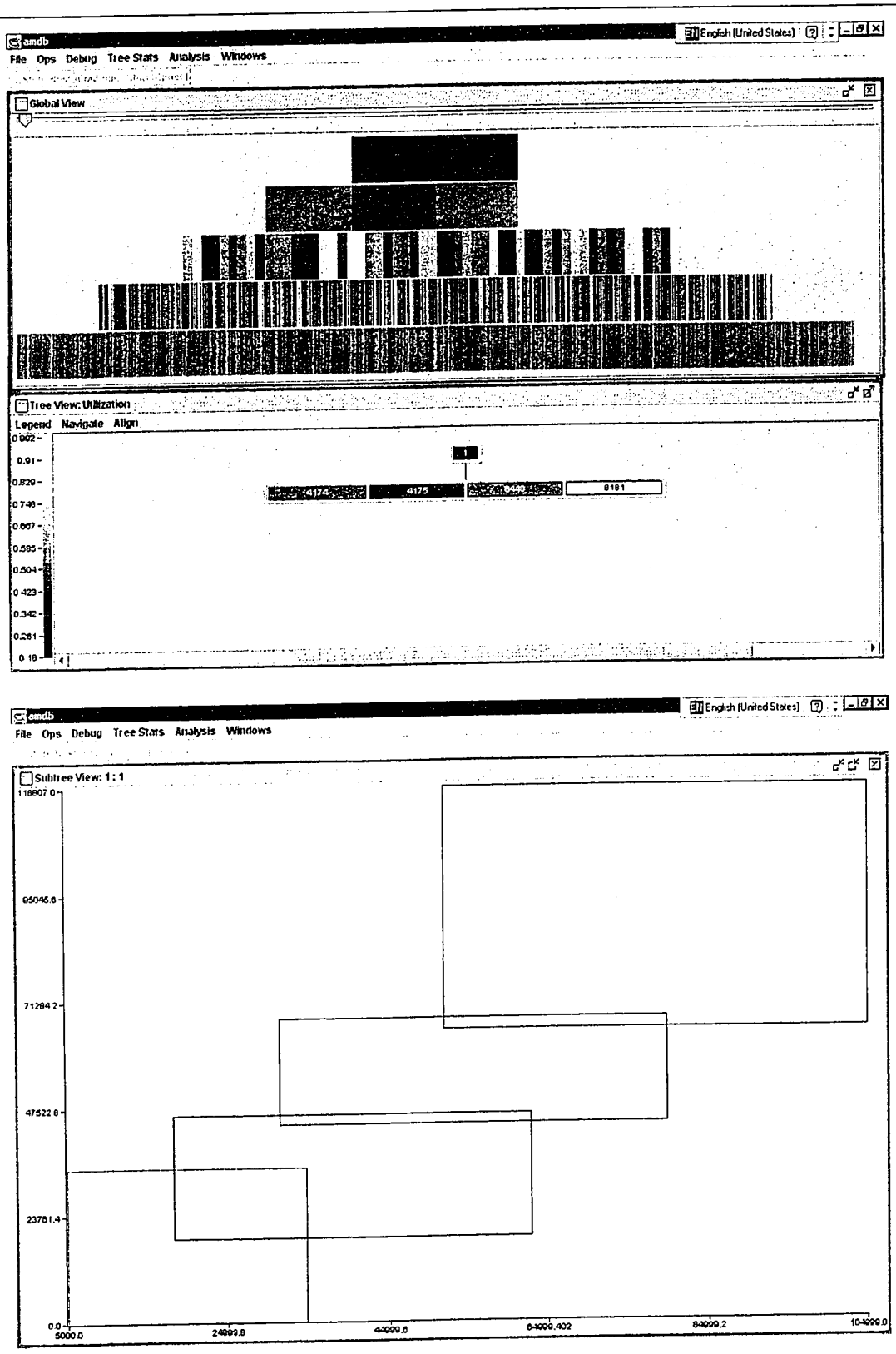


Figure 68 – R4_rect_UcNow (pNow = 100%)

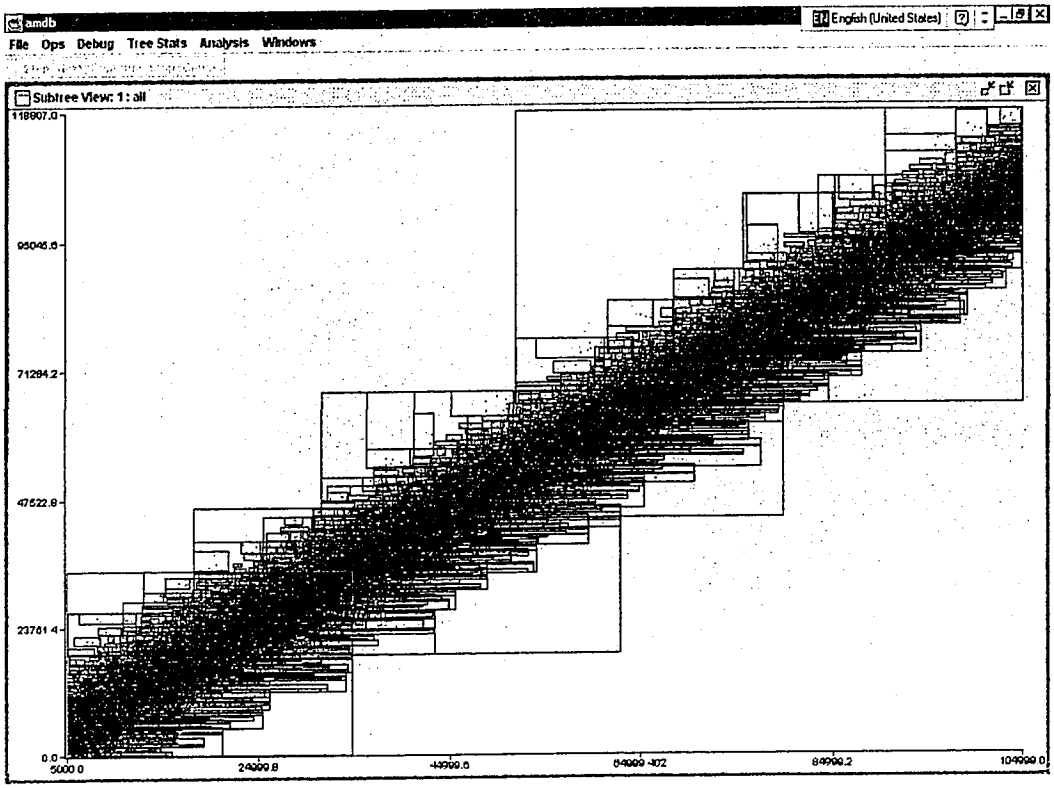


Figure 69 - All levels view for R4_rect_UcNow (pNow = 100%)

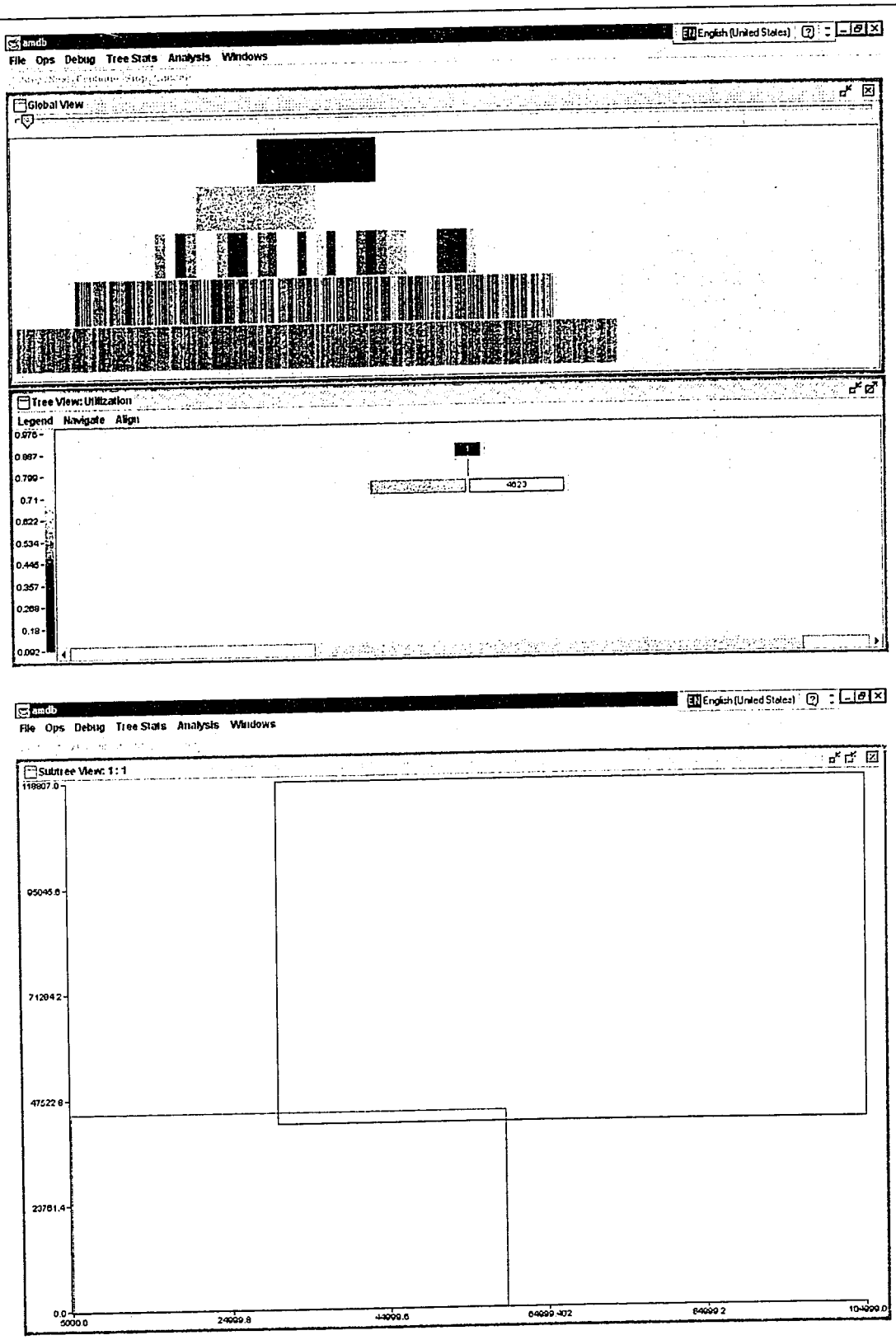


Figure 70 – R4_point_UcNow (pNow = 100%)

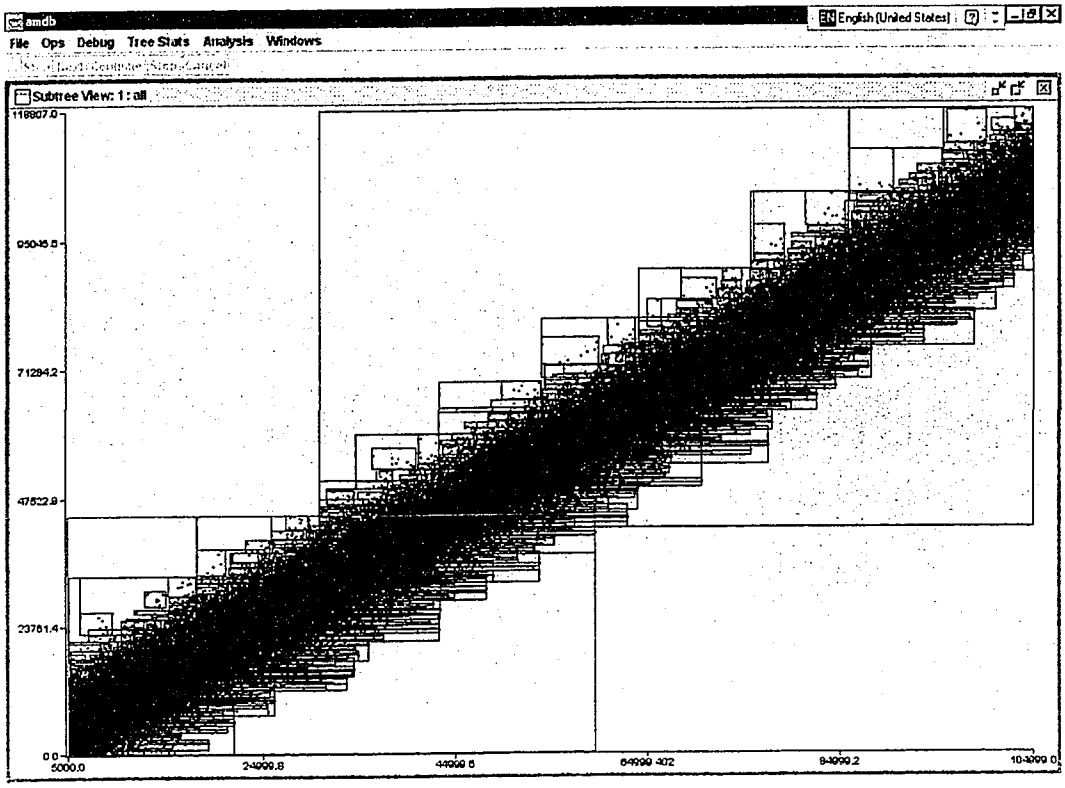


Figure 71 - All levels view for R4_point_UcNow (pNow = 100%)

In this the following subsections, we investigate two special cases:

4.3.3.1 pNow = 0%

In this case, all inserted records have temporal valid time with VT^e not equal *Now*. Of course for the R1_rect and R1_point, 100% of the records are stored in the unique subtree. For others, the records are distributed between the ***NoVar*** and the ***UC*** subtrees as follows:

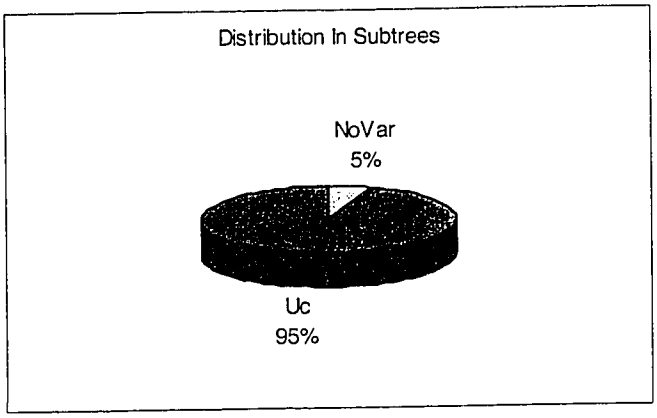


Figure 72 - Records stored in each subtree when pNow = 0%

Although the R4 versions are actually reduced in this case to their corresponding R2 versions, the R2 versions shows better performance. This is because the R2 versions actually use a buffer size that is twice as the buffer size of the R4 versions.

4.3.3.2 pNow =100%

In this case, all inserted records have temporal valid time with VT^e equal ***Now***. Of course for the R1_rect and R1_point, 100% of the records are stored in the unique subtree. For the R2 versions, all records are inserted in the ***Uc*** subtree, while for the R4 versions, all records are inserted in the ***UcNow*** subtree. In the ***Uc*** subtree, the rectangles are represented as intervals (vertical segments) and the points are represented as 3 dimensional points. While in the ***UcNow*** subtree, the rectangles and points are represented as 2 dimensional points. This representation affects the clustering of the data within the different indices and consequently the selectivity of the nodes.

4.3.4 Performance as a function of 'Ins'

In this experiment, we study the effect of varying the percentage of insert operations. A higher percentage means that we have fewer update operations (i.e. fewer delete operations). Also this means that in the 2R indices, most of the keys will be stored in the ***UC*** subtree. While for the 4R indices, most of the keys will be stored in the ***UC*** and ***UcNow*** subtrees. An Ins = 100% will mean that only the ***UC*** subtree is used for the 2R indices, and only the ***UC*** and the ***UcNow*** subtrees are used in the 4R subtrees, while the other subtrees will be completely empty.

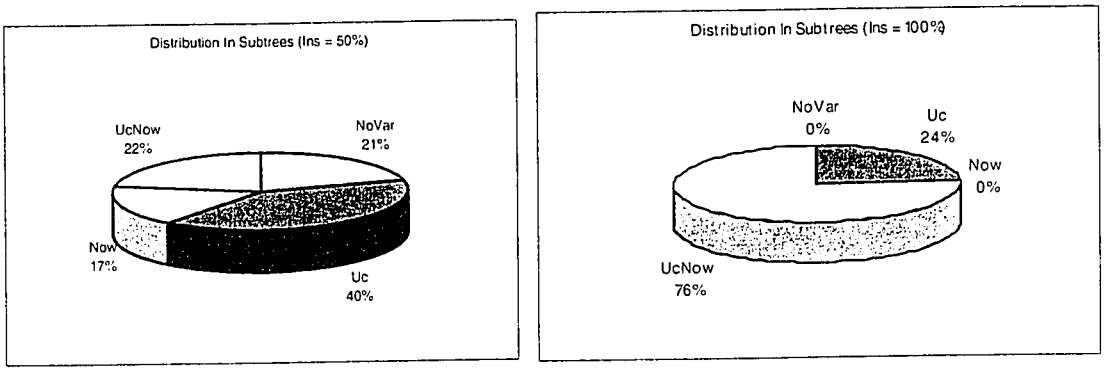
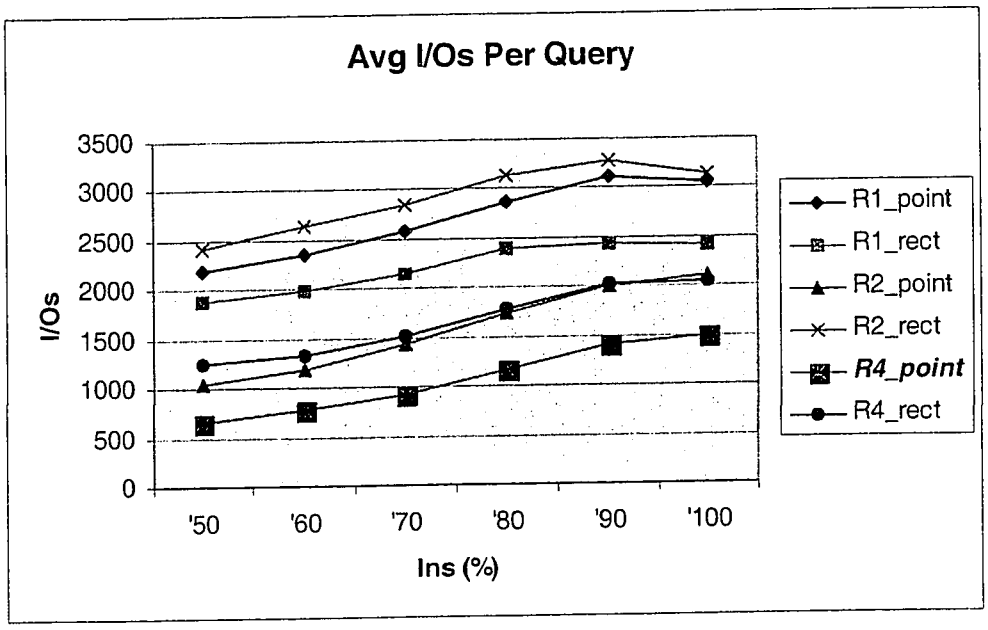


Figure 73 - Distribution of keys in Subtrees as *Ins* varies

Figure 74 clearly shows that as ***Ins*** increases, the performance of all indices is almost the same. The 4R_point index still has the best performance.



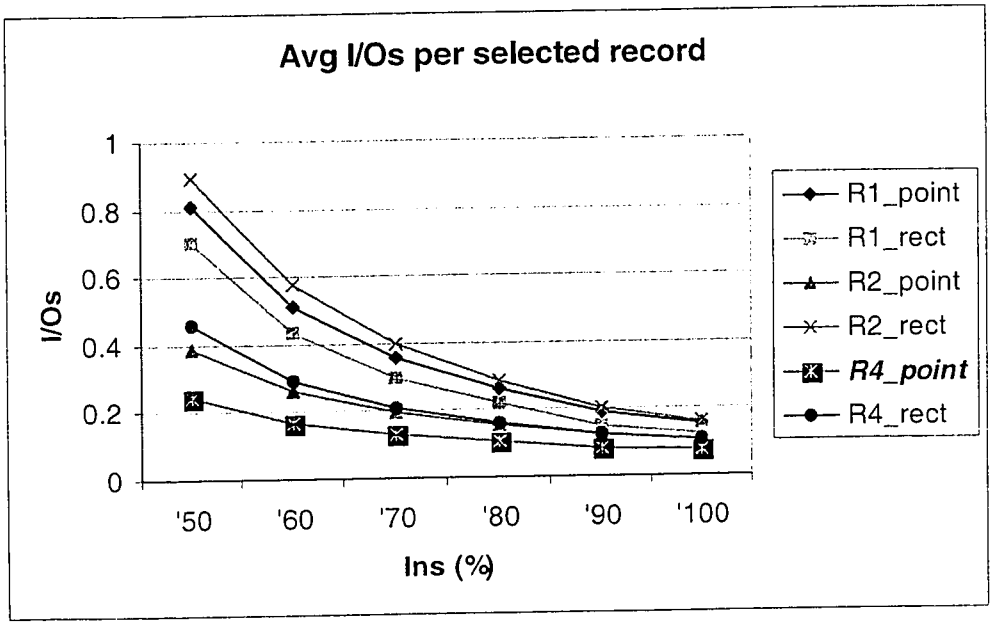
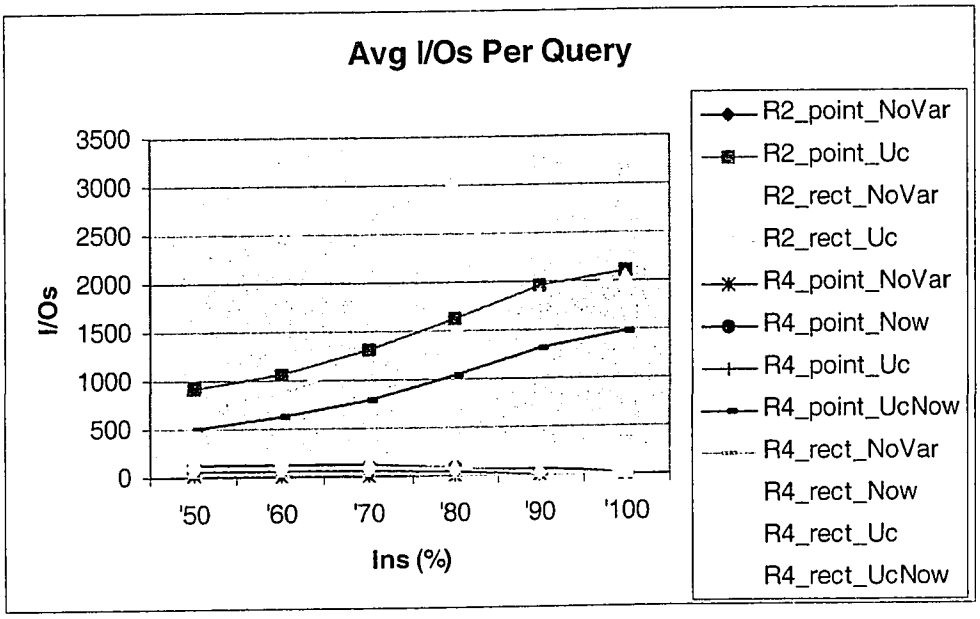


Figure 74 - Search performance for varying Ins

Looking closer and comparing the search performance between the subtrees, we find that the worst performance is coming from the R2_rect_UC and R2_point_Uc because most of the search results are found in these subtrees. However, the point version of the R2 index has much better performance.



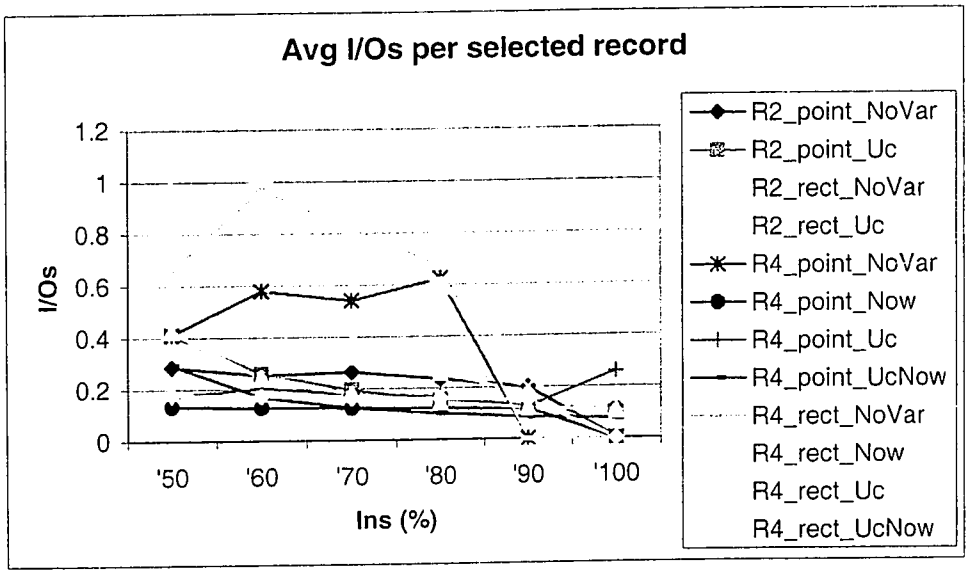


Figure 75 - Comparison of search performance between subtrees

Considering the search performance normalized to the size of the search results, we found that R4_rect_Uc and the R4_rect_NoVar has the worst performance (and selectivity). The corresponding point version subtrees have much better performance.

Also the update performance enhances as the ***Ins*** percentage increases, but the rect version of the trees have slightly better update performance than the point version.

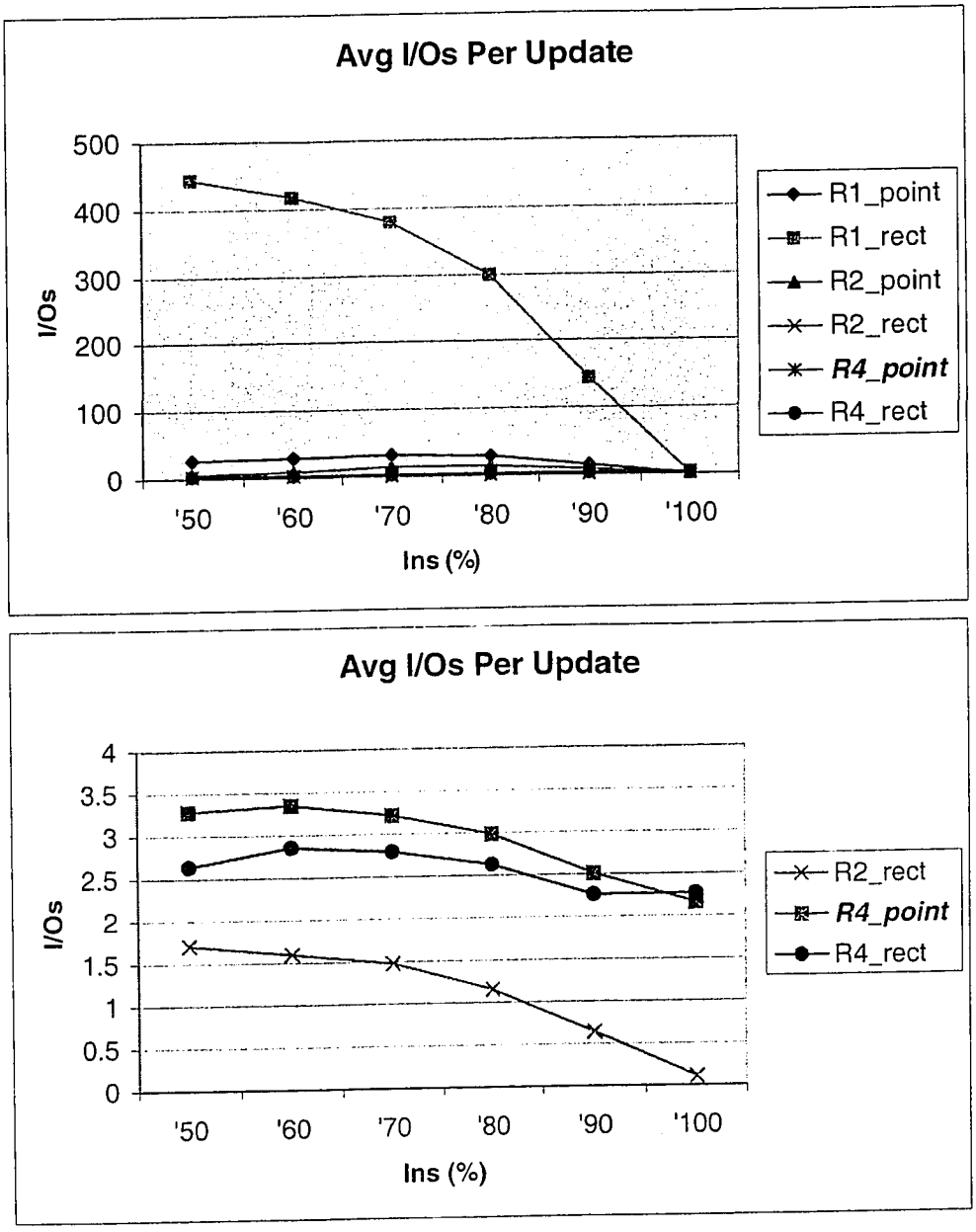


Figure 76 - Update performance for varying Ins

4.3.5 Performance Sensitivity to 'VL'

In this experiment, we study the effect of varying VL (Maximum valid time interval length) on the search and update performance. The results are consistent with the previous results.

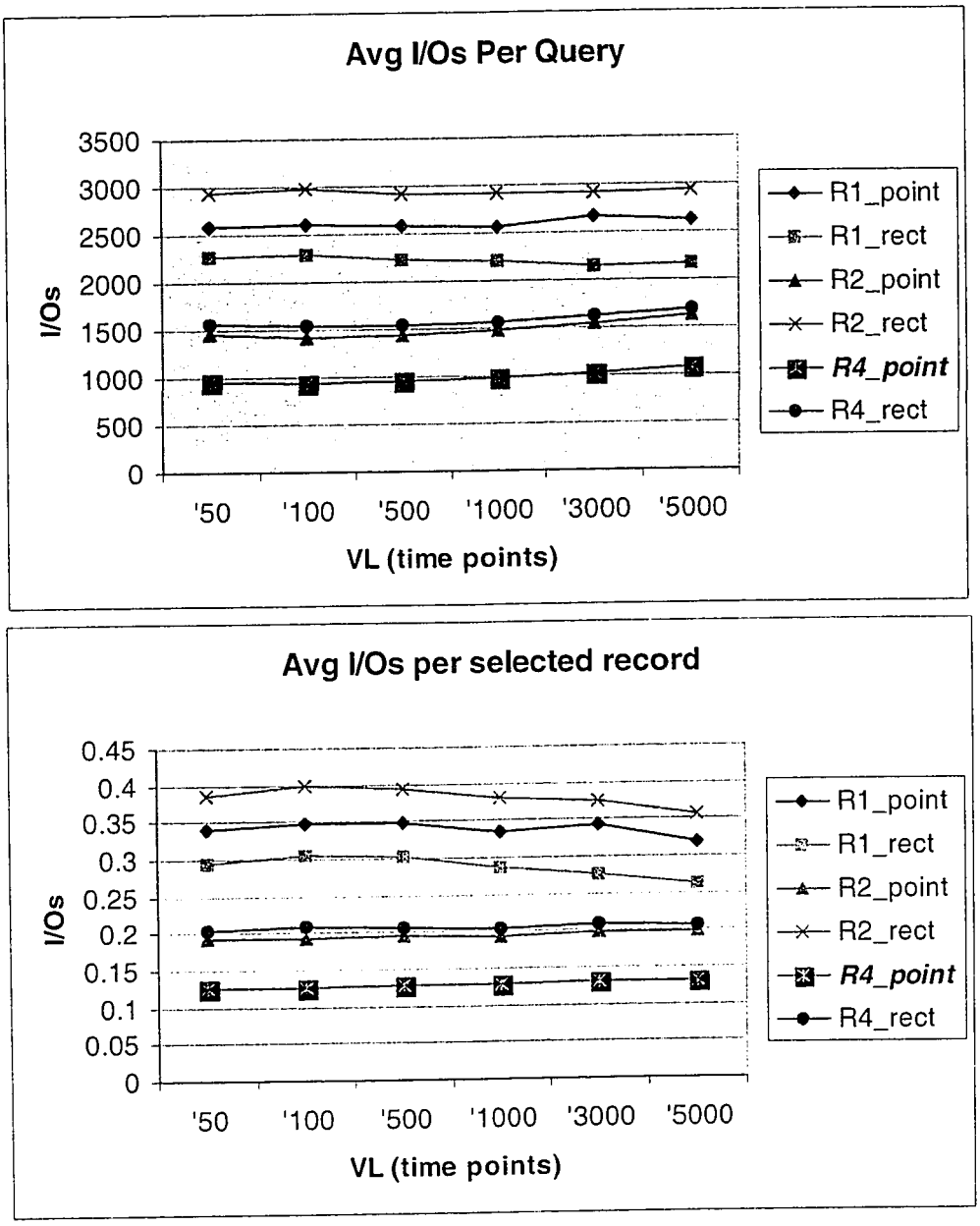


Figure 77 - Search performance for varying VL

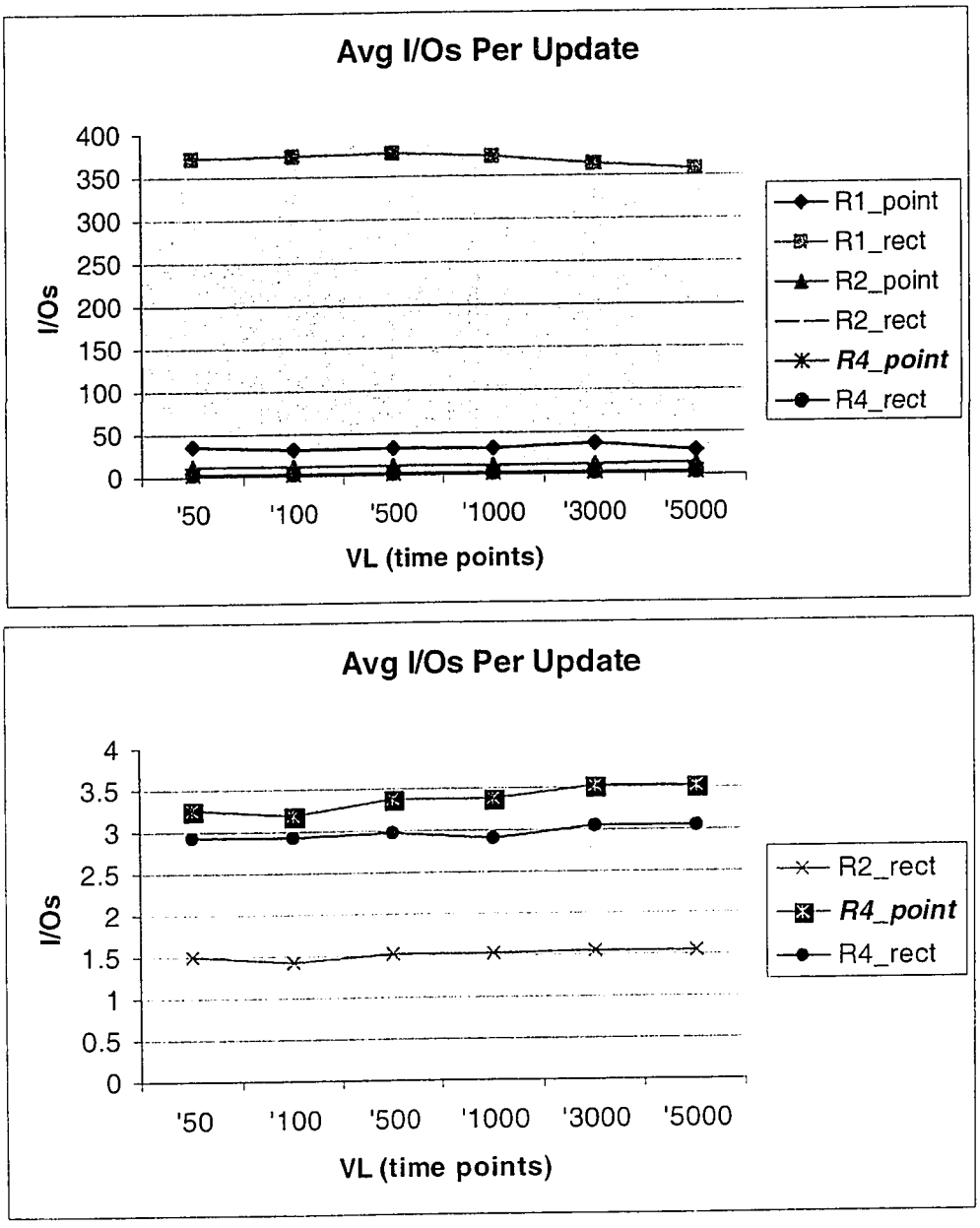


Figure 78 - Update performance for varying VL

Varying the VL value mainly affects the NoVar and the Uc subtrees that contain keys with VT^e not equal to *Now*. This results in larger rectangles stored in the NoVar subtree and longer vertical segments in the Uc subtree. This means that the search results in both these subtrees are larger, which in turn increases the selectivity and enhances the search

performance. However, in the R4_point index, the UC subtree is not sensitive to the change in the VL value.

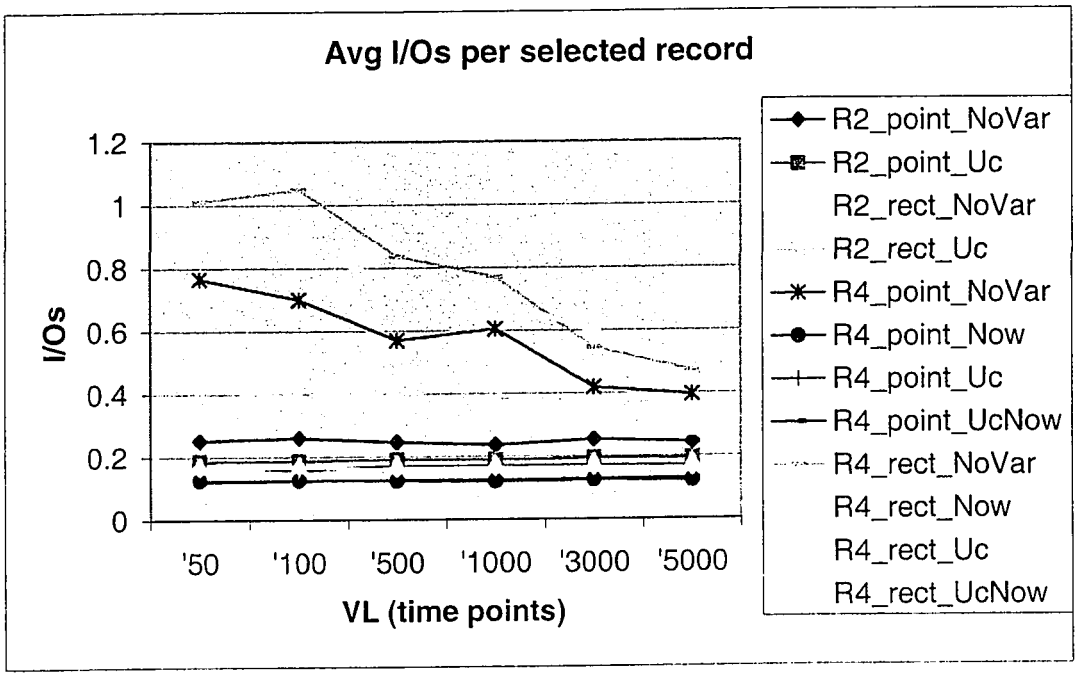


Figure 79 - comparison of R4 subtrees search performance

4.3.6 Impact of 'Dev'

In this experiment, we study the effect of varying Dev (Deviation of VT^s relative to TT^s as mean) on the search and update performance. In case of rectangle and segment representation, the value of Dev controls how long is the rectangle or the segment in the VT dimension. In case of point representation, it determines how far the points are scattered around the VT = TT line.

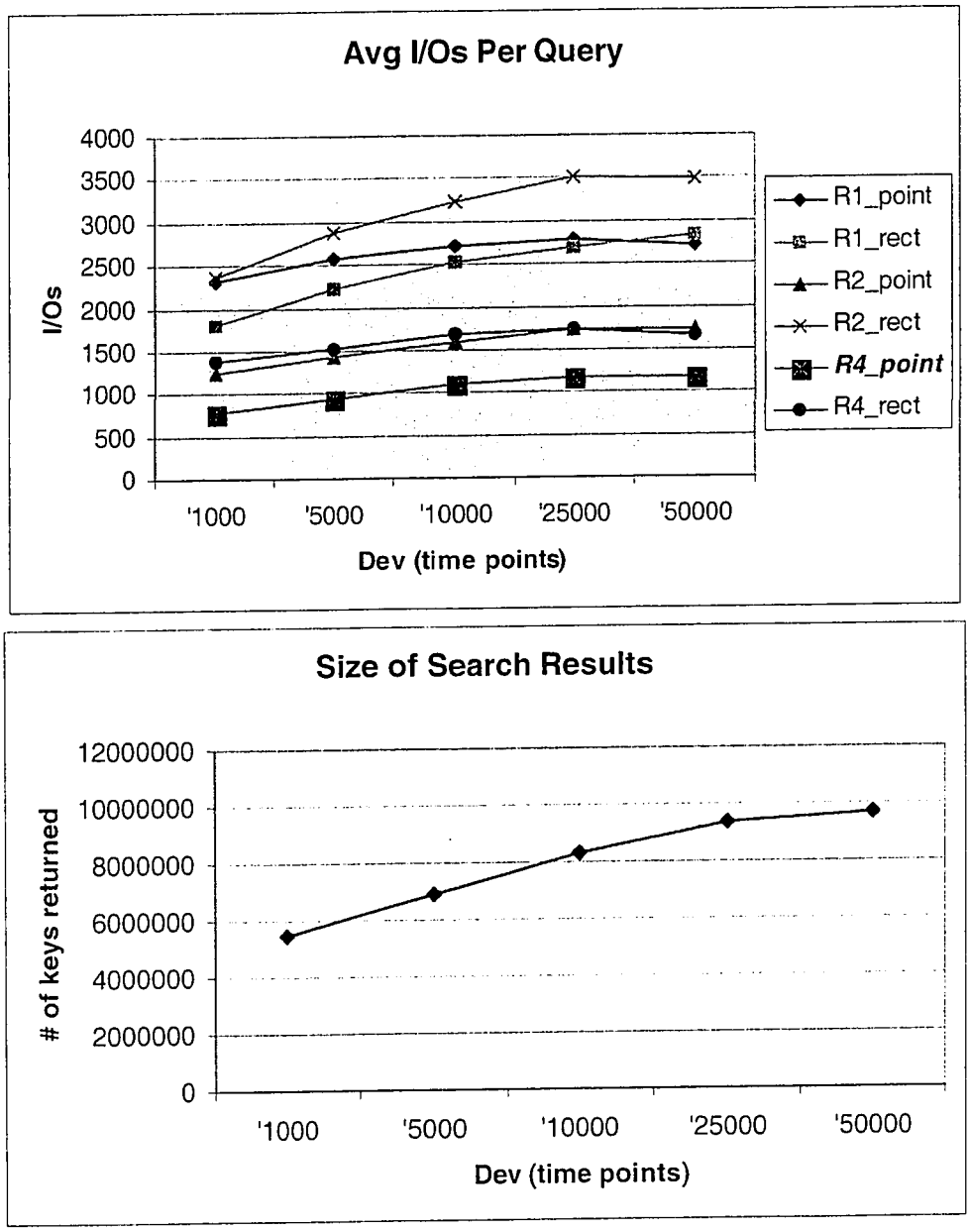


Figure 80 - Search performance for varying Dev

The results are generally consistent with the previous ones. The search performance decreases as Dev increases from 1000 to 25000, then tends to stabilize as Dev = 50000. This is explained by the fact that the search results increase as Dev increases from 1000 to 25000 then remain almost constant when Dev increases from 25000 to 50000.

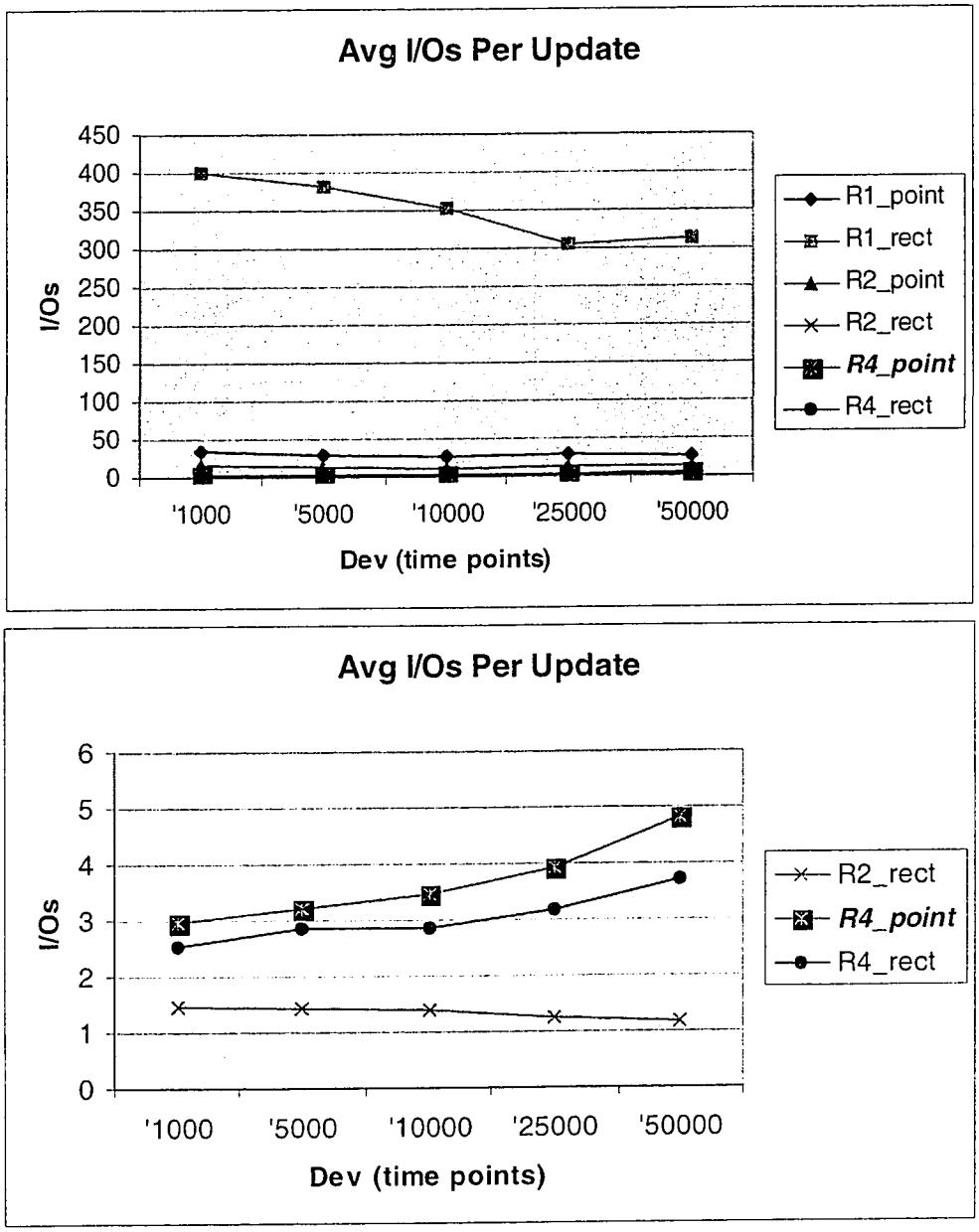


Figure 81 - Update performance for varying Dev

Also as ***Dev*** increases, the number of I/Os used by R2_rect during the update operations decreases, while it increases for the R4. Analyzing this phenomena, we found that the number of operations when Dev = 50000 is smaller.

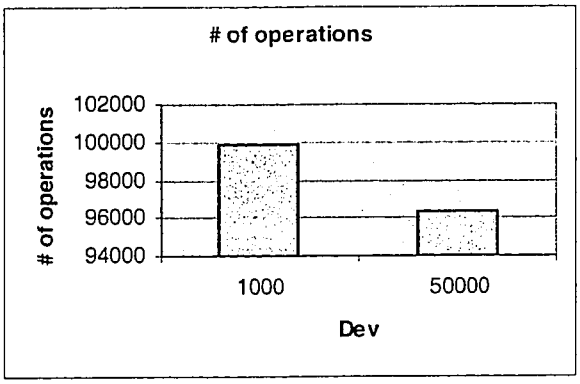


Figure 82 - Comparing the # of operations between the experiments (Dev = 1000 & Dev = 50000)

This explains the enhancement in the update performance for R2_rect, but does not explain the decrease in the update performance for the R4 indices. This means that the update performance even deteriorates more as the value of Dev increases. It was found that the subtree UcNow contributes to this deterioration. This can be explained by the fact that the keys are stored in UcNow subtree as points (TT^s, VT^s). But since Dev is larger, this results that points are scattered more in the temporal space and are then stored in different leaves although TT^s is sequentially increasing. This is also confirmed in the visual partitioning of the subtrees documented in A.26 to A.29 (page 177 to 179).

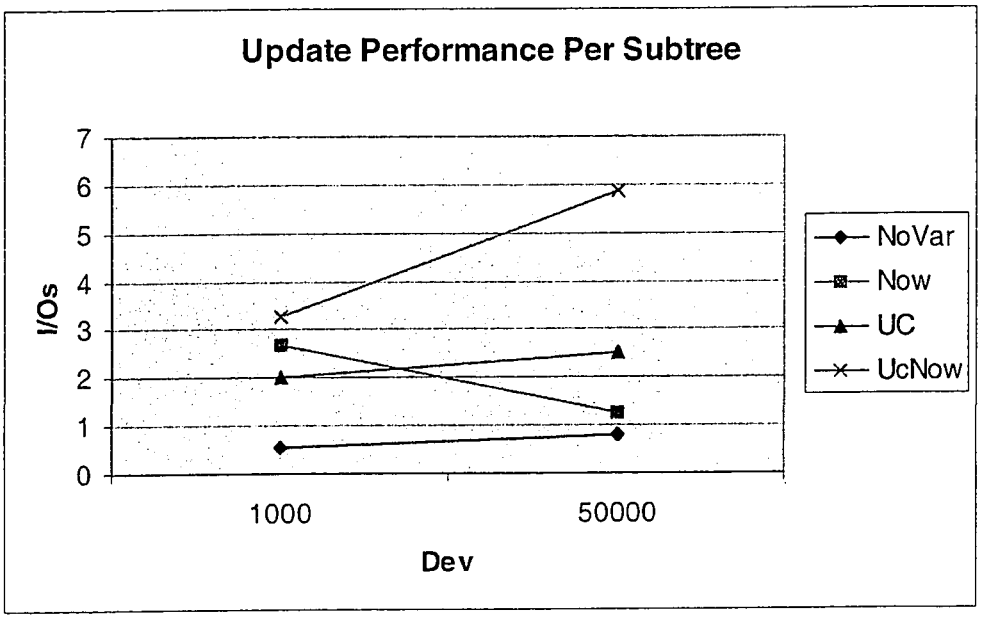


Figure 83 - Comparing update performance for different subtrees for varying Dev

In the case of the R2_rect, the NoVar subtree has smaller height and of course less nodes count (2229 versus 2730). Note that a larger Dev means that the best index is the one capable of partitioning keys that have mixed values of VT.

4.3.7 Experimenting with queries

In this section, we will focus on the query performance. We study different kind of queries, namely:

- Range,
- Timeslice, and
- Point

The timeslice query is the mostly used query especially to retrieve current information. We investigate the effect of varying the percentage of current time queries, and finally we investigate the effect of different VT and TT ranges.

In the first four experiments described in sections 4.3.7.1 to 4.3.7.4, we vary the percentage of queries whose $TT_q^u = CT$. Figure 84 shows that 75% of the keys are current (i.e. $TT^e = UC$), then as *Qcur* increases, the size of search results increases which generally results in higher number of I/Os.

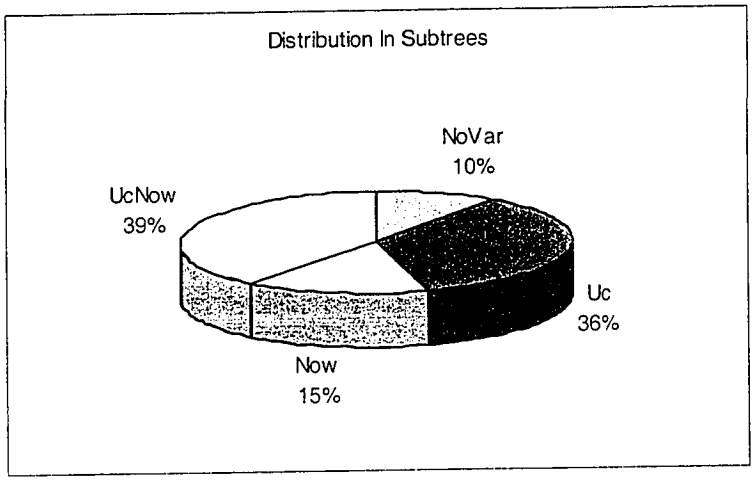


Figure 84 - distribution of temporal keys

4.3.7.1 Varying Current Queries percentage for Range Queries

In this experiment, we study the performance of pure range queries as *Qcur* varies between 0 and 100%. Figure 85 shows that the results are consistent with the previous ones: the R4_point has the best search performance.

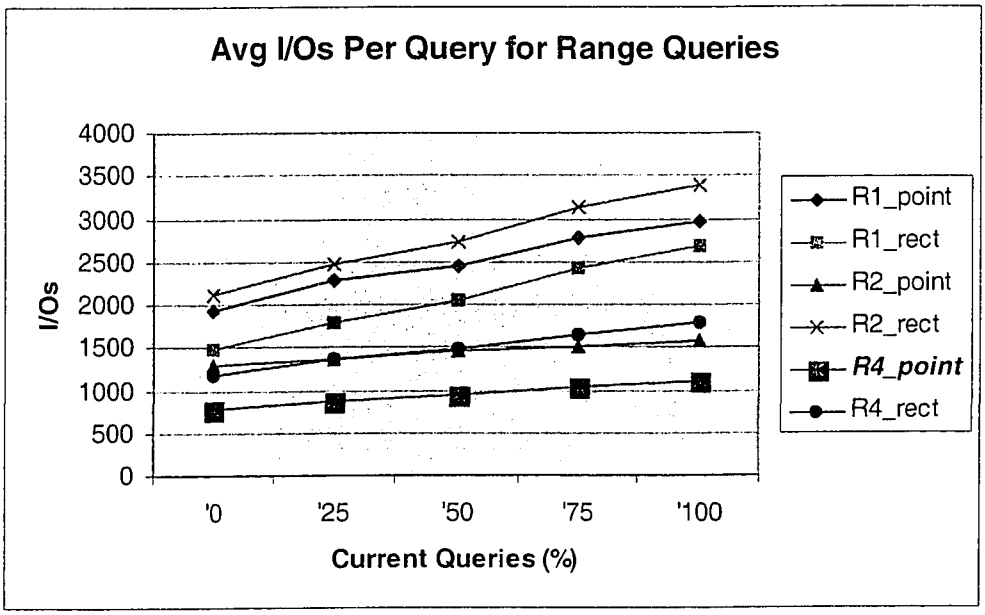


Figure 85 – Search performance of range queries for varying Qcur

4.3.7.2 Varying Current Queries percentage for Timeslice Queries

In this experiment, we study the performance of pure timeslice queries as *Qcur* varies between 0 and 100%. Timeslice queries are the most popular queries in the temporal databases. They are used to view the database at different points of time (i.e. rollback from the transaction time point of view). Figure 86 shows that the R4_point index always exhibit the best search performance for current or historical states of the index. This is the most important characteristic for a good temporal index.

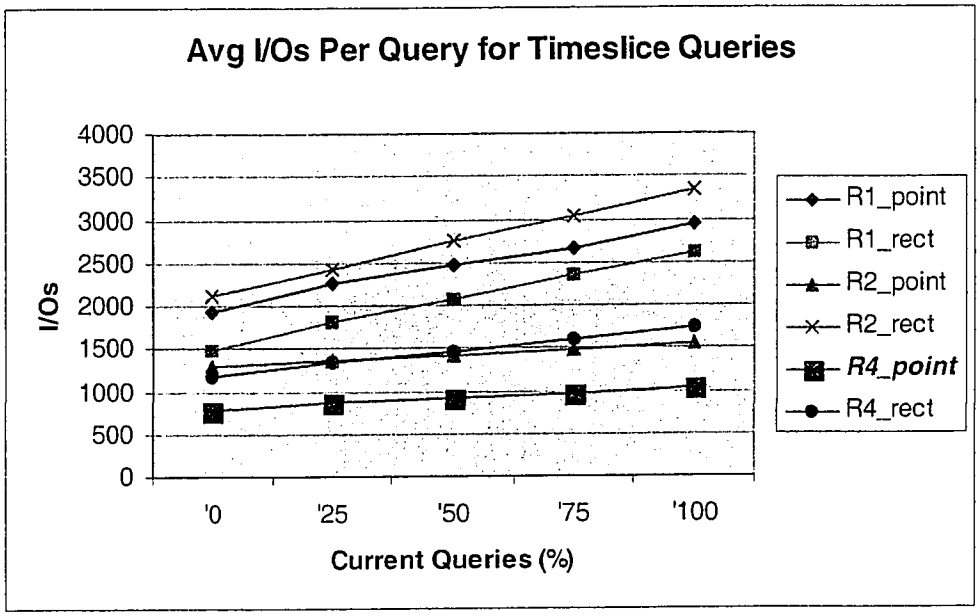
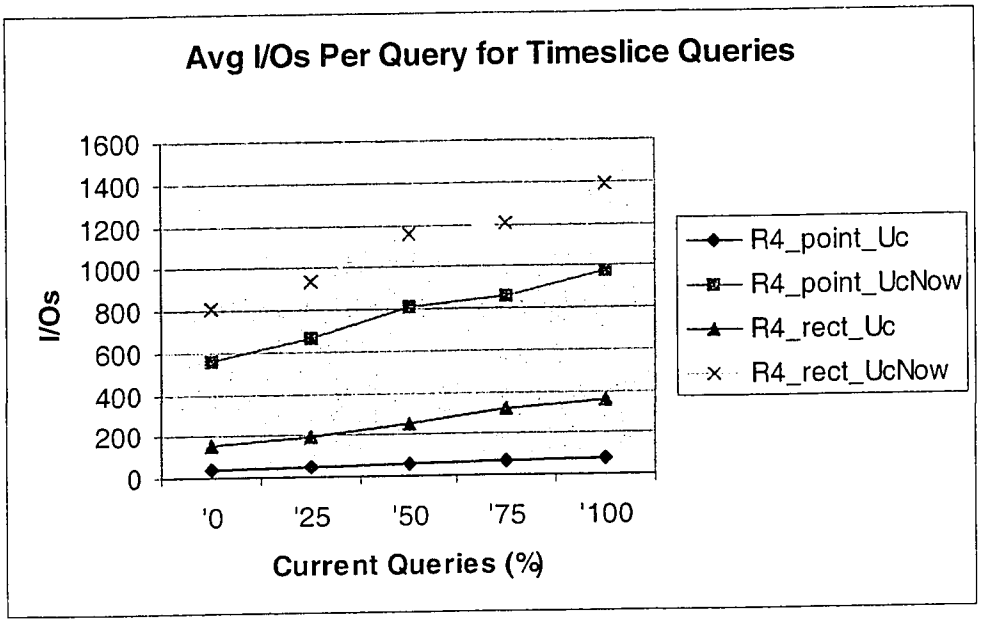


Figure 86 – Search performance of timeslice queries for varying Qcur

Analysis of the query performance for the two variants of the R4 trees shows that the point variant has much better performance than the rectangle variant. This is obvious from the performance graphs of Figure 87.



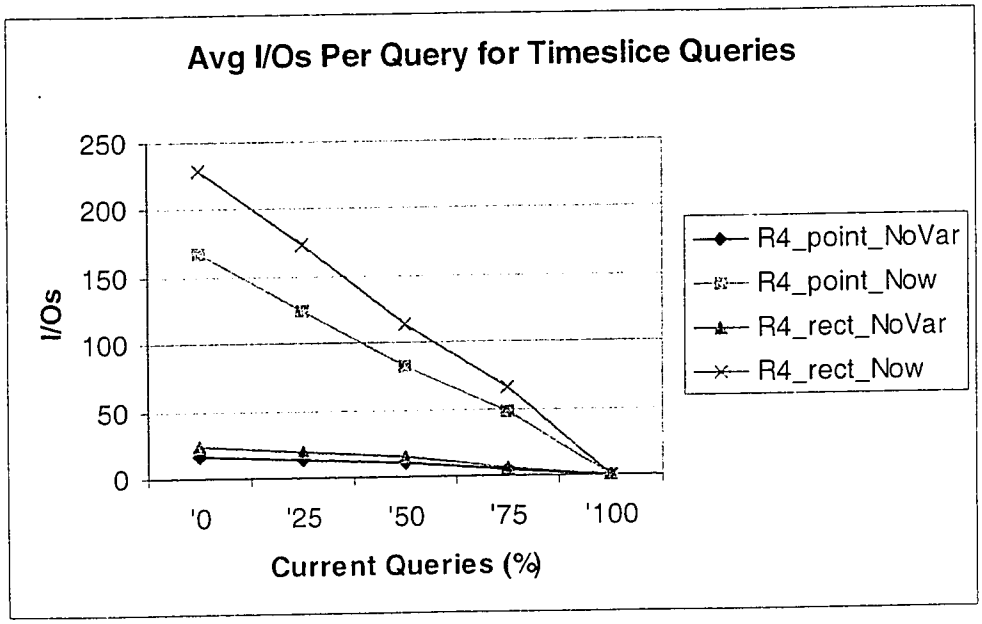


Figure 87 - Comparing R4 query performance for Timeslice queries

4.3.7.3 Varying Current Queries percentage for Point Queries

In this experiment, we study the performance of pure point queries. We vary the percentage of current queries (i.e. $TT_q^l = TT_q^u = CT$).

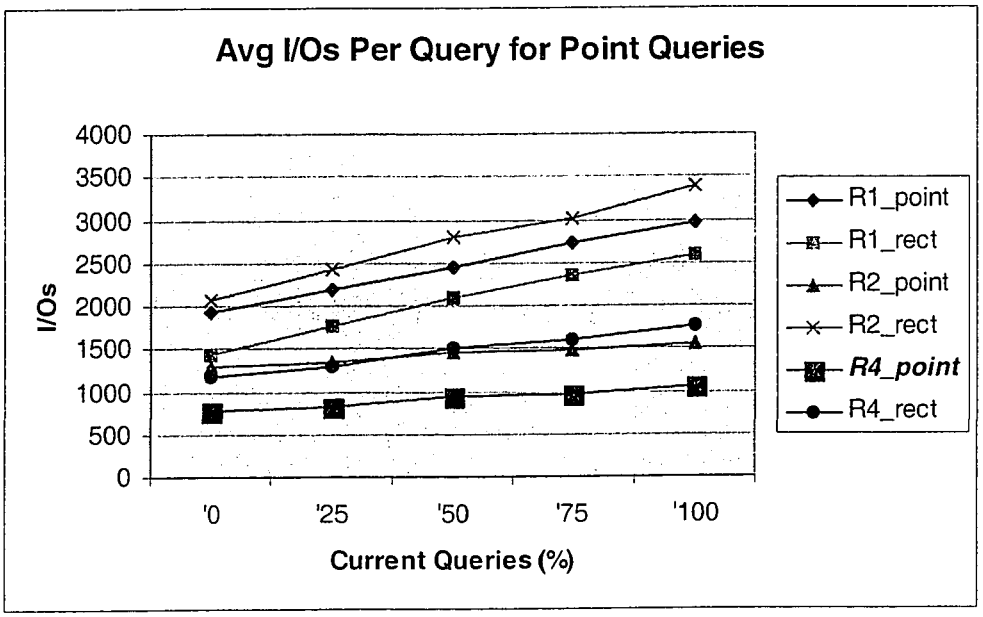


Figure 88 – Search performance of point queries for varying Qcur

The R4_point has the best point query search performance for all values of *Qcur*.

4.3.7.4 Varying Current Queries percentage for Mixed Queries

In this experiment, we study the performance of mixed queries where 50% are timeslice queries, 25% are range queries and the remaining 25% are point queries. We vary the percentage of current queries (i.e. $TT_q^u = CT$).

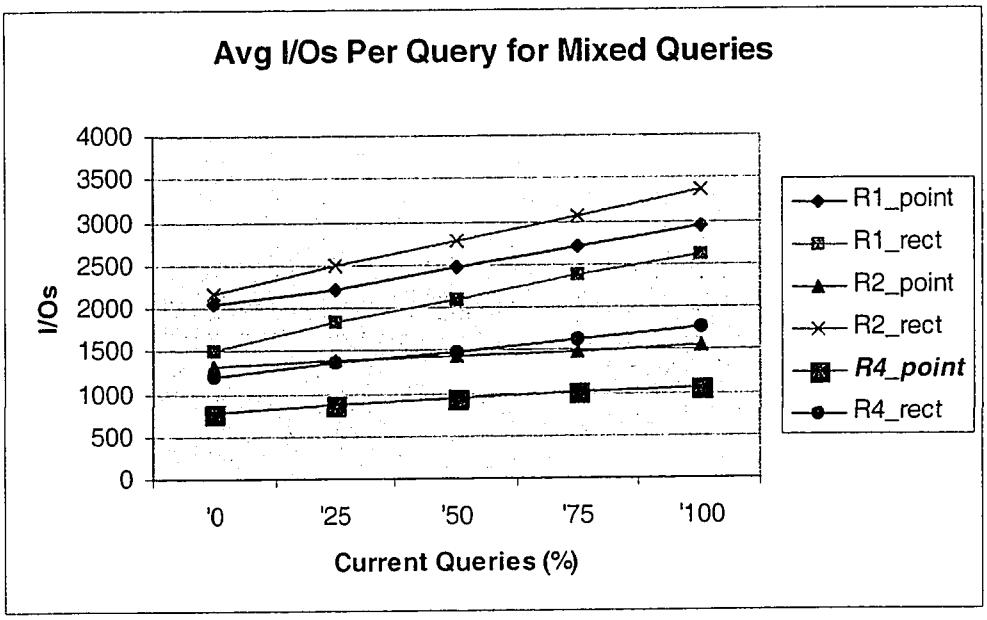


Figure 89 – Search performance of point queries for varying Qcur

The results are still consistent with the previous ones. The R4_point has the best search performance.

4.3.7.5 Varying Max query interval percentage for Range Queries

In this experiment, we study the performance of pure range queries. We vary the maximum length of the interval used in the query for both TT_q and VT_q. The larger ***Qmaxl***, the larger the size of search results. The results in Figure 90 clearly shows that the R4_point has the best search performance for all values of ***Qmaxl***. Moreover, the R4_point exhibits an almost steady performance for different values of ***Qmaxl***.

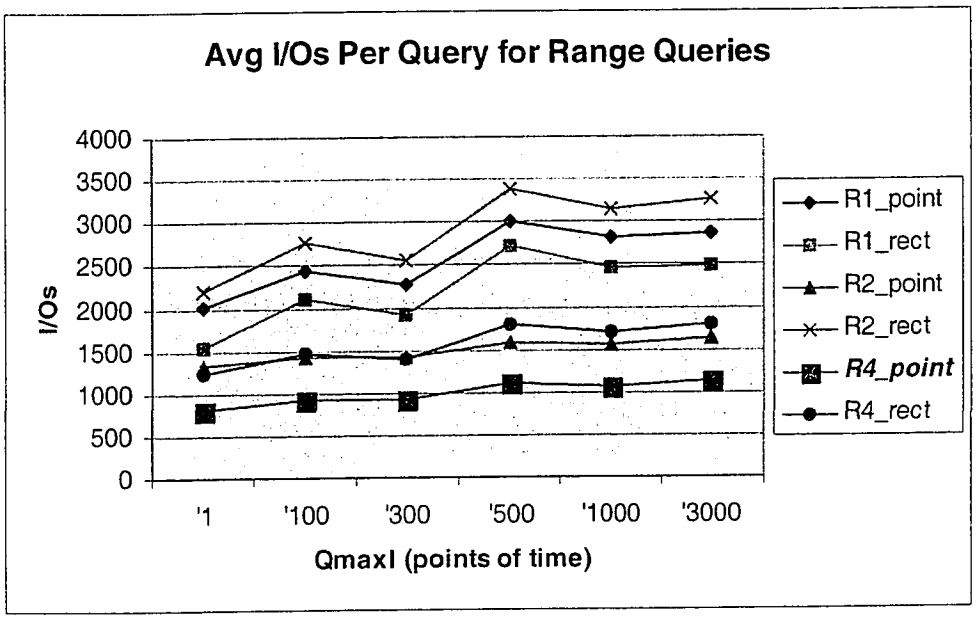


Figure 90 – Search performance of range queries for varying Qmaxl

4.3.7.6 Varying Max query interval percentage for Timeslice Queries

In this experiment, we study the performance of pure timeslice queries. We vary the maximum length of the interval used in the query for VT_q. The larger *Qmaxl*, the larger the size of search results. The results in Figure 91 are still consistent with the previous ones and proves that the R4_point has the best search performance.

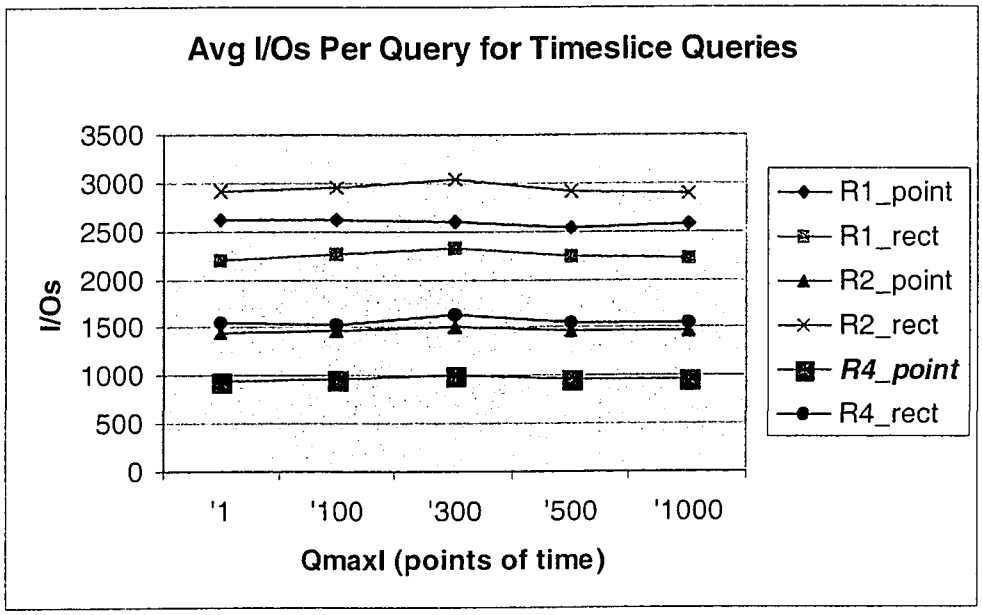


Figure 91 – Search performance of timeslice queries for varying Qmaxl

4.3.8 Experimenting with queries using different *UC* and *Now* semantics

In previous sections, we have used an implementation that searches all 4 subtrees in order to obtain the same search results as R1 and R2 indices. R1 and R2 indices have different semantics for *UC* and *Now* than R4 indices. R4 indices assume that *UC* and *Now* extends to the line VT = TT, while R1 and R2 assume that *UC* and *Now* extends to the maximum timestamps.

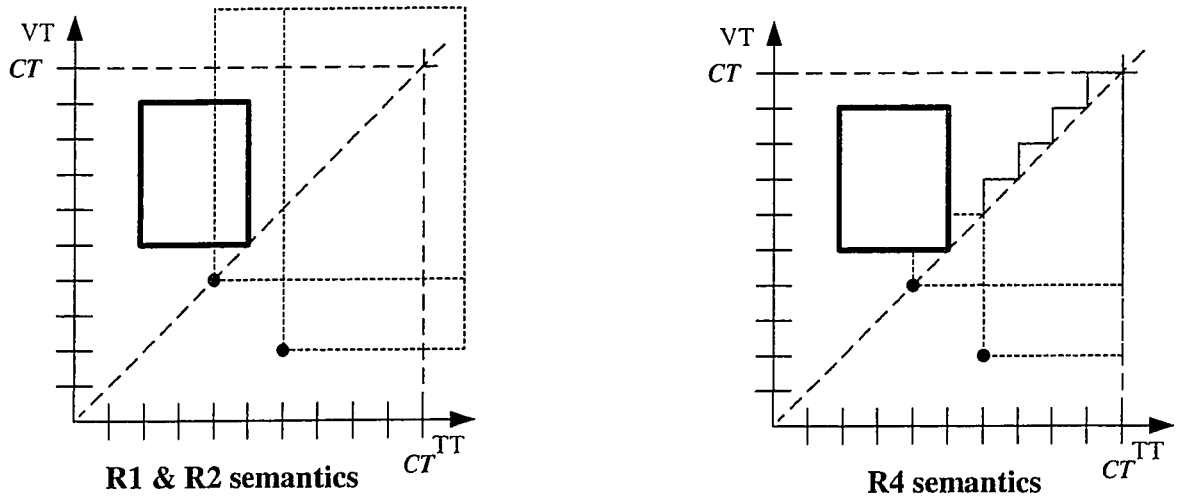


Figure 92 – Difference in semantics for *UC* and *Now*

In the following set of experiments, we have changed the implementation of the R4 indices to implement their own semantics for ***UC*** and ***Now*** (as described in the algorithm given in section 4.2.3.3 on page 87). This results in fewer search results in case of queries similar to the ones described in Figure 92.

4.3.8.1 Varying Current Queries percentage for Range Queries

In this experiment, we study the performance of pure range queries using the new semantic described above. We vary the percentage of queries whose $TT_q^u = CT$. Since 75% of the keys are current (i.e. $TT^e = UC$), then as ***Qcur*** increases, the size of search results increases which generally results in higher number of I/Os.

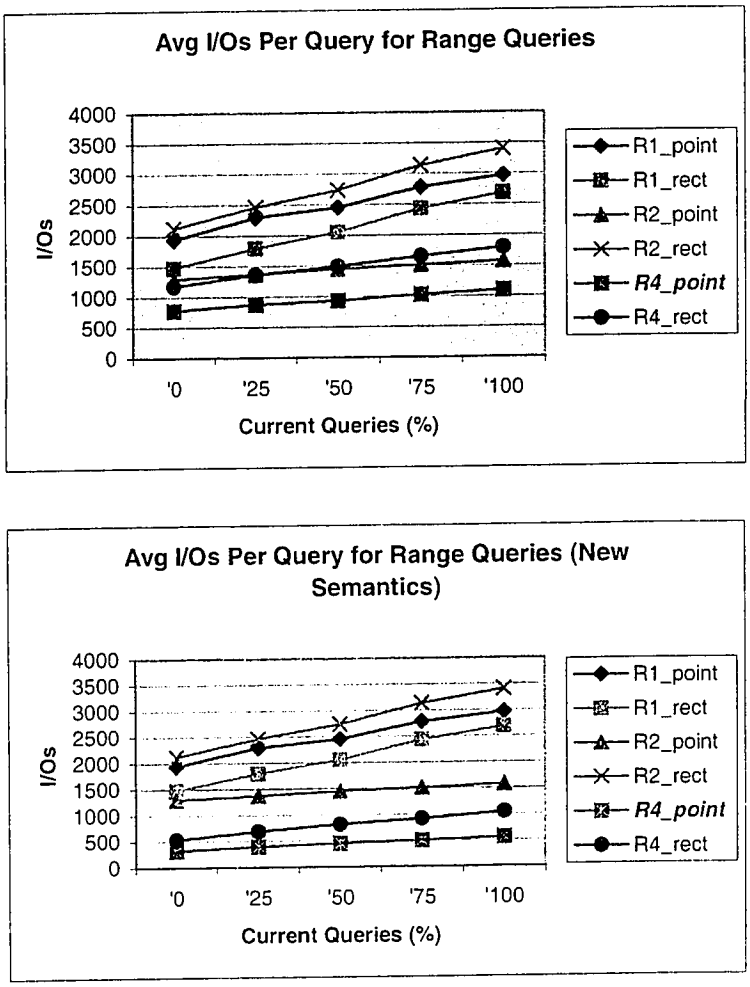


Figure 93– Search performance of range queries for varying Qcur (per query)

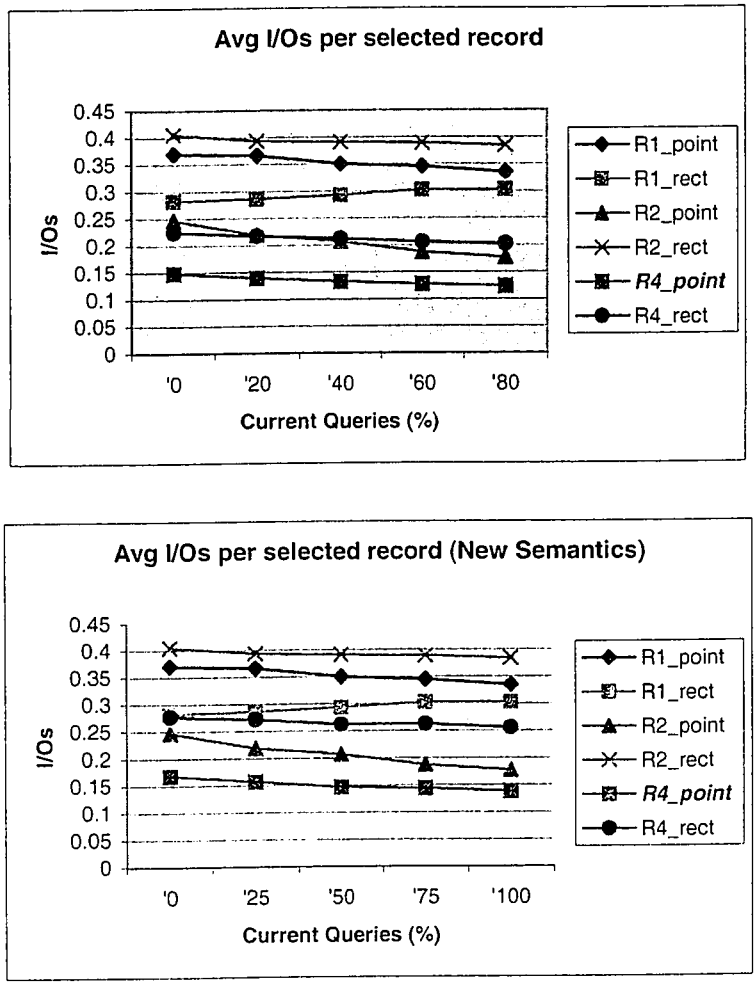


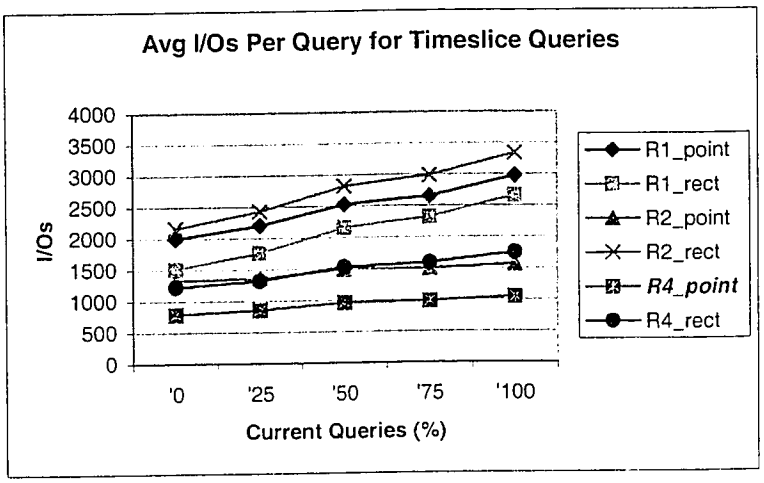
Figure 94 – Search performance of range queries for varying Qcur (per selected record)

In Figure 93 and Figure 94, we present the results using the old semantics on the left and the results using the new semantics on the right. The R4 indices now returns less results for the search and consequently consumes less I/Os. However their selectivity was relatively lowered (i.e. they consume more I/Os per selected record). Especially, the R4_rect version has used much more I/Os per selected record.

4.3.8.2 Varying Current Queries percentage for Timeslice Queries

In this experiment, we study the performance of pure time slice queries using the new semantic described above. We vary the percentage of queries whose $TT_q^l = TT_q^u = CT$. Since 75% of the keys are current (i.e. $TT^e = UC$), then as *Qcur* increases, the size of search results increases which generally results in higher number of I/Os.

The results are still consistent with the previous ones and the R4_point has the best search performance.



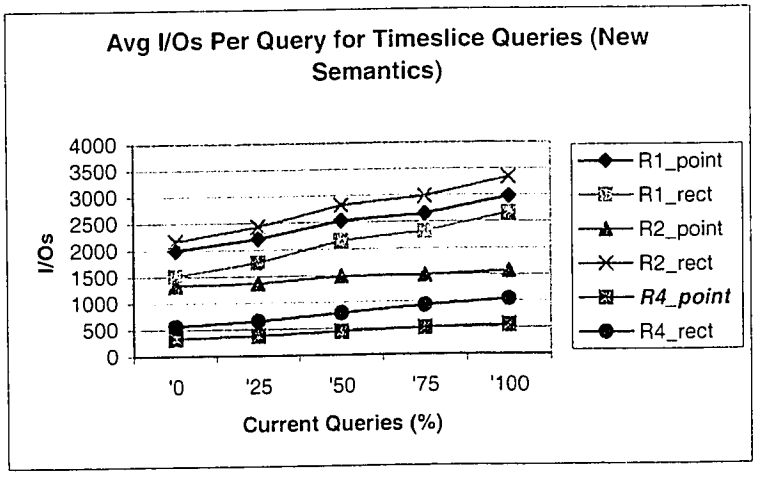
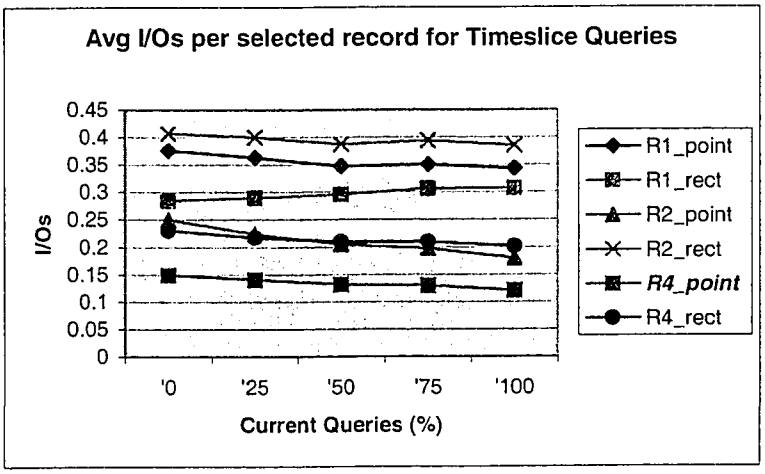


Figure 95– Search performance of Timeslice queries for varying Qcur (per query)



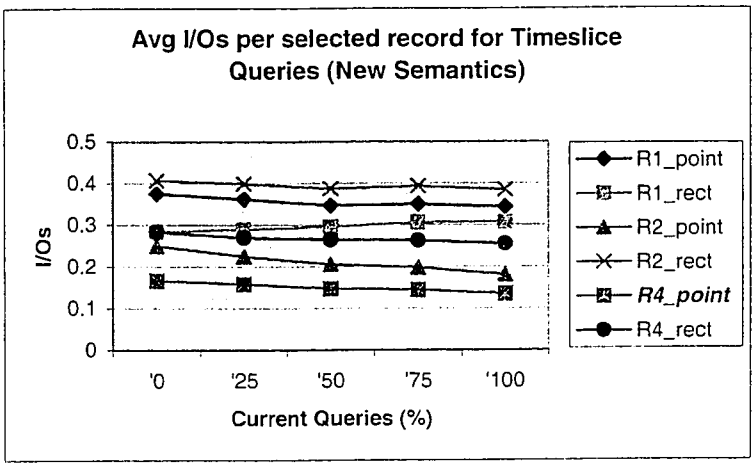
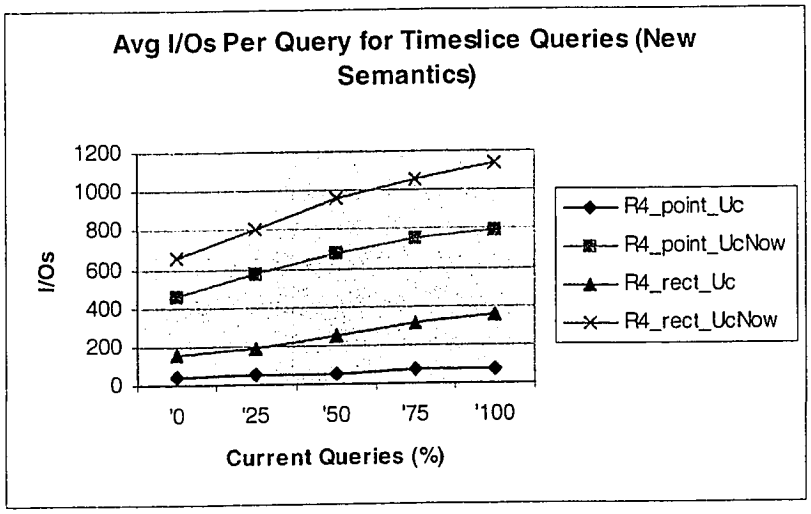


Figure 96 – Search performance of Timeslice queries for varying Qcur (per selected record)

In Figure 95 and Figure 96, we present the results using the old semantics on the left and the results using the new semantics on the right. The R4 indices now returns less results for the search and consequently consumes less I/Os. However their selectivity was relatively lowered (i.e. they consume more I/Os per selected record). Especially, the R4_rect version has used much more I/Os per selected record.

Analysis of the query performance for the two variants of the R4 trees shows that the point variant has much better performance than the rectangle variant.



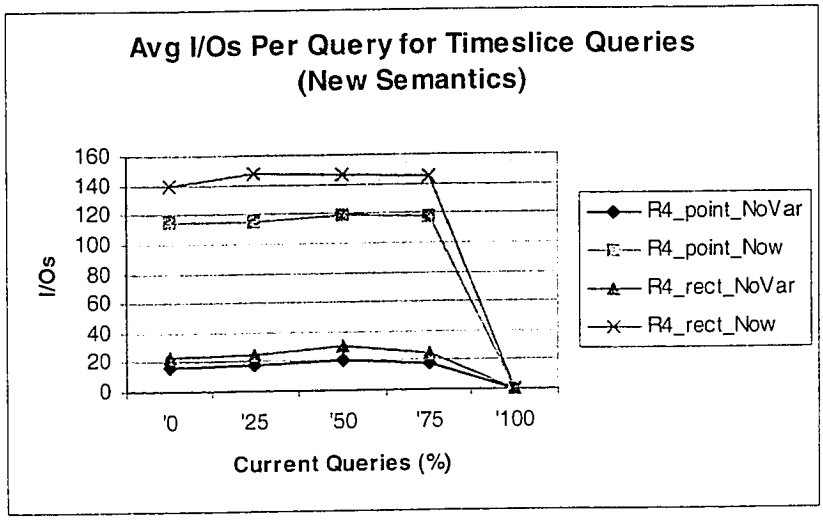


Figure 97 - Comparing R4 query performance for Timeslice queries (New Semantics)

To better understand these results, we study the distribution of the queries in the different subtrees using the new semantics.

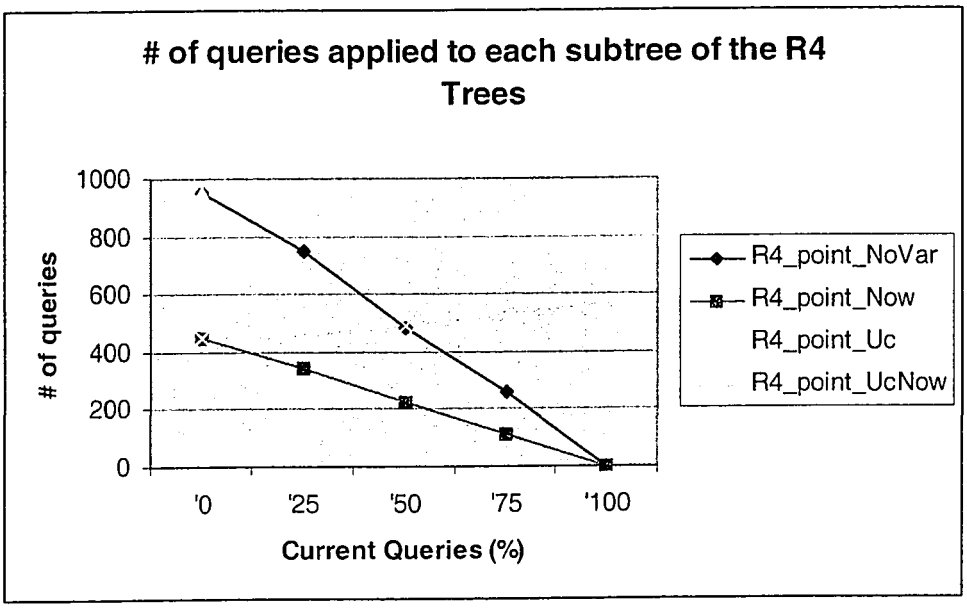


Figure 98 - distribution of queries applied to each subtree using the new semantics

Figure 98 shows that the UC subtree is always queried, while queries for the NoVar and the Now subtrees decreases as the % of current queries increases ands reaches zero when we have 100% current queries. The Uc and UcNow subtrees stores records with TT^c = *UC* and are constantly receiving queries even with different % of current queries.

4.3.8.3 Varying Current Queries percentage for Point Queries

In this experiment, we study the performance of pure point queries using the new semantic described above and varying ***Qcur*** from 0% to 100%. The R4_point has the best search performance.

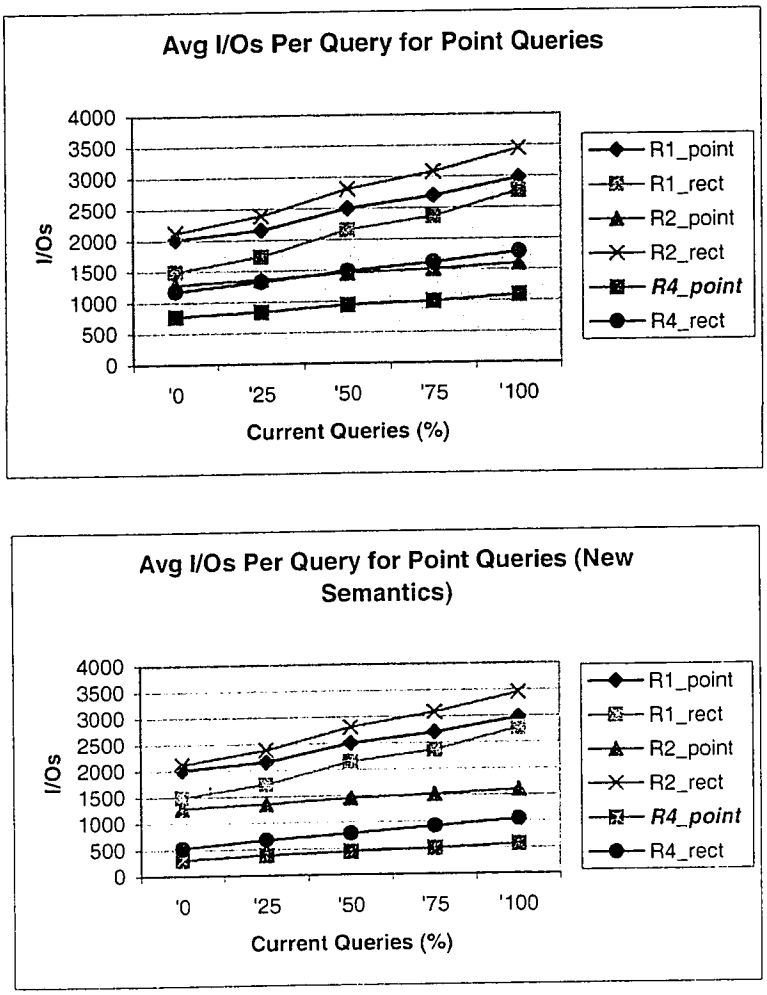


Figure 99– Search performance of Point queries for varying Qcur (per query)

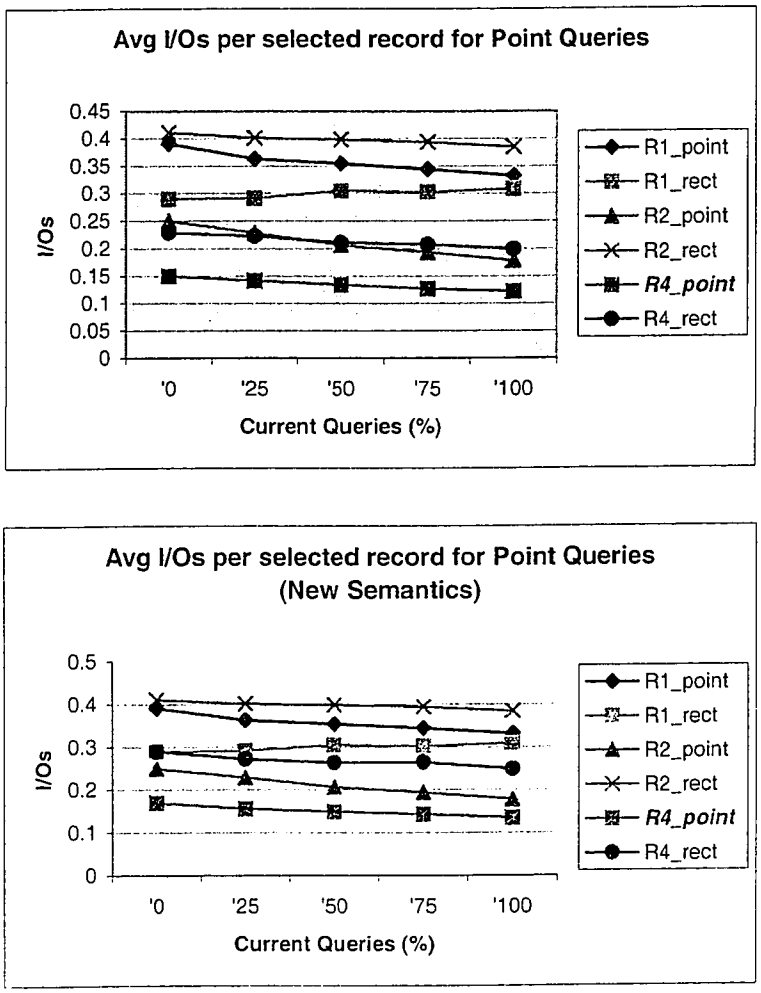
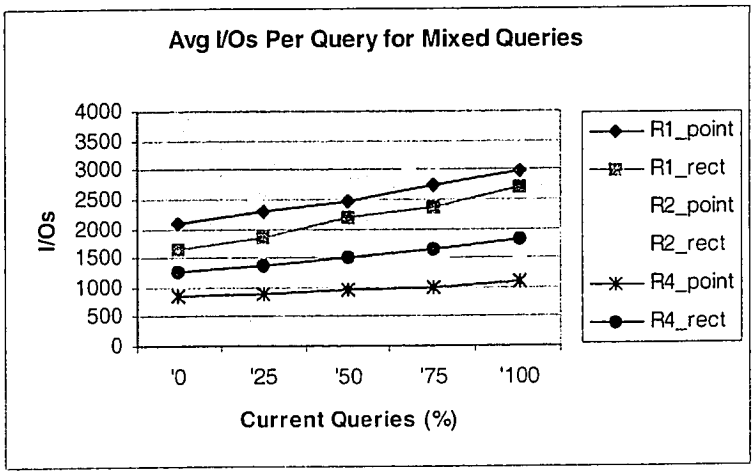


Figure 100 – Search performance of Point queries for varying Qcur (per selected record)

In Figure 99 and Figure 100, we present the results using the old semantics on the left and the results using the new semantics on the right. The R4 indices now returns less results for the search and consequently consumes less I/Os. However their selectivity was relatively lowered (i.e. they consume more I/Os per selected record). Especially, the R4_rect version has used much more I/Os per selected record.

4.3.8.4 Varying Current Queries percentage for Mixed Queries

In this experiment, we study the performance of mixed queries using the new semantic described above. The mixed queries contain 50% timeslice queries, 25% range queries and the remaining 25% are point queries. We vary the percentage of current queries (i.e. $TT_q^u = CT$).



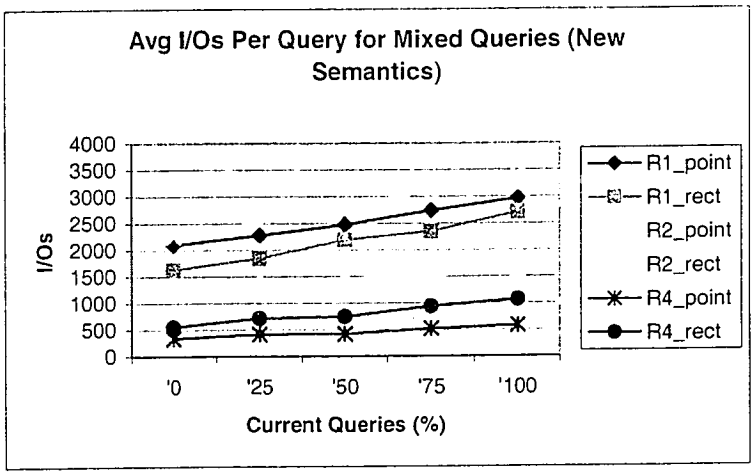


Figure 101– Search performance of Mixed queries for varying Qcur (per query)

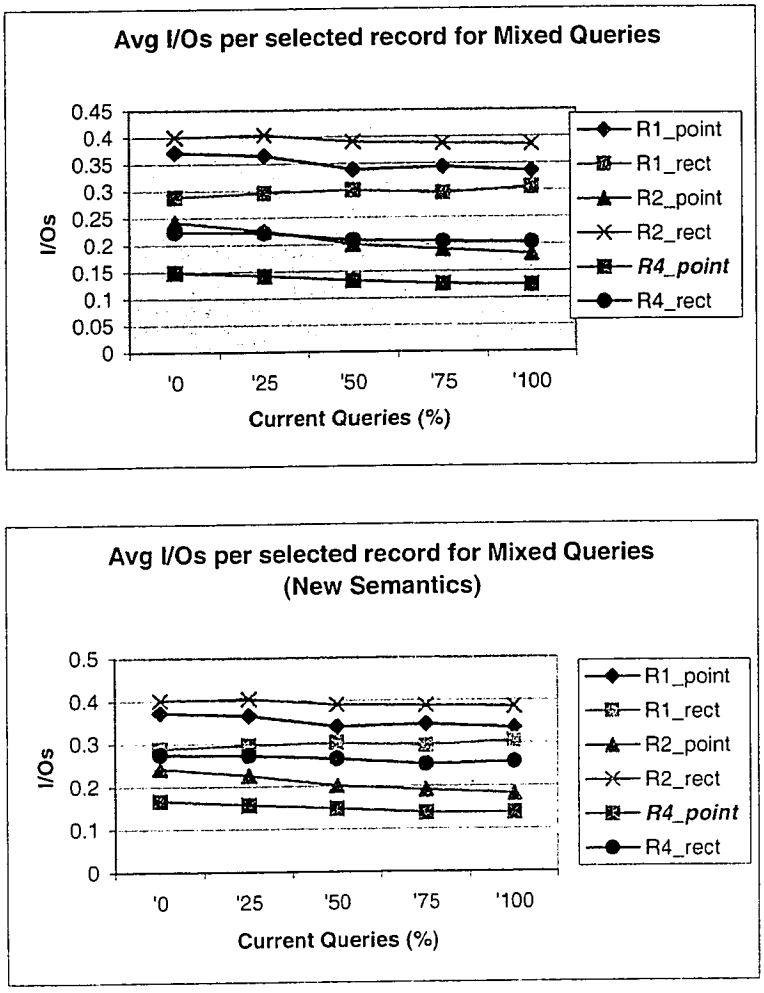
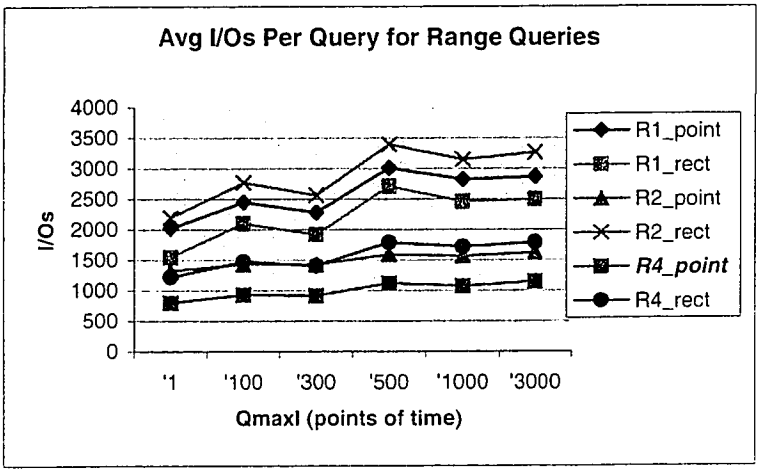


Figure 102 – Search performance of Mixed queries for varying Qcur (per selected record)

In Figure 101 and Figure 102, we present the results using the old semantics on the left and the results using the new semantics on the right. The R4 indices now returns less results for the search and consequently consumes less I/Os. However their selectivity was relatively lowered (i.e. they consume more I/Os per selected record). Especially, the R4_rect version has used much more I/Os per selected record.

4.3.8.5 Varying Max query interval percentage for Range Queries

In this experiment, we study the performance of pure range queries. We vary the maximum length of the interval used in the query for both TT_q and VT_q. The larger *Qmaxl*, the larger the size of search results.



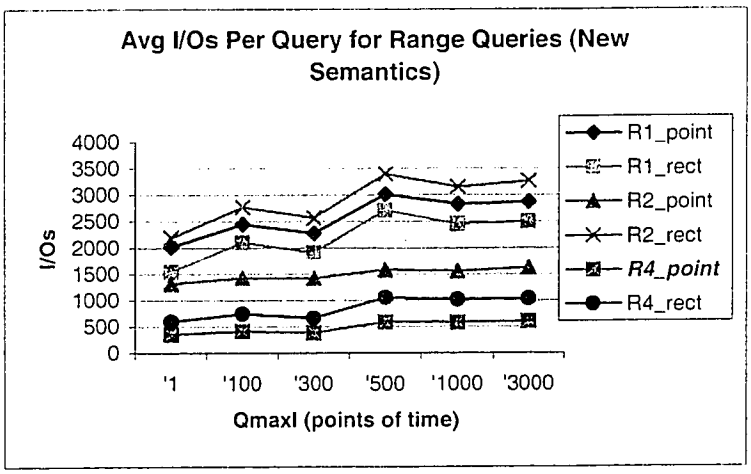


Figure 103– Search performance of range queries for varying Qmaxl (per query)

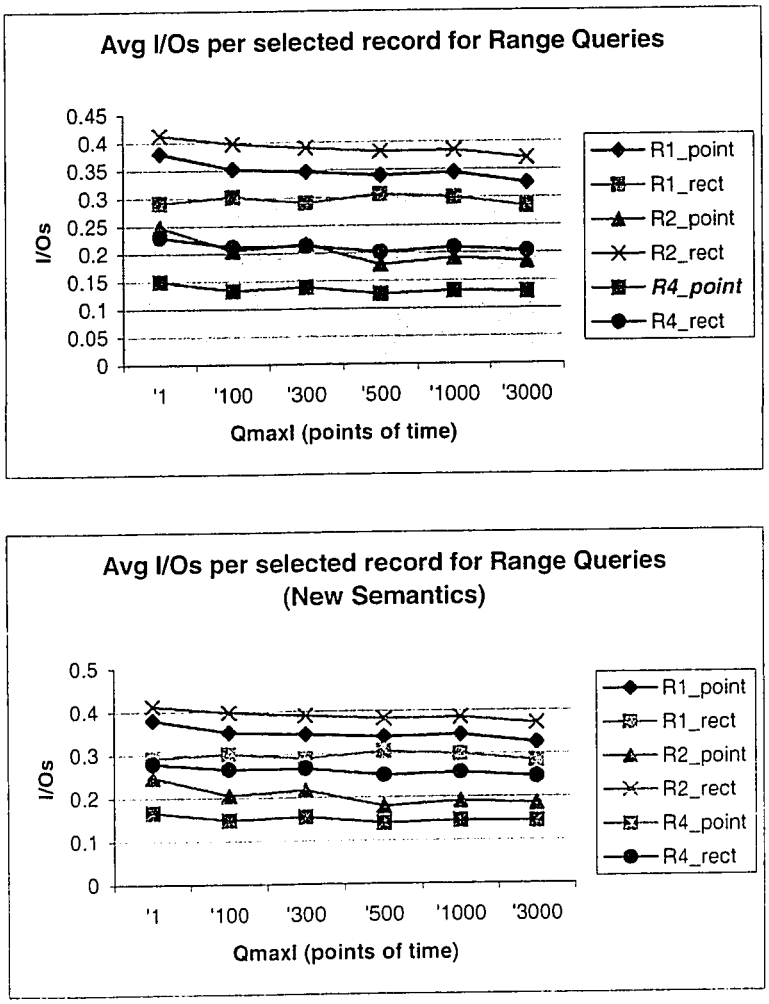
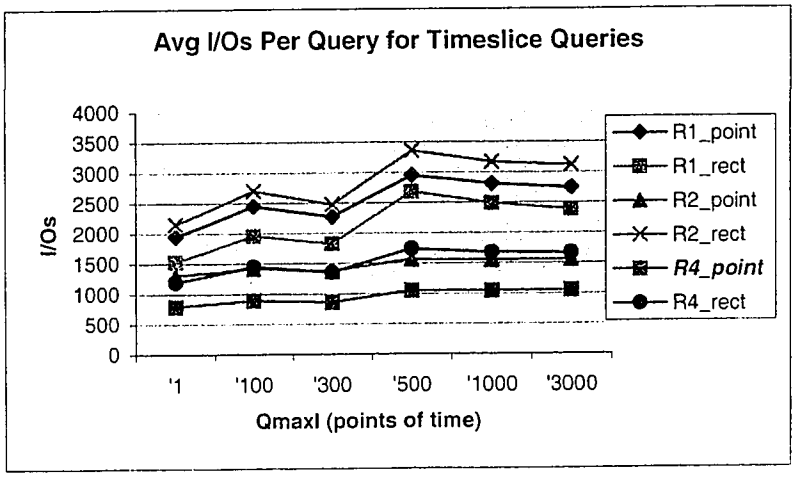


Figure 104 – Search performance of range queries for varying QmaxI (per selected query)

In Figure 103 and Figure 104, we present the results using the old semantics on the left and the results using the new semantics on the right. The R4 indices now returns less results for the search and consequently consumes less I/Os. However their selectivity was relatively lowered (i.e. they consume more I/Os per selected record). Especially, the R4_rect version has used much more I/Os per selected record.

4.3.8.6 Varying Max query interval percentage for Timeslice Queries

In this experiment, we study the performance of pure timeslice queries. We vary the maximum length of the interval used in the query for VT_q. The larger ***QmaxI***, the larger the size of search results.



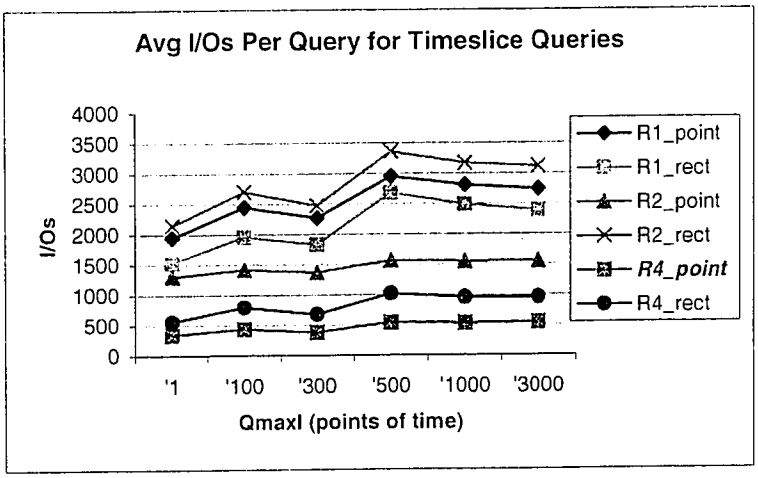


Figure 105– Search performance of timeslice queries for varying QmaxI (per query)

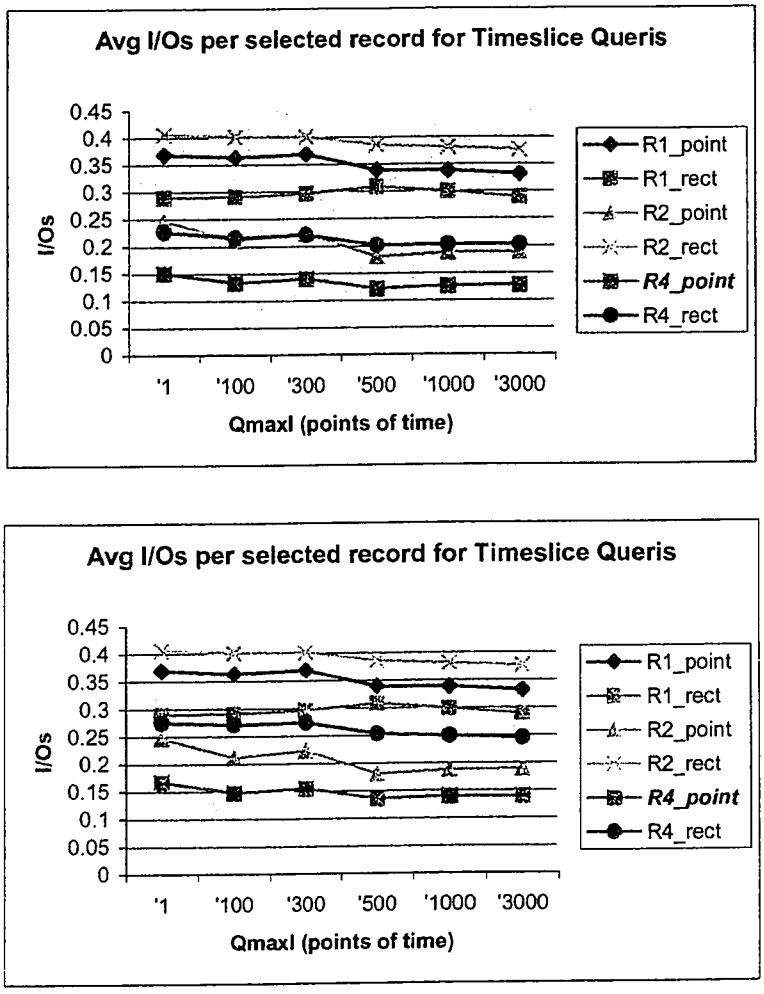


Figure 106 – Search performance of timeslice queries for varying QmaxI (per selected record)

In Figure 105 and Figure 106, we present the results using the old semantics on the left and the results using the new semantics on the right. The R4 indices return fewer results for the search and consequently consume less I/Os. The R4_rect version has used much more I/Os per selected record, which means that the selectivity was relatively lowered (i.e. they consume more I/Os per selected record).

4.3.9 Experimenting with varying 'Buffers Size'

In this experiment, we study the effect of varying the buffers size on the search and update performance. In general, enlarging the buffers size enhances the update performance for all indices. However, the enhancement is much more in case of the R4 indices. The R2_rect performance exceeds the R4 by 2 I/Os when the total buffer size is 80. This difference was reduced to only 1 I/O when we increased the total buffer size to 240.

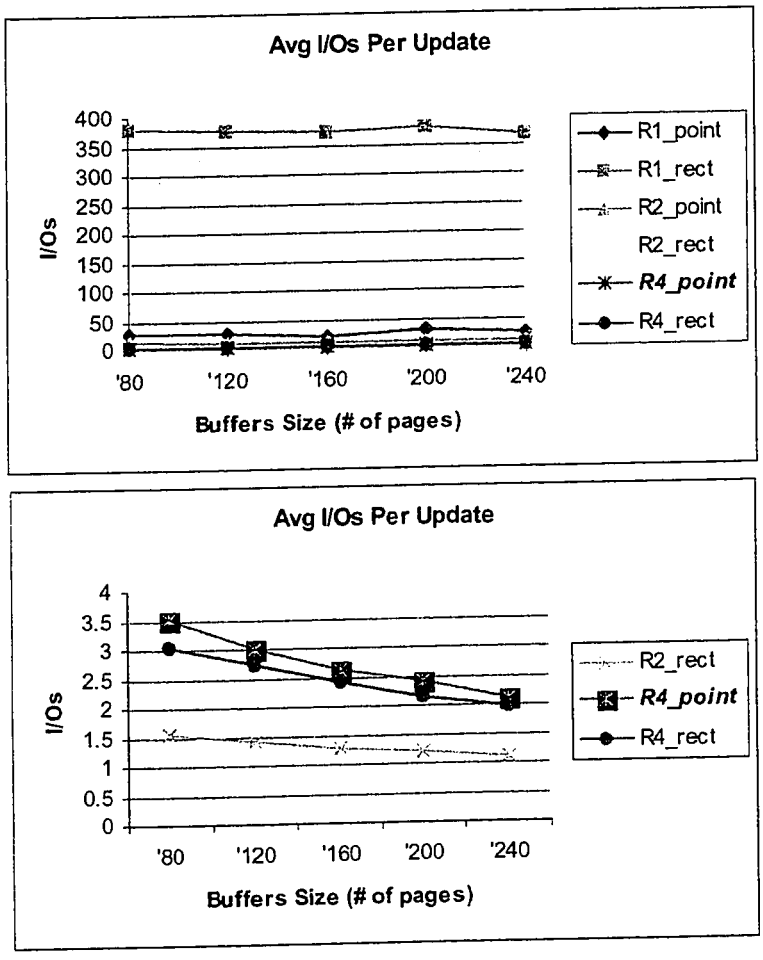


Figure 107 - Update performance for varying Buffers Size

Increasing the buffer size enhances the search performance as well but with the same rate for all indices.

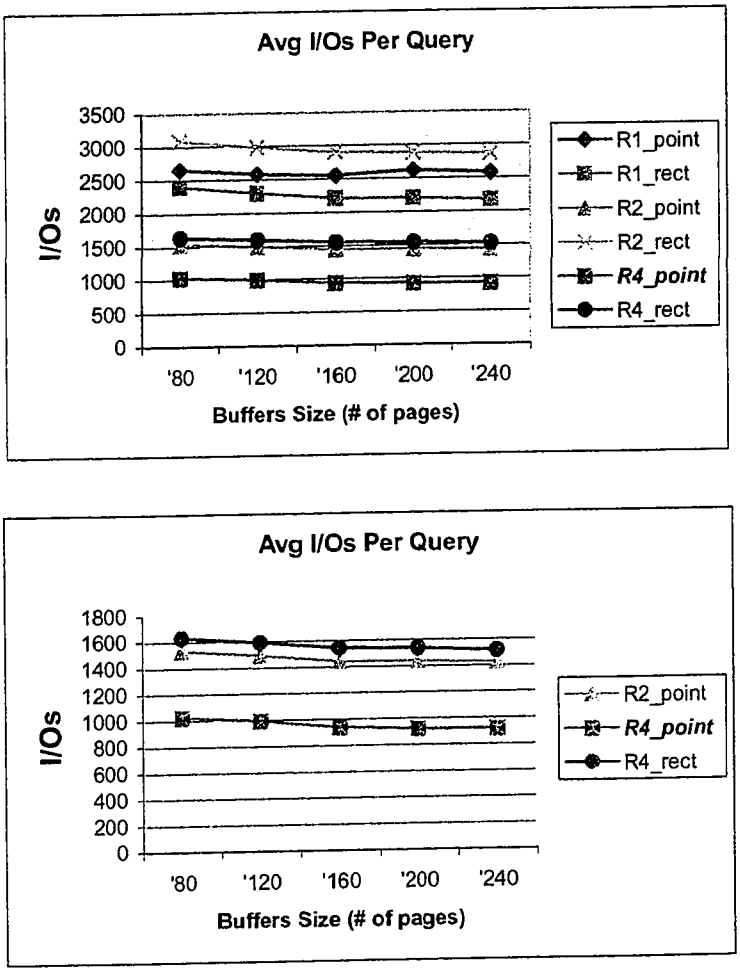


Figure 108 - Search performance for varying Buffers Size

4.3.10 Experimenting with varying 'Page Size'

In this experiment, we study the effect of varying the page size on the search and update performance. In general, enlarging the page size enhances the update performance for all indices. However, the enhancement is much more in case of the R4 indices. The R2_rect performance exceeds the R4 by 2 I/Os when the total page size is 1 KB. The R4_point

performance exceeds the R4 by 2 I/Os when the total page size is 1 KB. The R4_point almost reaches the same I/O level when we increased the page size to 5 KB. Note however, that increasing the page size is involving an increase in the buffer size. The buffer size is calculated by multiplying the buffers count by the page size.

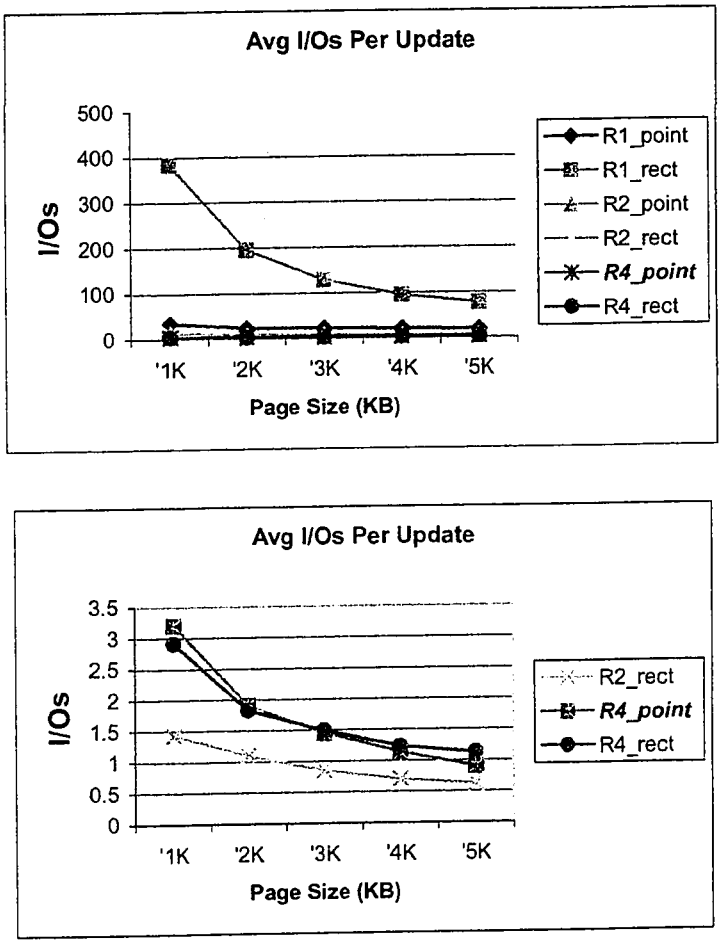


Figure 109 - Update performance for varying Page Size

Increasing the page size enhances the search performance as well but with the same rate for all indices. However, the R4_point always exhibits the best search performance.

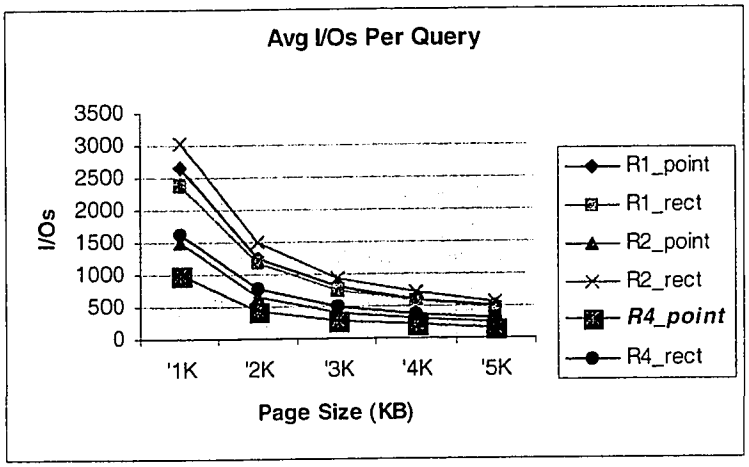


Figure 110 - Search performance for varying Page Size

Chapter 5 Conclusions and Future Work

5.1 Summary

Most applications require storing multiple versions of data and involve temporal semantics in their schema. This requires maintenance and querying of temporal relations. A Bitemporal DBMS will simplify the development and maintenance of such applications by moving temporal support from the application into the DBMS engine. The success of such Bitemporal DBMSs relies mainly on the availability of high performance indices that handles update and search operations efficiently.

5.2 Contributions Of the Thesis

In this thesis, we have studied the proposed bitemporal indices and especially those that support now-relative valid- and transaction-time. We have selected one that is implemented using on-the-shelf R*-tree and proposed an enhanced version of the R4 Tree. The enhanced version, which we call R4_point, uses point representation of the temporal attributes of the keys. This led to new mappings for the space for each subtree. We have then used Libgist v2.0 C++ library that allows using a generic framework to define indices. We had to first enhance the library to allow specification of the buffer size (cache) and the page size of the indices at run-time instead of hard coding them in the source. We also enhanced the replacement strategy, which selects the page that will be removed from the buffer when it is full and a new one needs to be brought in. Finally, we had to extend Libgist 2.0 to support now-relative temporal keys.

In order to evaluate the performance of the index, we have used a workload generator that simulates the lifetime of a temporal index. Many scripts had to be developed to run

different experiments and analyze results and present them in tabular format ready to be graphed. In each experiment, we compared the performance of six different indices: R1_rect, R1_point, R2_rect, R2_point, R4_rect, and finally our proposed index R4_point. All these indices are R*-tree based and differ in two aspects:

- The temporal attributes representation: rectangle versus point, and
- The number of subtrees used by the index: 1, 2, or 4 subtrees.

To fairly compare the performance of the six indices, the same workload was applied for each index to simulate its lifetime for a particular experiment. We also automatically compared the search results to guarantee that all indices are generating the same search results. It is worth notice here that we had to conduct two sets of experiments each set using a different implementation:

- In order to let all the indices generate the exact search results, we had to drop the search algorithm described in section 4.2.3.3 on page 87, and simply search all 4 subtrees.
- In the second implementation, we have implemented the search algorithm described in section 4.2.3.3 on page 87. This resulted that the R4 indices were producing a smaller search result.

All search results were consistent and clearly proved that the suggested temporal index R4_point has a much better search performance in different workloads. The update performance however is not the best, but is very close to the R4_rect. Only the R2_rect proved to have better update performance over the R4 indices.

R4_point index is suitable for most applications because they are generally search bounded and R4_point has been shown to provide the best search performance. However,

in applications that have more updates, it is not the best index structure. It is also known that for bulk loading of data (i.e. updates), it is advisable to temporarily drop the index anyway.

5.3 Future Work

Due to resource constraints, we had to limit the size of each experiment to 100,000 operations. But the size of a real temporal database is normally very large and it grows much faster than the size of a traditional (non-temporal) database. The main reason is that not only an insertion of a record but an update will also create a record in a temporal database. We believe that it is especially important for a temporal file structure to be adaptable to the fast growth of a database. The new index structure "TGF: Temporal Grid File" proposed in [LT98] seems very promising to be used instead of the R*-trees. However, TGF as described in [LT98] is supporting only one time dimension. We would like to explore the possibility of extending this structure to handle bitemporal data and compare its performance with the results obtained in our thesis.

References

[A84] J. Allen. Towards a General Theory of Action and Time. Artificial Intelligence, Vol. 23(2), pp. 123-154, 1984.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley 1995.

[B+97] Elisa Bertino, Beng Chin Ooi, Ron Sacks-Davis, Kian-Lee Tan, Justin Zobel, Boris Shidlovsky, and Barbara Catania. Indexing Techniques For Advanced Database Systems. Kluwer Academic Publishers, pp. 113-149, 1997.

[B93] B. Becker et al. On optimal multiversion access structures. In Proc. of Symposium on Large SpatialDatabases, pp. 123–141, June 1993.

[BFG97] E. Bertino, E. Ferrari, and G. Guerrini. T_Chimera: A Temporal object-oriented data model. International Journal on Theory and Practice of Object Systems (TAPOS), 3(2), pp. 103-125, 1997.

[BJSS00] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas, "Light-Weight Indexing of General Bitemporal Data," in Proceedings of 12th International Conference on Scientific and Statistical Database Management SSDBM'2000 conference, 2000.

[BJSS98] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas, "R-tree-based Indexing of Now-Relative Bitemporal Data," in Proceedings of the 24th International Conference on Very Large Databases, New York City, NY, 1998.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger,. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. Proceedings of ACM SIGMOD, pp. 322-331, 1990.

[BWJ95] C. Bettini, X. Sean Wang, and S. Jajodia. Temporal Semantics assumptions and their use in databases. IEEE Transactions on Knowledge and Data Engineering. To appear. (Revised version of ISSE-TR-95-105, ISSE Technical Report, George Mason University).

[C+98] Jensen Curtis Dyreson et al. The Consensus Glossary of Temporal Database Concepts. In Temporal Databases: Research and Practice, O. Etzion, S. Jajodia, and S. Sripada (eds), Springer-Verlag, pp. 367–405, 1998.

[C99] J. Celko. Joe Celko's Data and Databases: Concepts in Practice. Morgan Kaufman Publishers 1999.

[CC97] L. Chittaro and C. Combi. Temporal indetermancy in deductive databases: an approach based on the event calculus. In the 2nd International Workshop on Active, Real-Time and Temporal Database Systems (ARTDB-97), September 1997.

[CLI97] J. Clifford et al. On the Semantics of "NOW" in Databases. ACM TODS, 22(2), pp.171-214, 1997.

[CT95] J. Clifford and A. Tuzhilin (eds.). Recent Advances in Temporal Databases: Proceedings of the International Workshop on Temporal Databases. Workshops in Computing Series. Springer-Verlag 1995.

[DRI89] J. R. Driscoll et al. Making Data Structures Persistent. Journal of Computer and System Sciences, 38(1), pp. 86-124, 1989.

[DS93] C. E. Dyreson, and R. T. Snodgrass. "Timestamp Semantics and Representation." Information Systems, 18, No. 3 (1993), pp. 143–166.

[DW90] C. J. Date, and C. J. White. "A Guide to DB2." Reading, MA: Addison-Wesley, 1990. Vol. 1, 3rd edition.

[E83] H. Edelsbrunner. A new approach to rectangle intersections: Part I. International Journal of Computer Mathematics, 13(3-4), pp. 209-219, 1983.

[EJS98] O. Etzion, S. Jajodia, and S. Sripada (eds.). Temporal Databases: Research and Practice. LNSC 1399, Springer-Verlag, pp. 367–405, 1998.

[EN94] R. Elmasri and S. Navathe. Fundamentals of Database Systems. Benjamin/Cummings 1994.

[G84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. Proceedings of ACM SIGMOD, pp. 47-57, 1984.

[GOS97] I. A. Goralwalla, M. T. Ozsu, and D. Szafron. A framework for temporal data models: Exploiting object-oriented technology. In Proc. Of the 13th International Conference and Exhibitio on Technology of Object-Oriented Languages and Systems (TOOLS'97) USA, July 1997.

[GS93] H. Gunadhi, and A. Segev. Efficient Indexing methods for temporal relation. IEEE Transactions on Knowledge and Data Engineering, 5(3), pp. 496-509, 1993.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. Proc. 21st Int'l Conf. on Very Large Data Bases, Zürich, pp. 562-573, September 1995.

[JS96] Christian S. Jensen and Richard T. Snodgrass. Semantics of time-varying information. Information Systems, 21(4), pp. 311-352, 1996.

[KF94] I. Kamel, and C. Faloutsos. Hilbert R-tree: An improved R-Tree using Fractals. Proceedings of VLDB, pp. 500-509, 1994.

[KTF95] A. Kumar, V.J. Tsotras, and C. Faloutsos. Access Methods for Bitemporal Databases. In Recent Advances in Temporal Databases, J. Clifford, and A. Tuzhilin (eds), Springer-Velag, pp. 235-254, 1995.

[KTF98] A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. IEEE Transactions on Knowledge and Data Engineering, 10 (1), pp. 1-20, 1995.

[LL98] S. T. Leutenegger, and M. A. Lopez. The Effect of Buffering on the Performance of R-Trees. Proceedings of ICDE, pp. 164-171, 1998.

[LL99] M. Levene and G. Loizou. A Guided Tour of Relational Databases and Beyond. Springer-Verlag 1999.

[LT98] Chiang Lee and Te-Ming Tseng. Temporal Grid File: A file structure for interval data. Data and Knowledge Engineering, 26(1), pp. 71-97, 1998.

[M96] J. Melton (editor). SQL/Temporal. July 1996. (ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-0012.)

[MS93] L. Melton, and A. R. Simon. "Understanding the New SQL: A Complete Guide." San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993.

[NDE96] M. A. Nascimento, M. H. Dunham, and R. Elmasri. M-IVTT: An index for bitemporal databases. In Proc. of the 7th Intl. Conf. on Databases and Expert Systems Applications, pp. 779–790, September 1996.

[RL85] N. Roussopoulos, and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In Proc. ACM SIGMOD International Conference on Management of Data, pp. 17-31, 1985.

[S00] R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann Publishers 2000.

[S87] R. T. Snodgrass. The Temporal Query Language TQuel. ACM TODS, 12(2), pp. 247–298, 1987.

[S93] R. T. Snodgrass (ed.). Proceedings of the International Workshop on an Infrastructure for Temporal Databases. Arlington, TX, June 1993.

[S94] B. Salzberg. On indexing spatial and temporal data. Information Systems, 19(6), pp. 447-465, 1994.

[S97] N. L. Sarda. Modeling Valid Time: an efficient representation. In Proc. Of the International Conference on Database Systems for Advanced Applications (DASFAA'97), April 1997.

[SA85] R. T. Snodgrass, and I. Ahn. "A Taxonomy of Time in Databases," in Proceed-ings of ACM SIGMOD International Conference on Management of Data.Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 236–246.

[SA86] R. Snodgrass and I. Ahn. Temporal Databases. IEEE Computer Management, September, pp. 35-41, 1986.

[SA86] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35--42, 1986.

[SAM90] H. Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley, 1990.

[SJ99] S. Saltenis, and C. S. Jensen. "R-Tree Based Indexing of General Spatio-Temporal Data," TimeCenter Technical Report TR-45, December 1999.

[SKH99] M. Shah, M. Kornacker, and J. Hellerstein. Amdb: A Visual Access Method Design Tool. Proc. User Interfaces for Data Intensive Systems (UIDIS), September 1999.

[SKKM94] R. Snodgrass, K. Kulkarni, H. Kucera and N. Mattos. Proposal for a new SQL Part-Temporal. Proc. ISO/IEC JTC1/SC21/WG3 DBL RIO-75, X3H2-94-481, 1994.

[SOL94] H. Shen, B. C. Ooi, and H. Lu. The TP-Index: A dynamic and efficient indexing mechanism for temporal databases. In Proc. 10th International Conference on Data Engineering, pp. 274-281, 1994.

[SRF87] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multidimensional Objects. Proceedings of VLLDB, pp. 507-518, 1987.

[ST99] B. Salzberg and V. J. Tsotras. A Comparison of Access Methods for Time-Evolving Data. ACM Computing Surveys, 31(2), pp. 158-221, 1999.

[T+94] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass (eds.). Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings Publishers 1994.

[T97] Adullah Uz Tansel. Temporal relational data model. IEEE Transactions on Knowledge and Data Engineering, 9(3), pp. 464-479, May 1997.

[TSJ98] V. J. Tsotras, R. T. Snodgrass, and C. S. Jensen. An Extensible Notation for Spatiotemporal Index Queries. ACM SIGMOD Record, 27(1), pp. 47-53, March 1998.

[TSN99] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. Proc. 6th International Symposium on Spatial Databases, pp. 147-164, 1999.

[TW99] V. J. Tsotras and X. S. Wang. Temporal Databases. Encyclopedia of Electrical and Electronics Engineering, John Wiley and Sons, 1999.

[VV97] P.J. Varman and R.M. Verma. An efficient multiversion access structure. IEEE Transactions on Knowledge and Data Engineering, 9(3), pp. 391–409, 1997.

[WJL91] G. Wiederhold, S. Jajodia and W. Litwin. "Dealing with Granularity of Time in Temporal Databases," in Proc. 3rd Nordic Conf. on Advanced Information Systems Engineering. Trondheim, Norway: May 1991.

[WJL93] G. Wiederhold, S. Jajodia and W. Litwin. "Integrating Temporal Data in a Heterogeneous Environment," in Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings, 1993. Chap. 22. pp. 563–579.

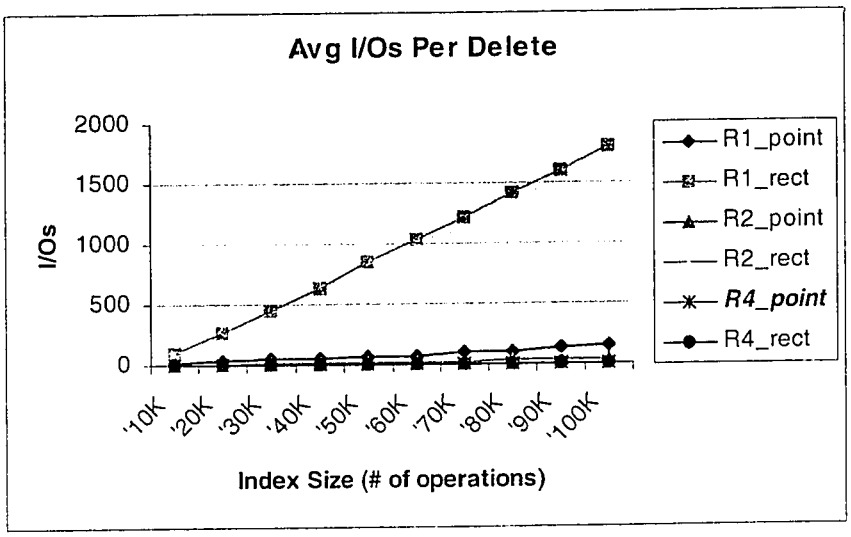
[WJW98] Yu Wu, Sushil Jajodia, and X. Sean Wang. Temporal Database Bibliography Update. Temporal Databases: Research and Practice. Lecture Notes in Computer Science, 1399, pp. 338-366, 1998.

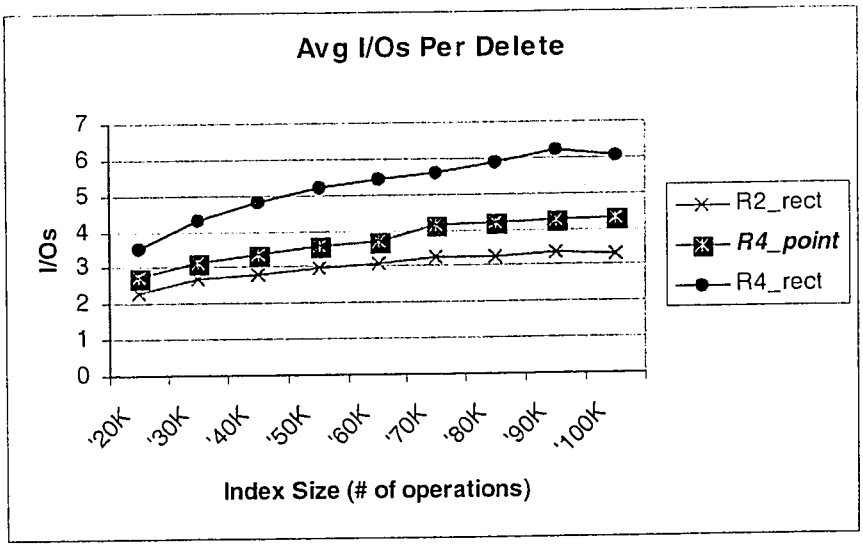
[Z+97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. Advanced Database Systems. Morgan Kaufmann Publishers 1997.

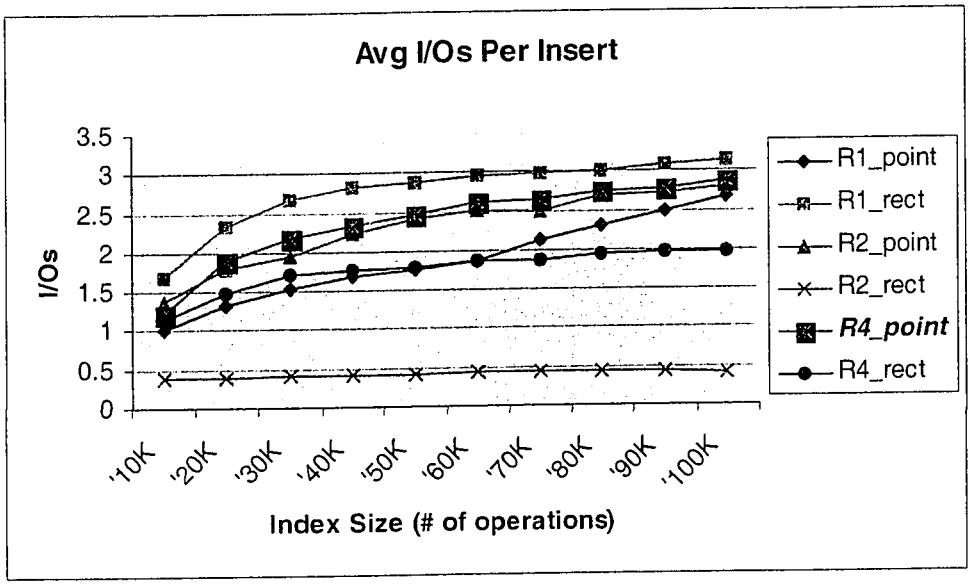
Appendix A

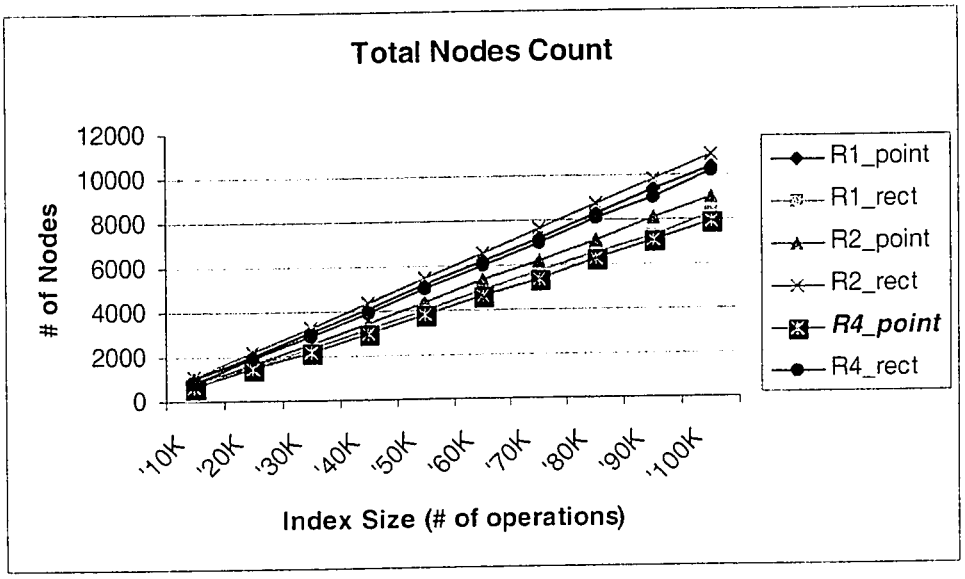
A.1 Additional results for the Size experiment

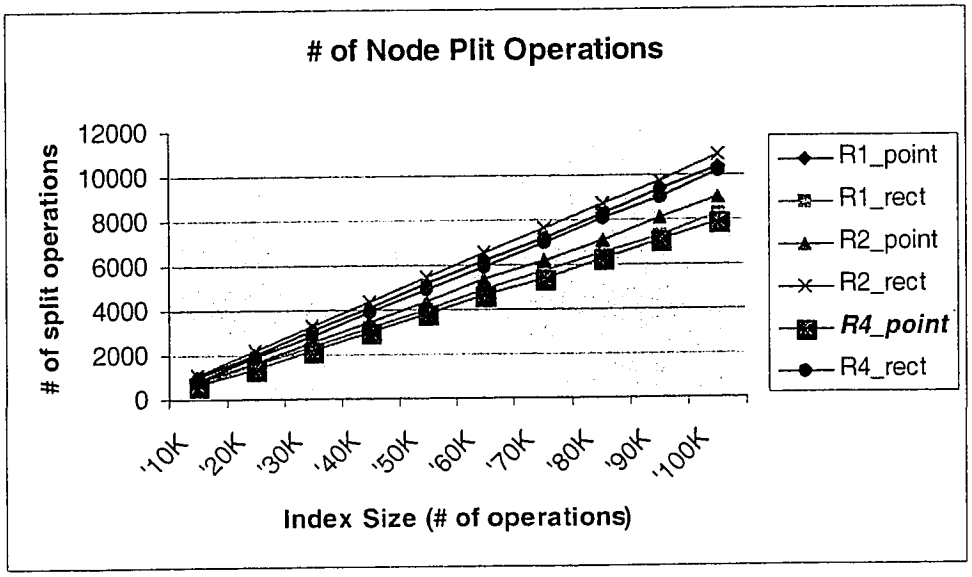
Here are some additional results for the experiment described in section 4.3.2 on page 89.



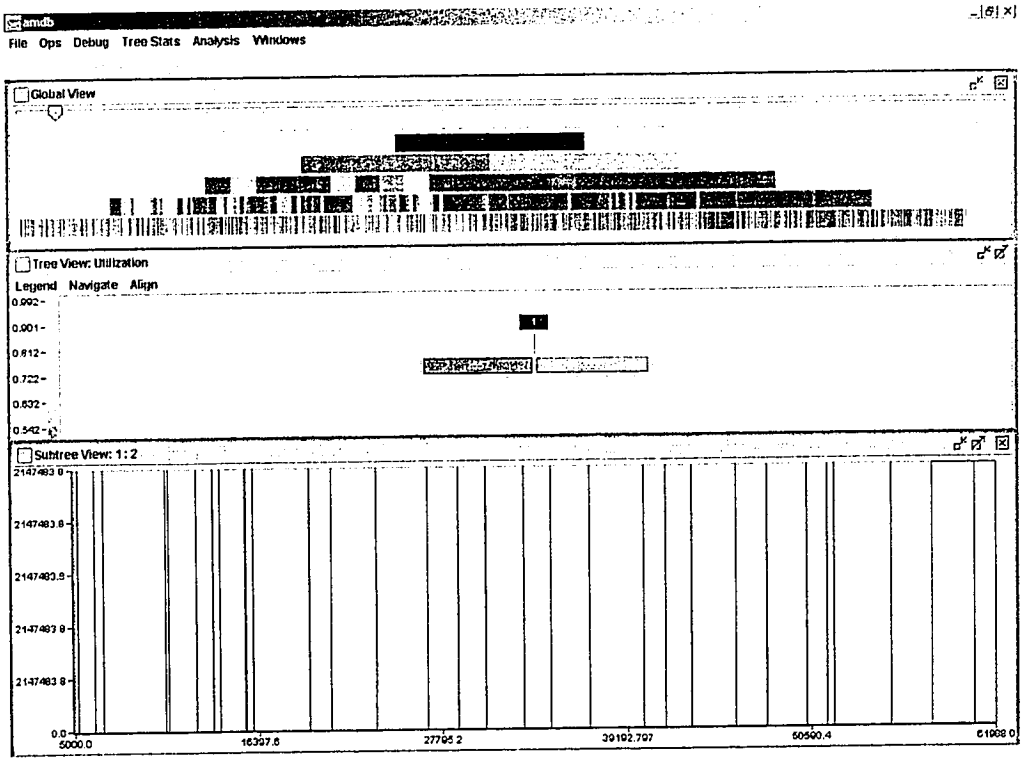



Avg I/Os Per Insert
I/Os
3.5
3
2.5
2
1.5
1
0.5
0
'10K
'20K
'30K
'40K
'50K
'60K
'70K
'80K
'90K
'100K
Index Size (# of operations)
R1_point
R1_rect
R2_point
R2_rect
R4_point
R4_rect

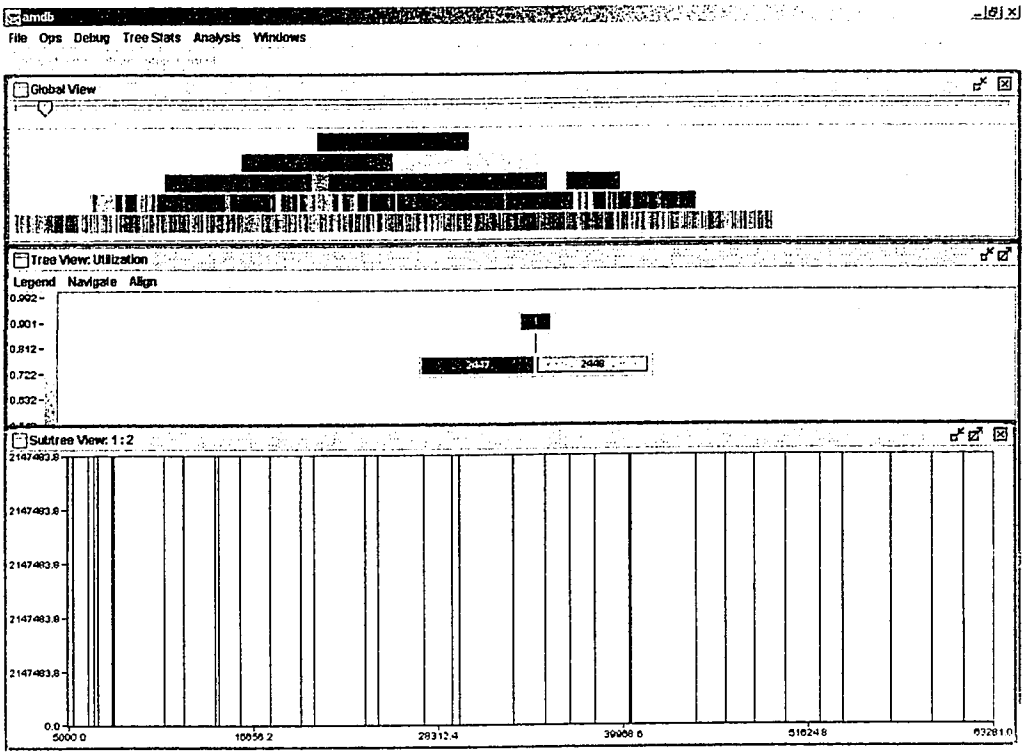

Total Nodes Count
# of Nodes
12000
10000
8000
6000
4000
2000
0
'10K
'20K
'30K
'40K
'50K
'60K
'70K
'80K
'90K
'100K
Index Size (# of operations)
R1_point
R1_rect
R2_point
R2_rect
R4_point
R4_rect



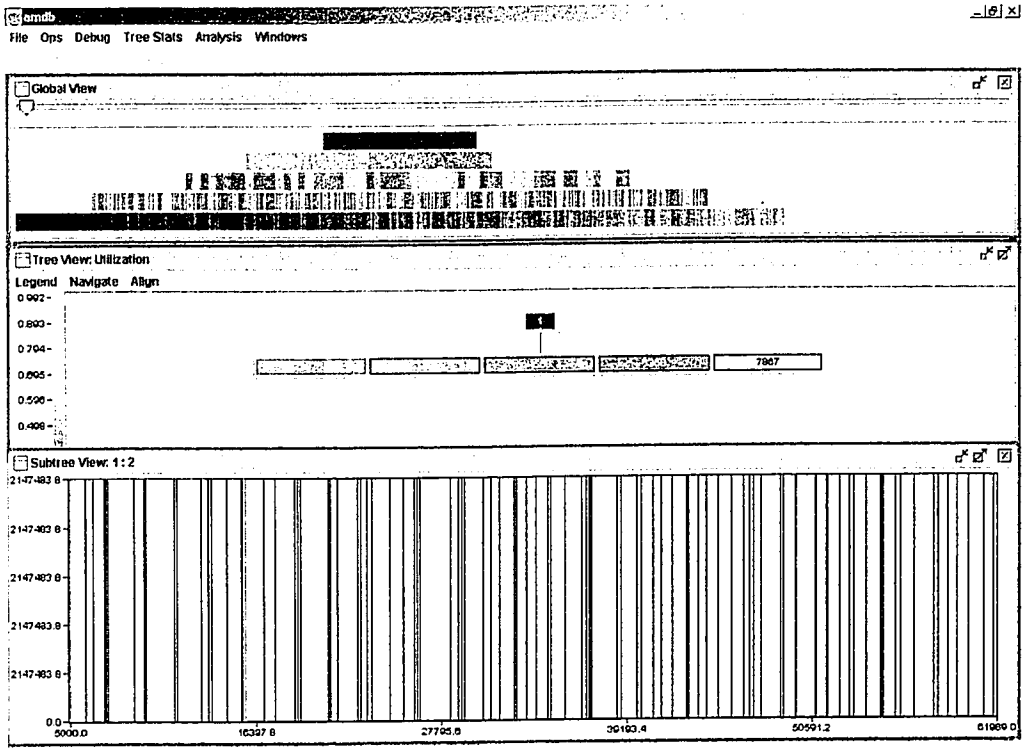
A.2 VL_50_R2_rect_NoVar



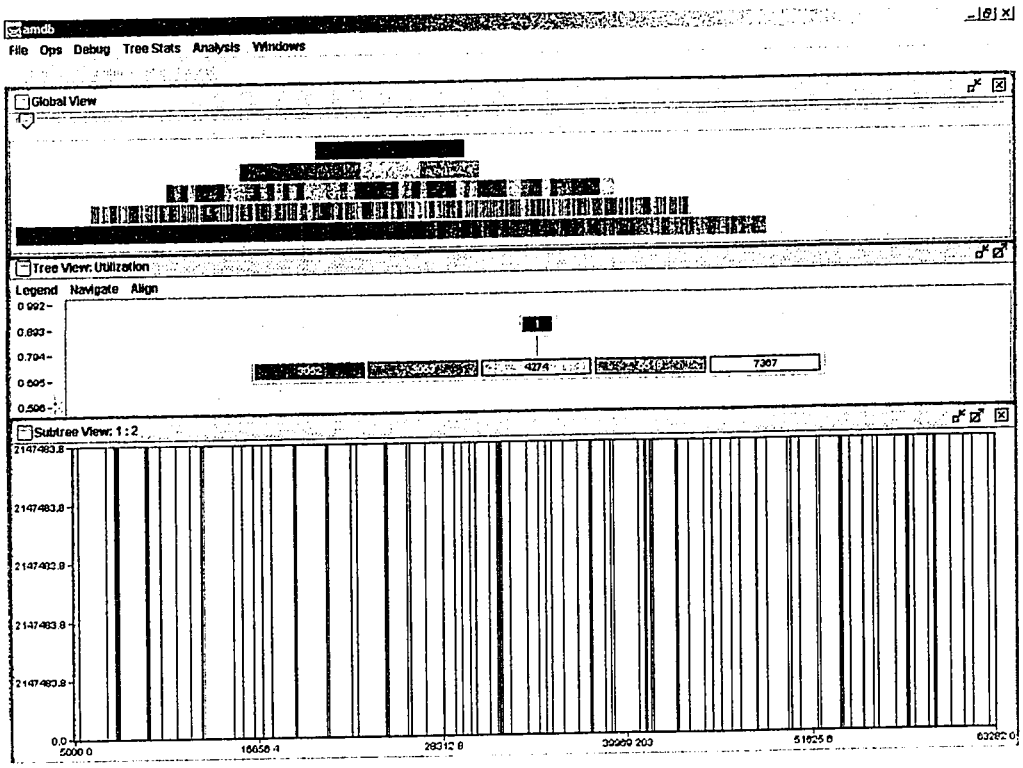
A.3 VL_5000_R2_rect_NoVar



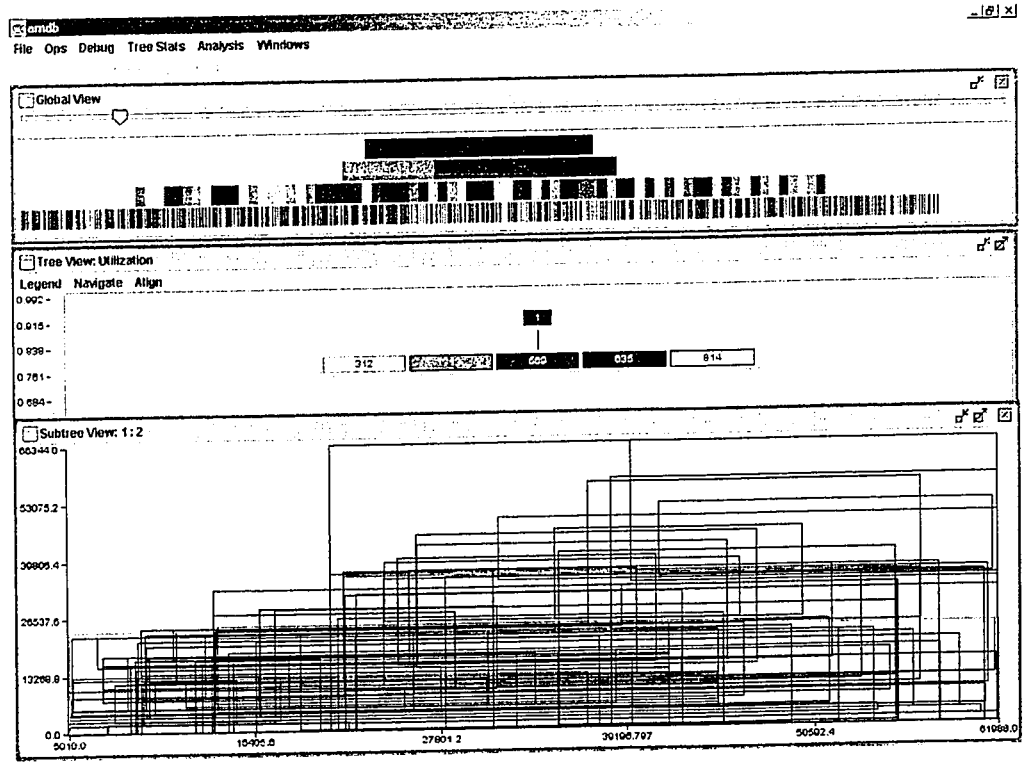
A.4 VL_50_R2_rect_Uc



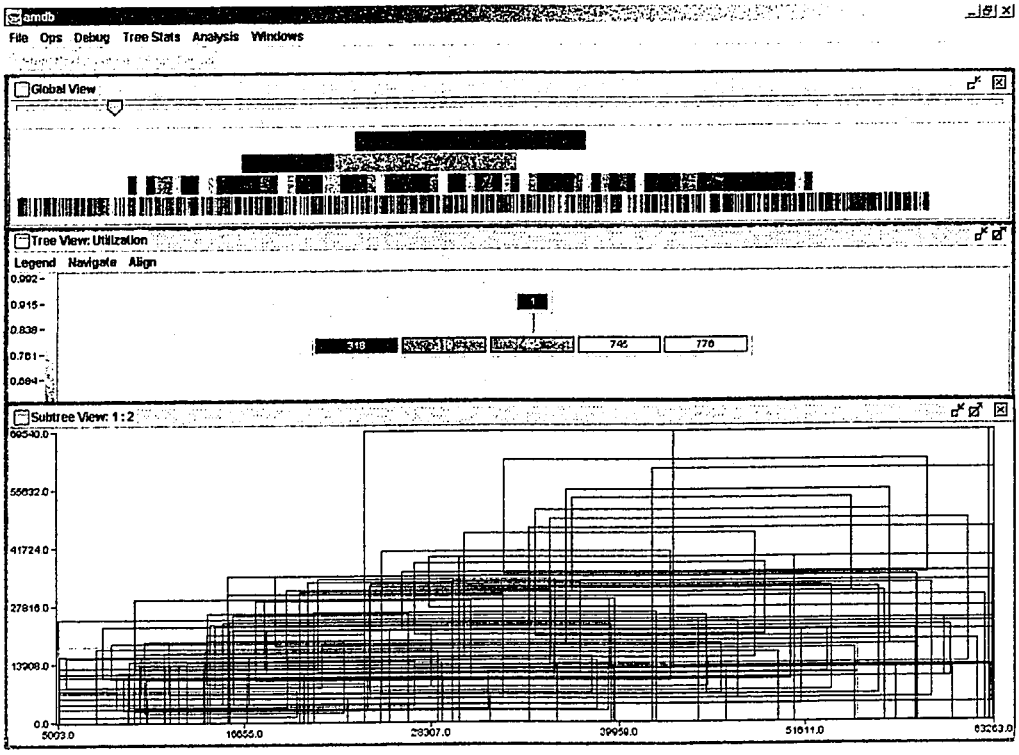
A.5 VL_5000_R2_rect_Uc



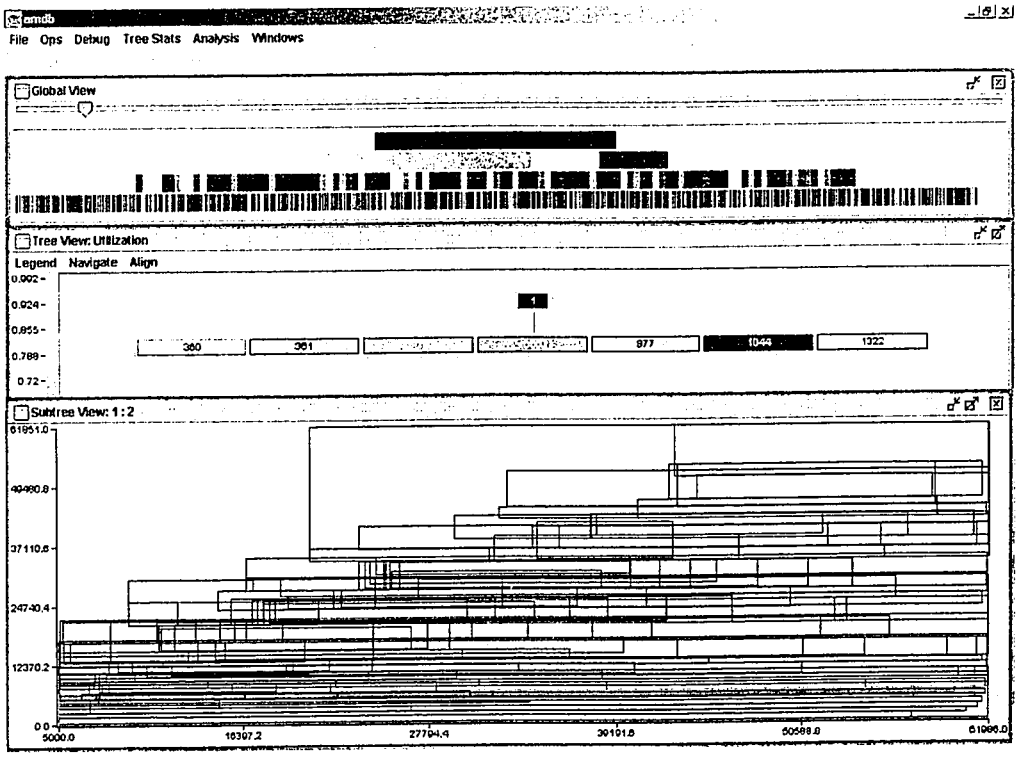
A.6 VL_50_R4_rect_NoVar



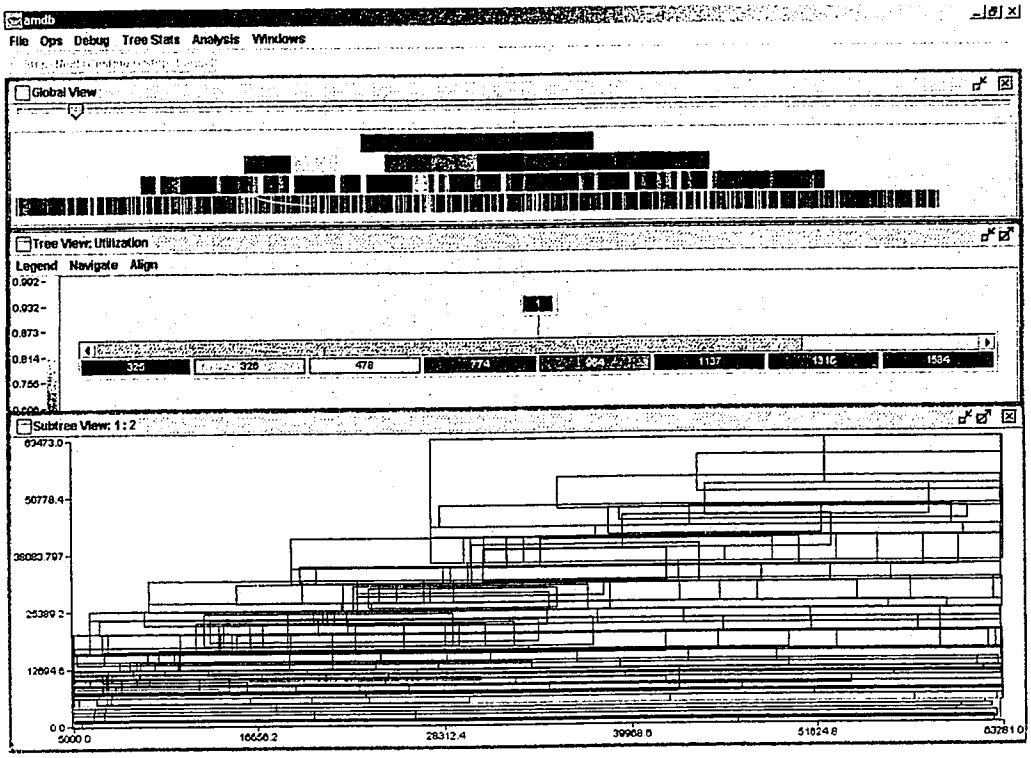
A.7 VL_5000_R4_rect_NoVar



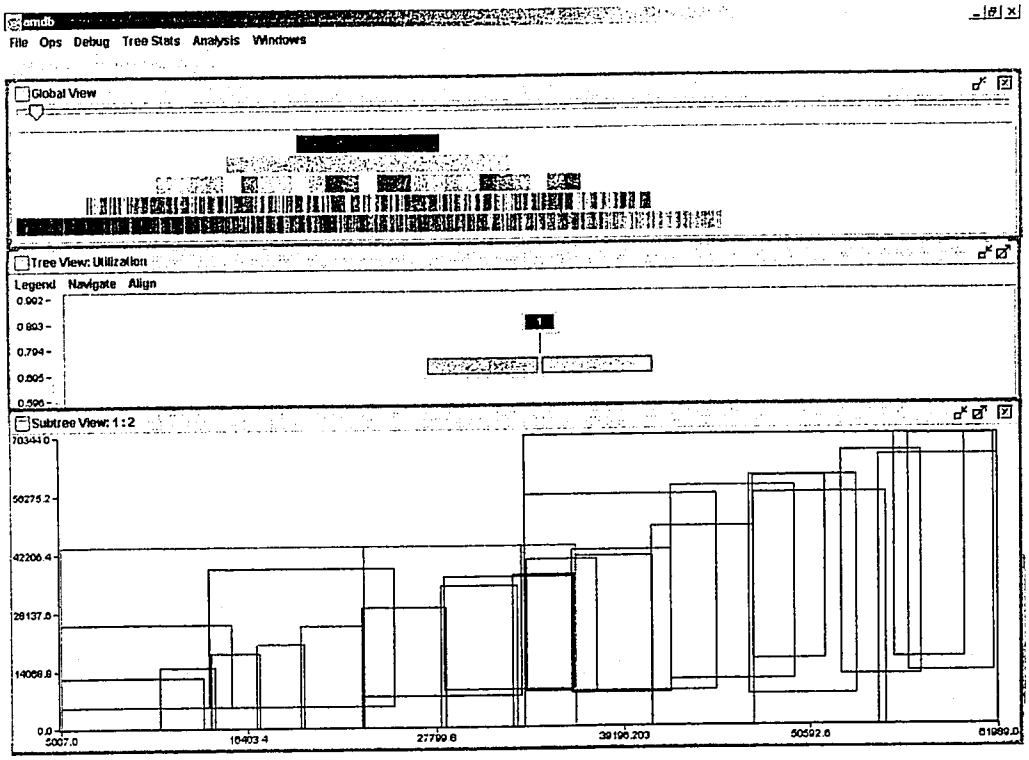
A.8 VL_50_R4_rect_Now



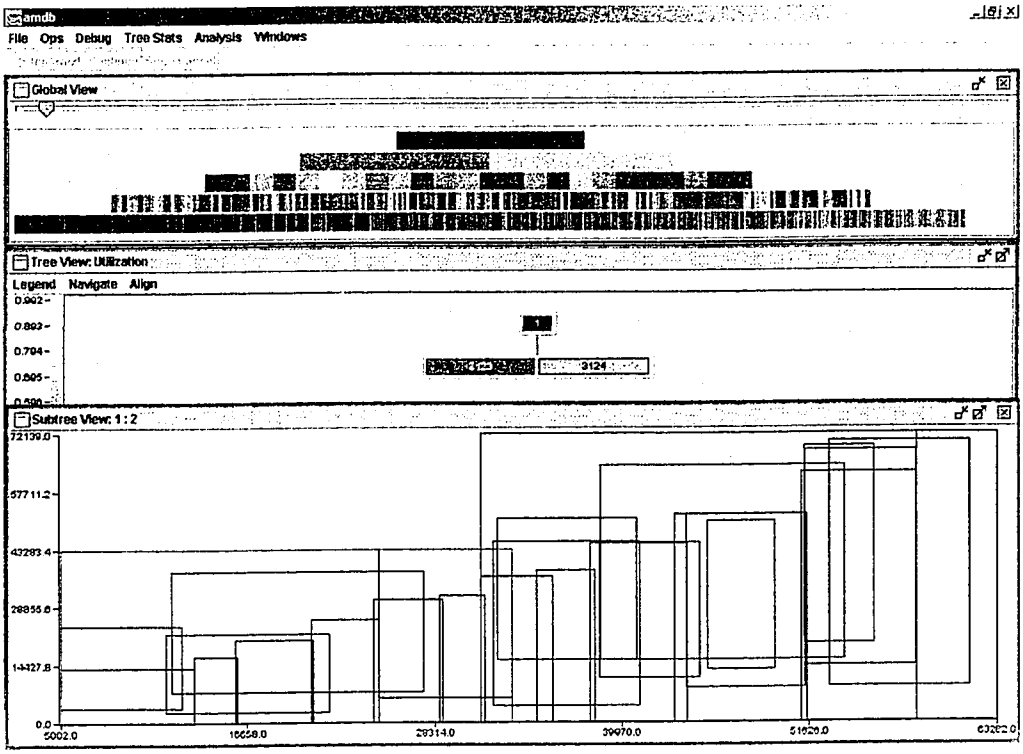
A.9 VL_5000_R4_rect_Now



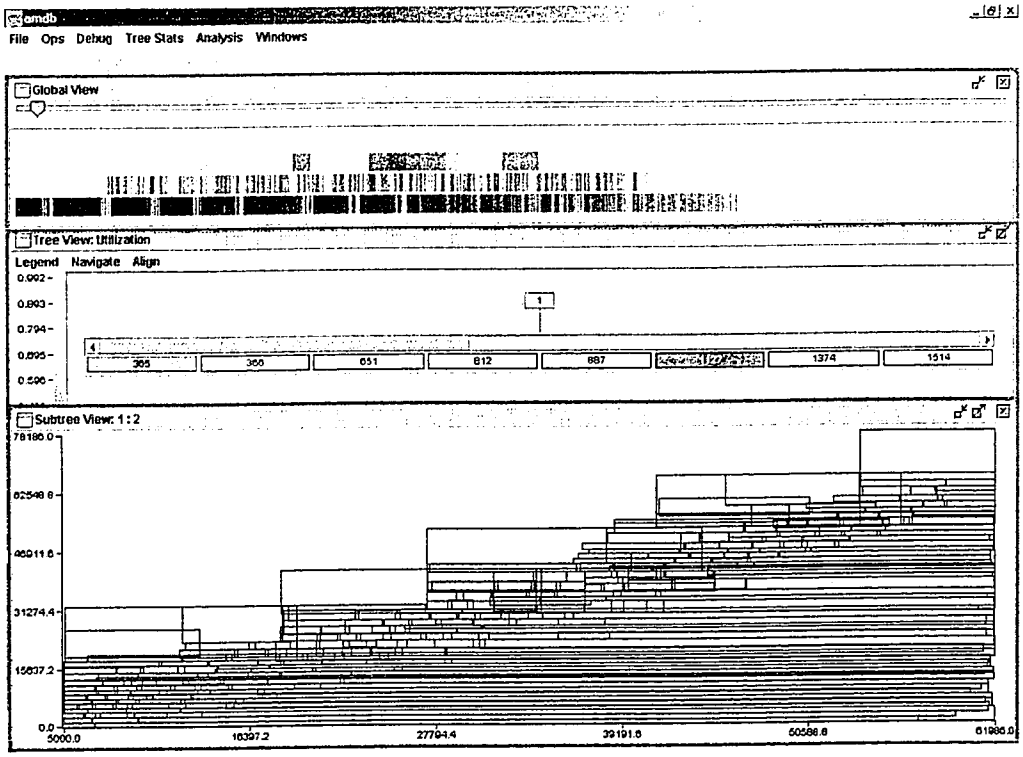
A.10 VL_50_R4_rect_Uc



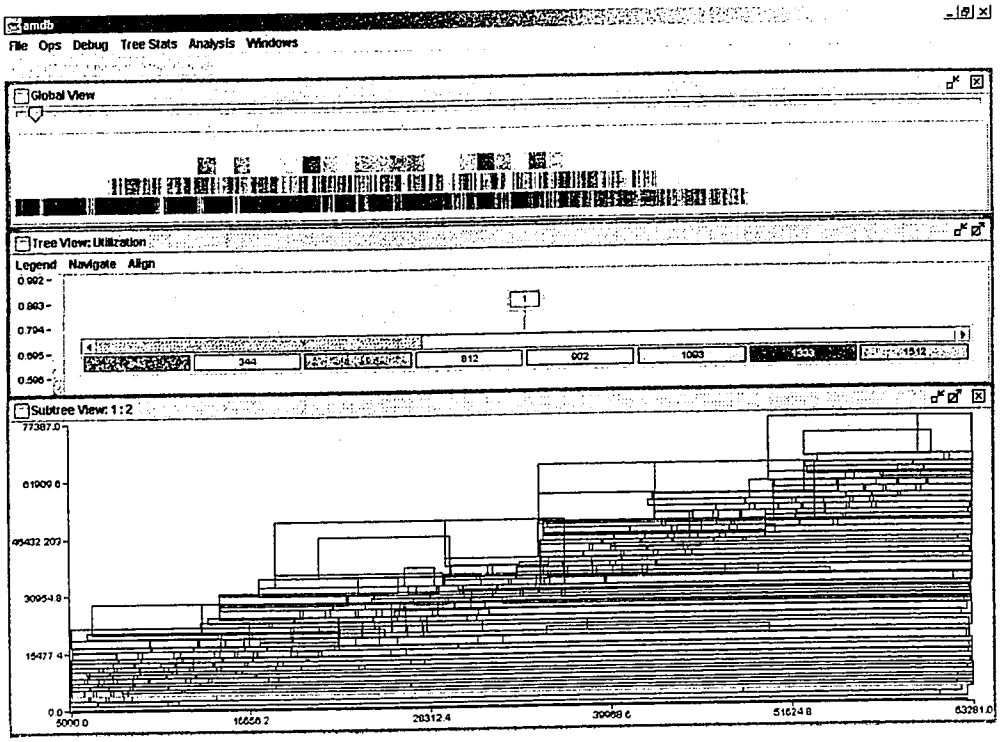
A.11 VL_5000_R4_rect_Uc



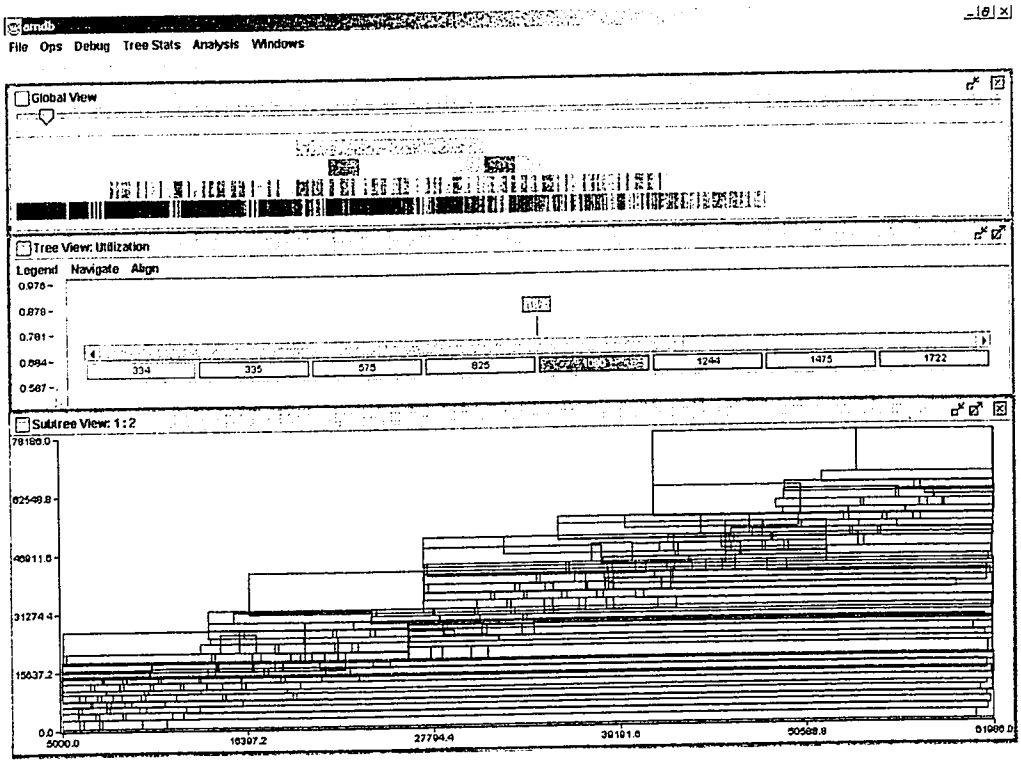
A.12 VL_50_R4_rect_UcNow

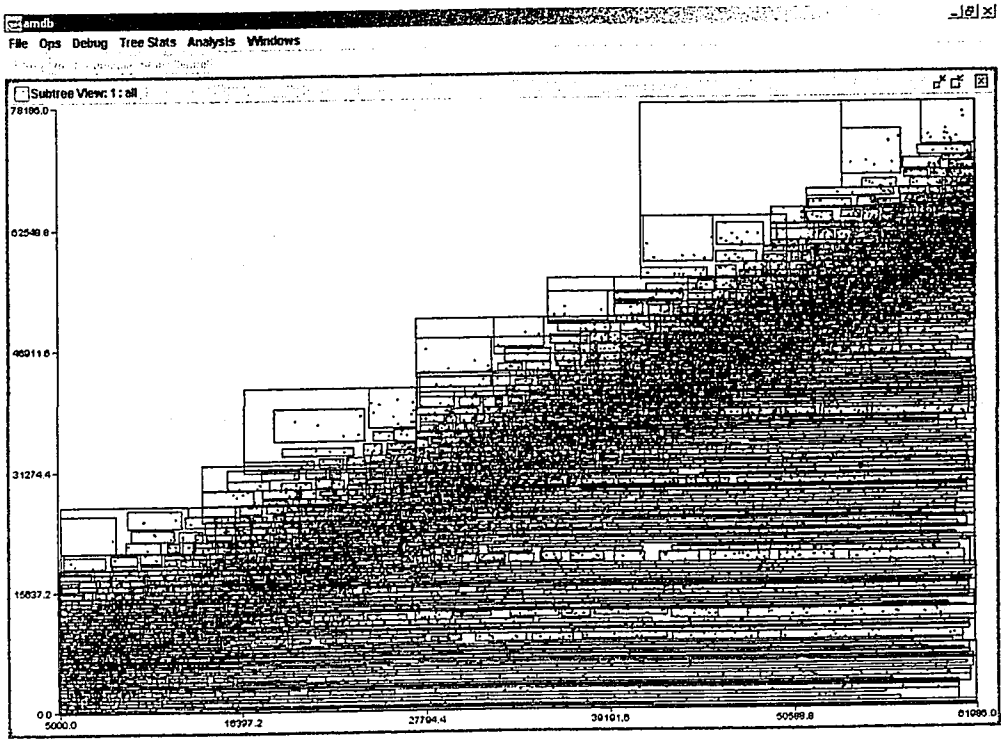


A.13 VL_5000_R4_rect_UcNow

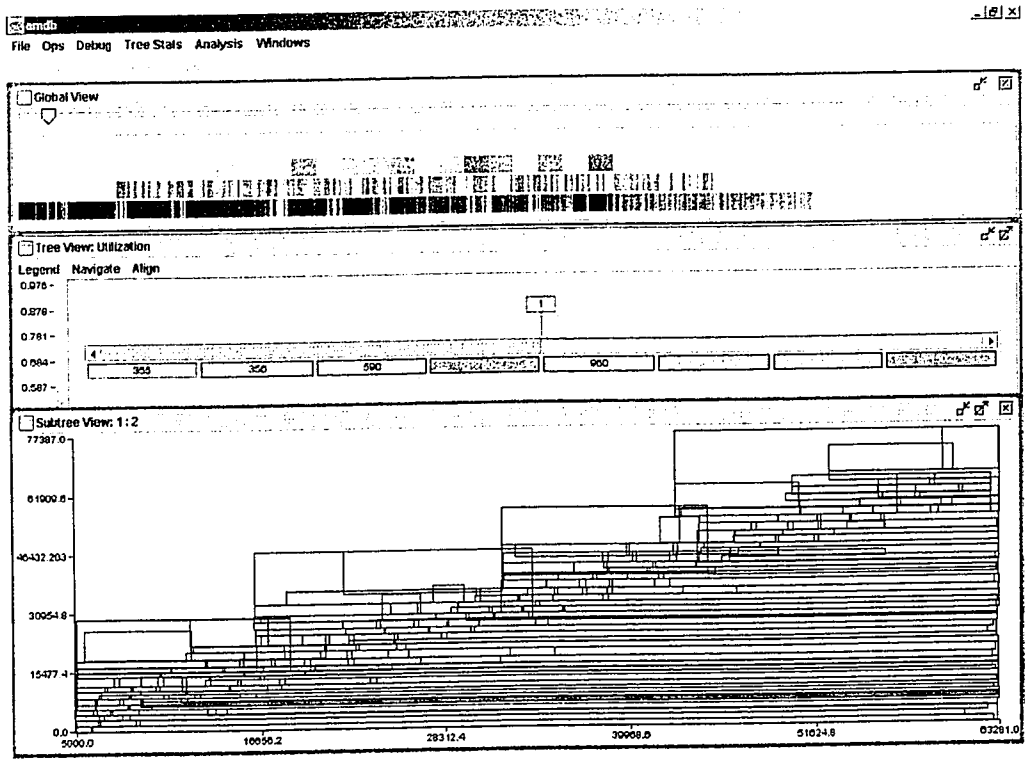


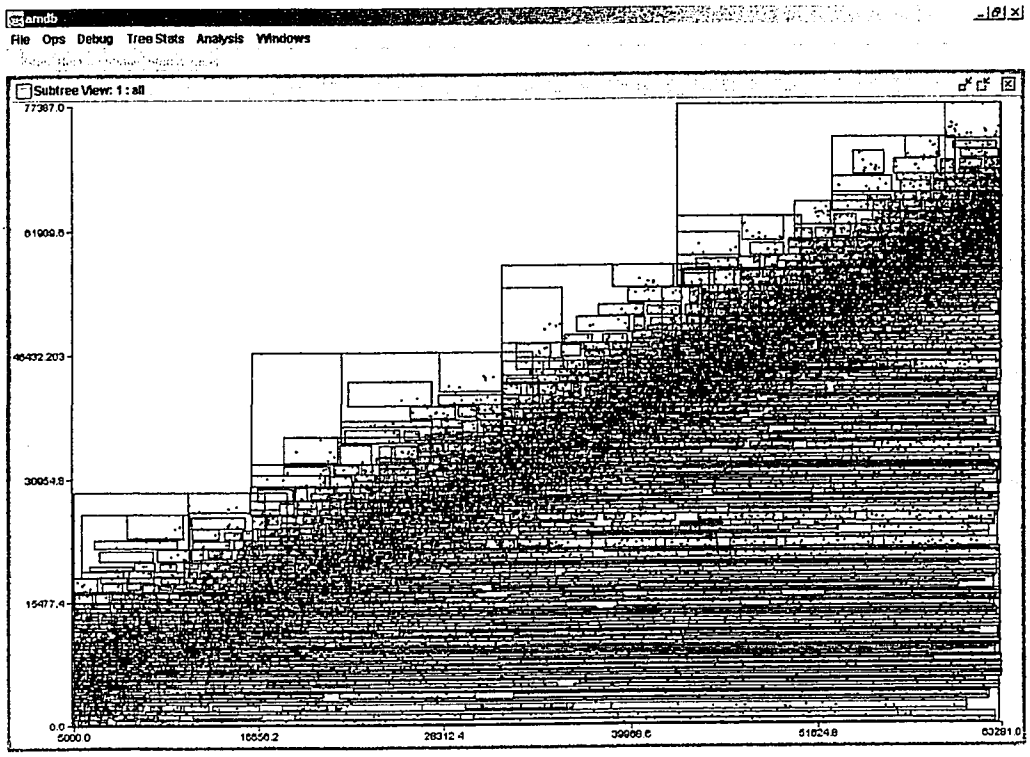
A.14 VL_50_R4_point_UcNow



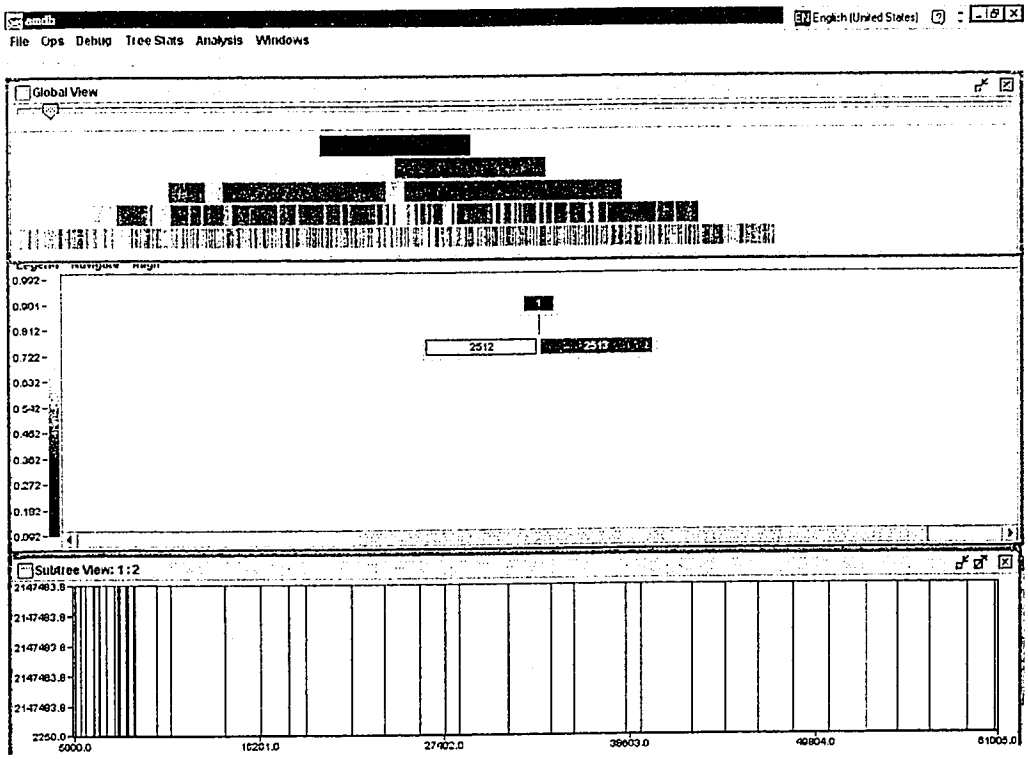


A.15 VL_5000_R4_point_UcNow

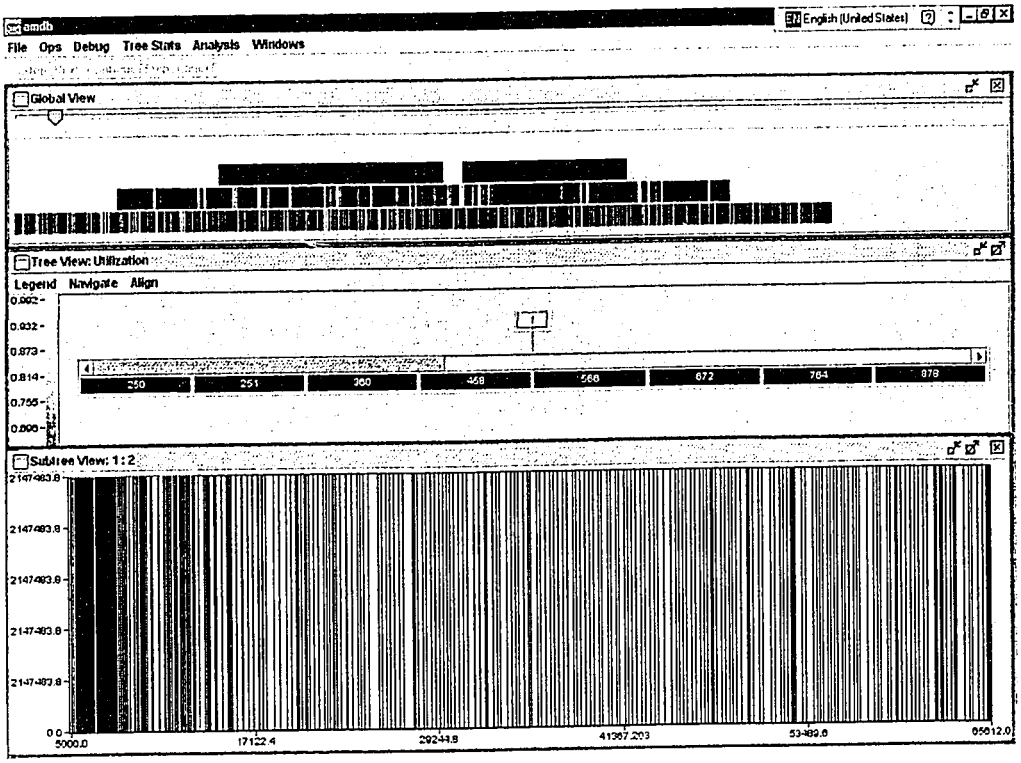




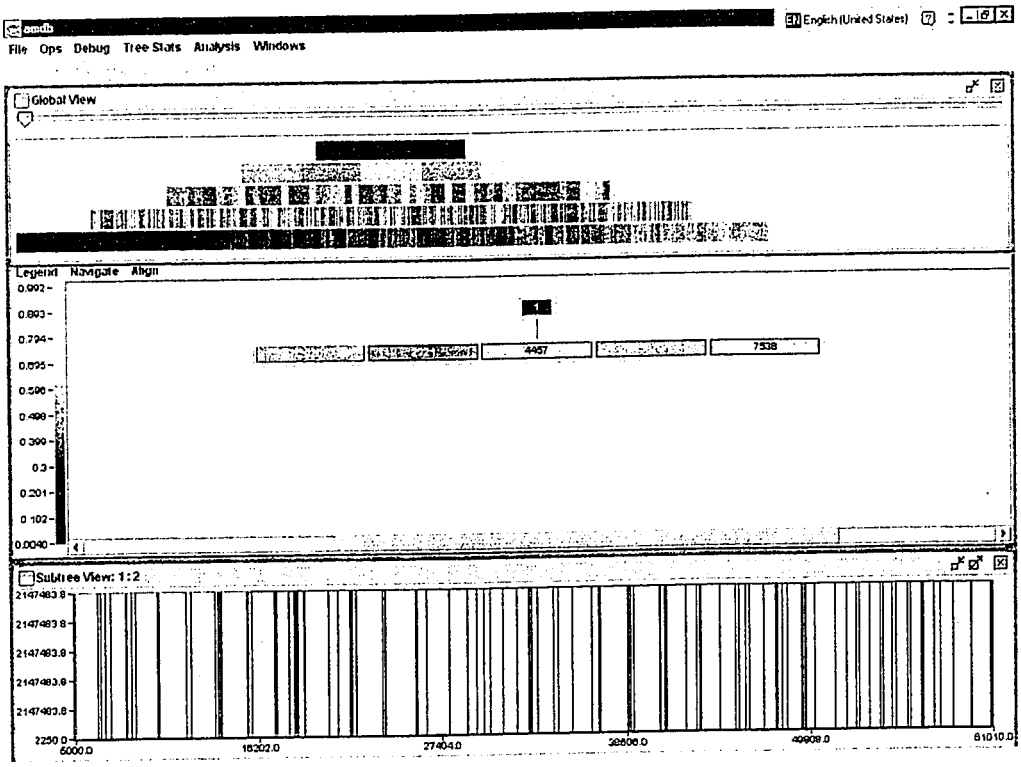
A.16 Dev_1000_R2_rect_NoVar



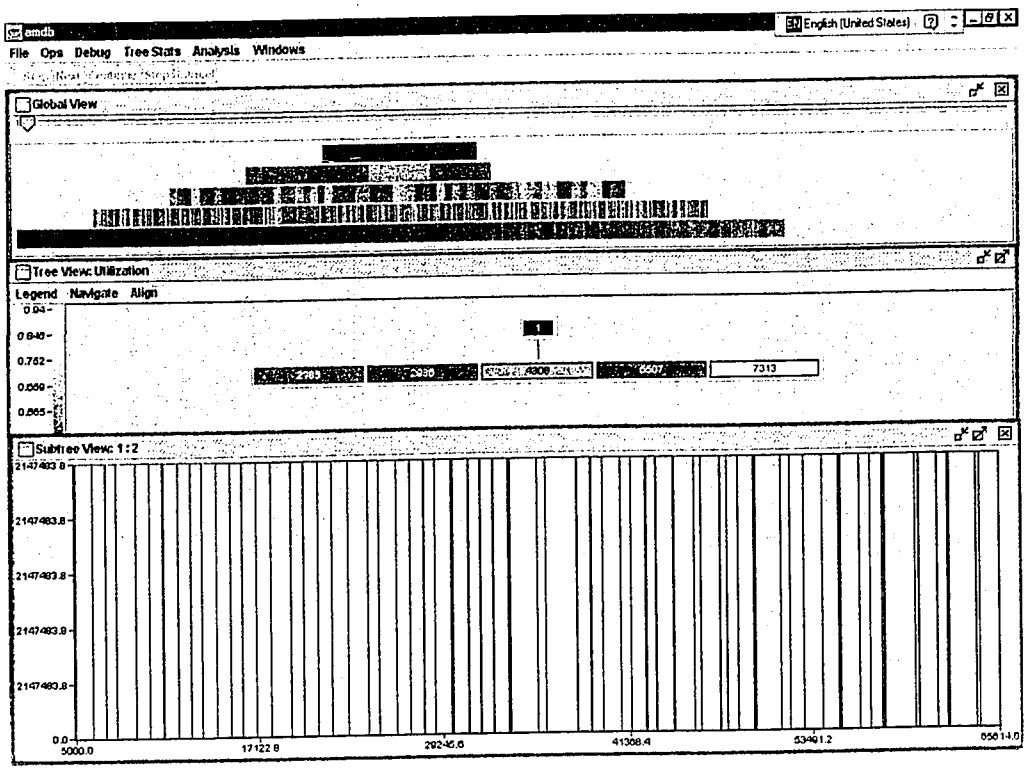
A.17 Dev_50000_R2_rect_NoVar



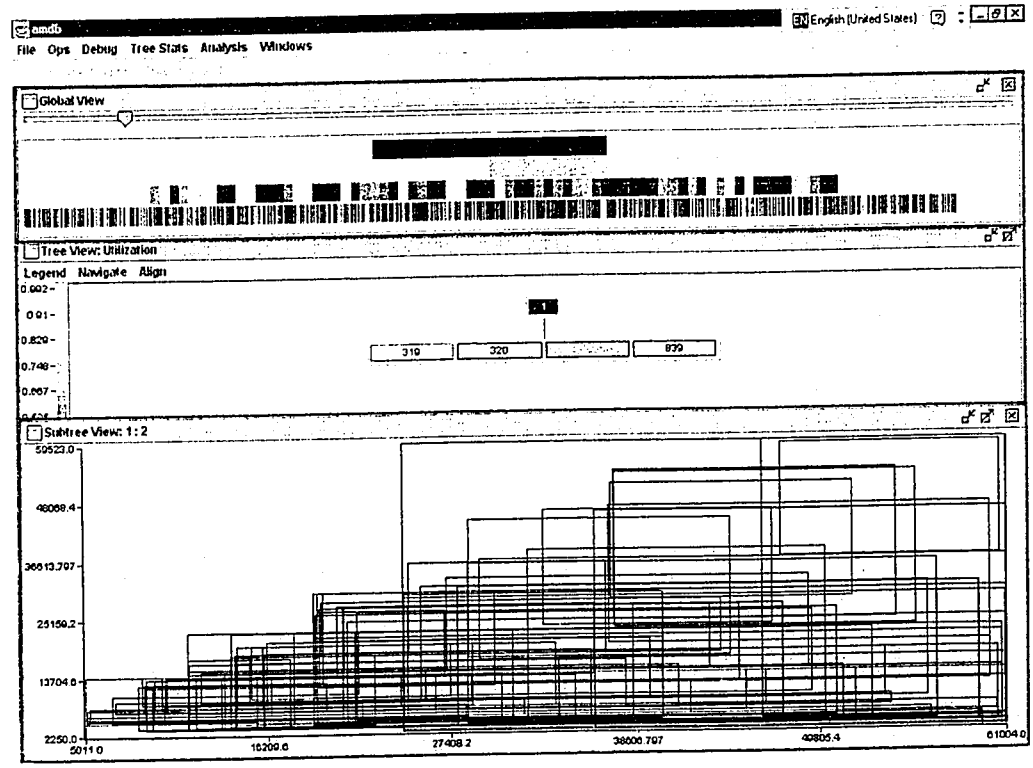
A.18 Dev_1000_R2_rect_Uc



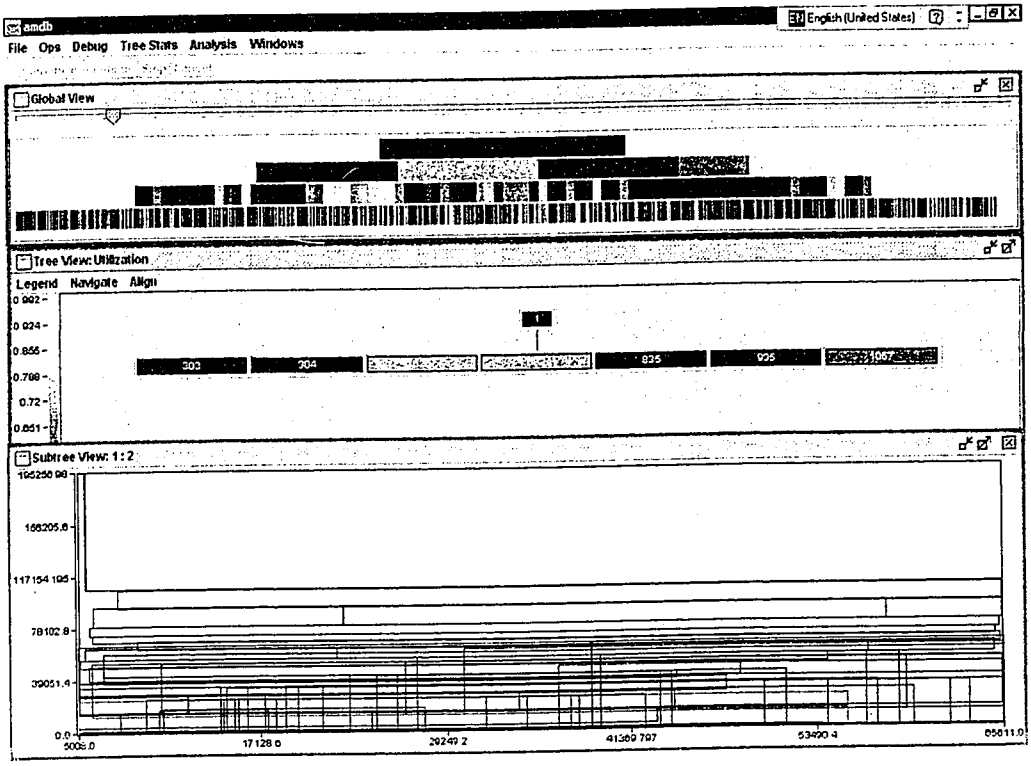
A.19 Dev_50000_R2_rect_Uc



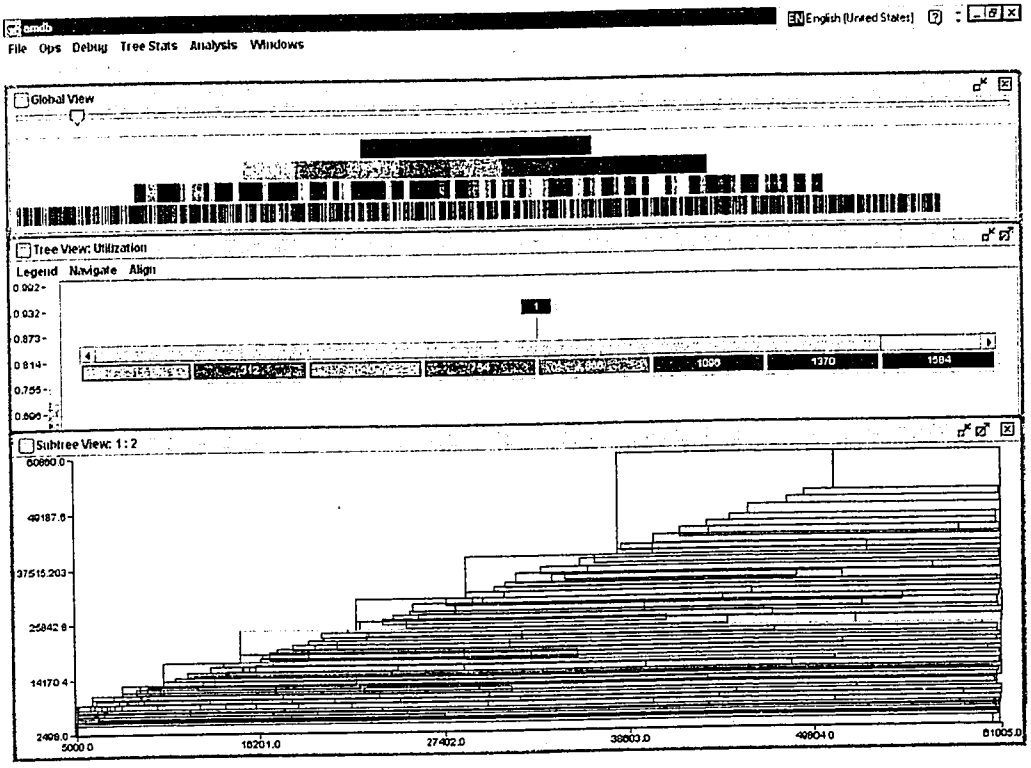
A.20 Dev_1000_R4_rect_NoVar



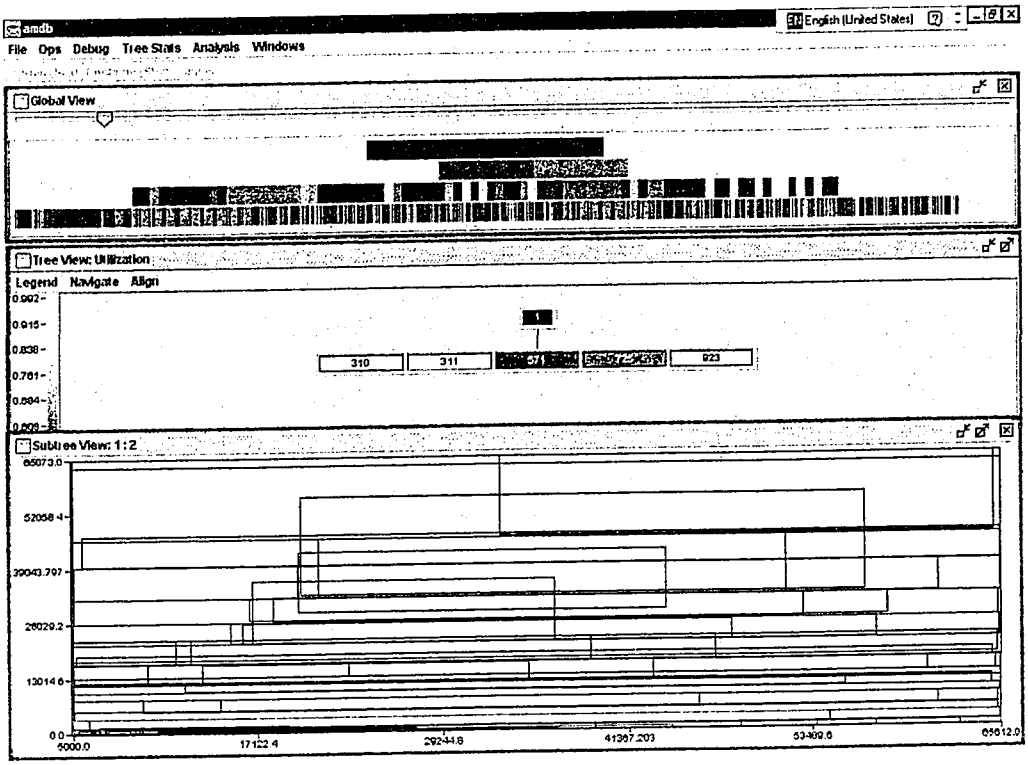
A.21 *Dev_50000_R4_rect_NoVar*



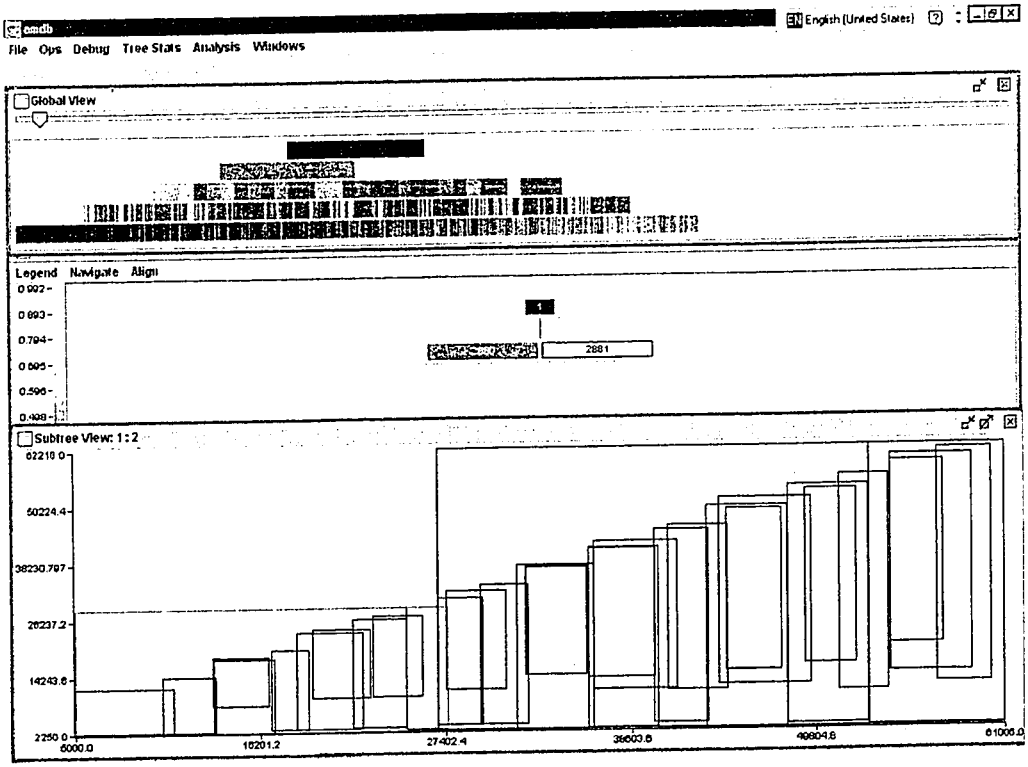
A.22 Dev_1000_R4_rect_Now



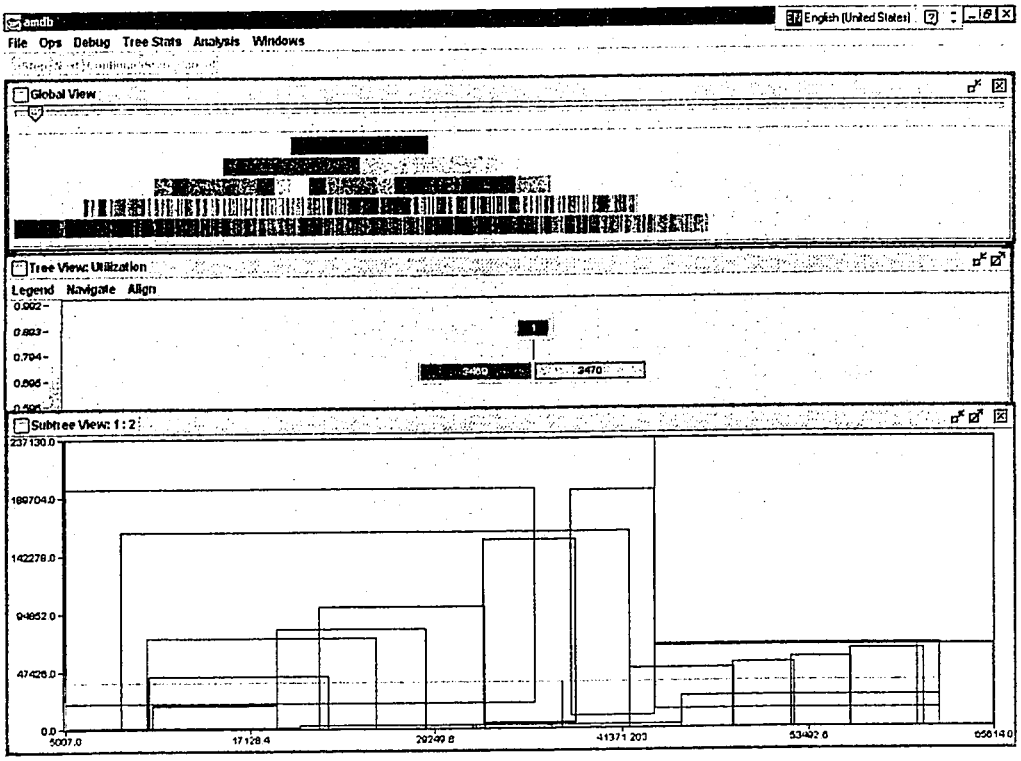
A.23 Dev_50000_R4_rect_Now



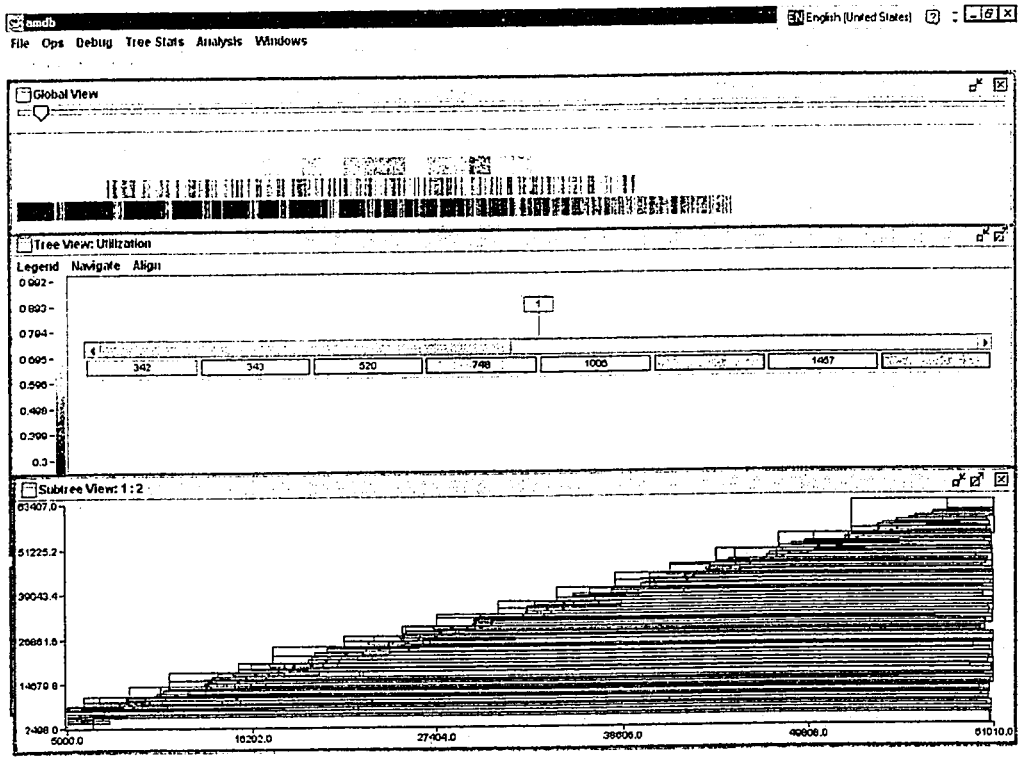
A.24 Dev_1000_R4_rect_Uc



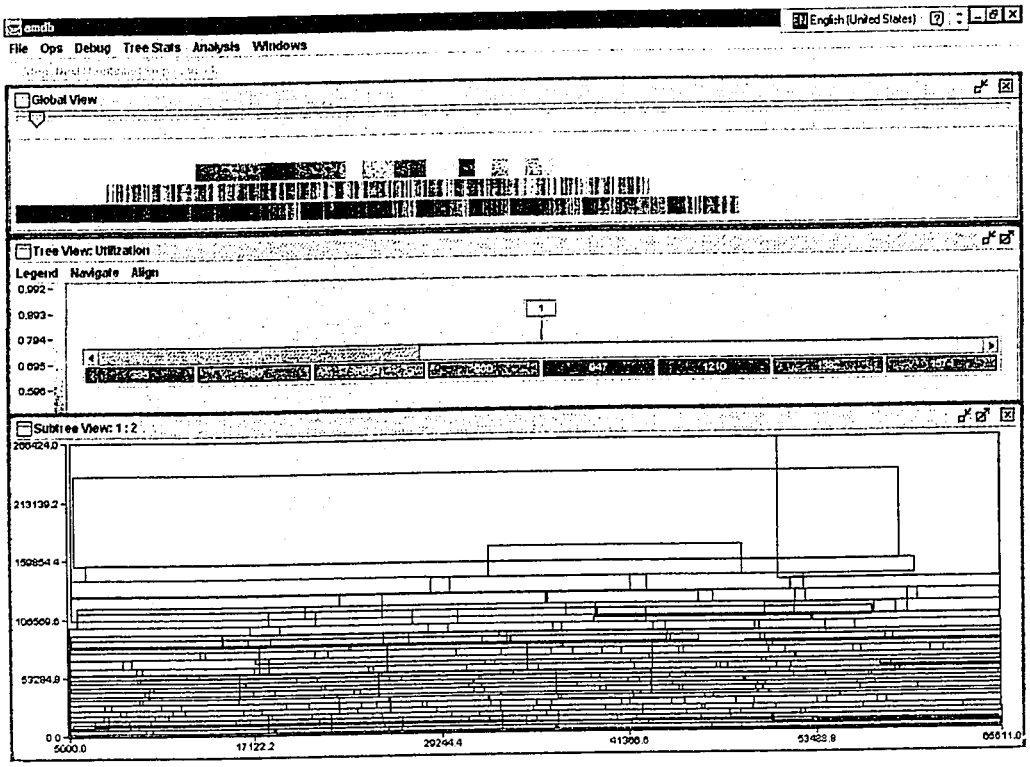
A.25 *Dev_50000_R4_rect_Uc*



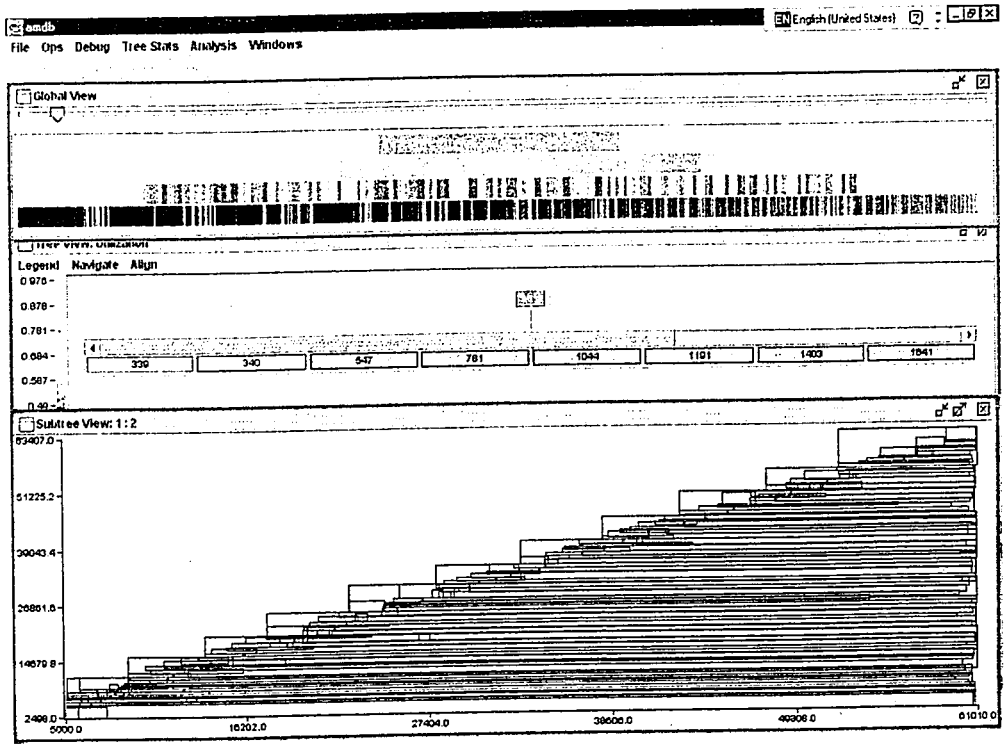
A.26 Dev_1000_R4_rect_UcNow

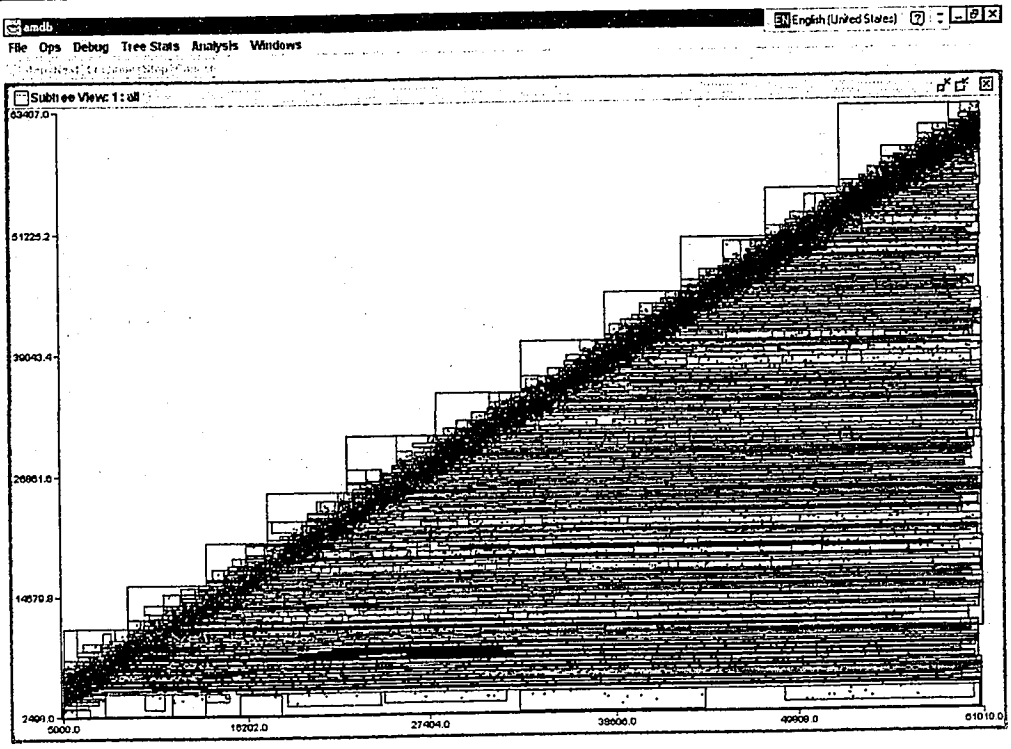


A.27 Dev_50000_R4_rect_UcNow

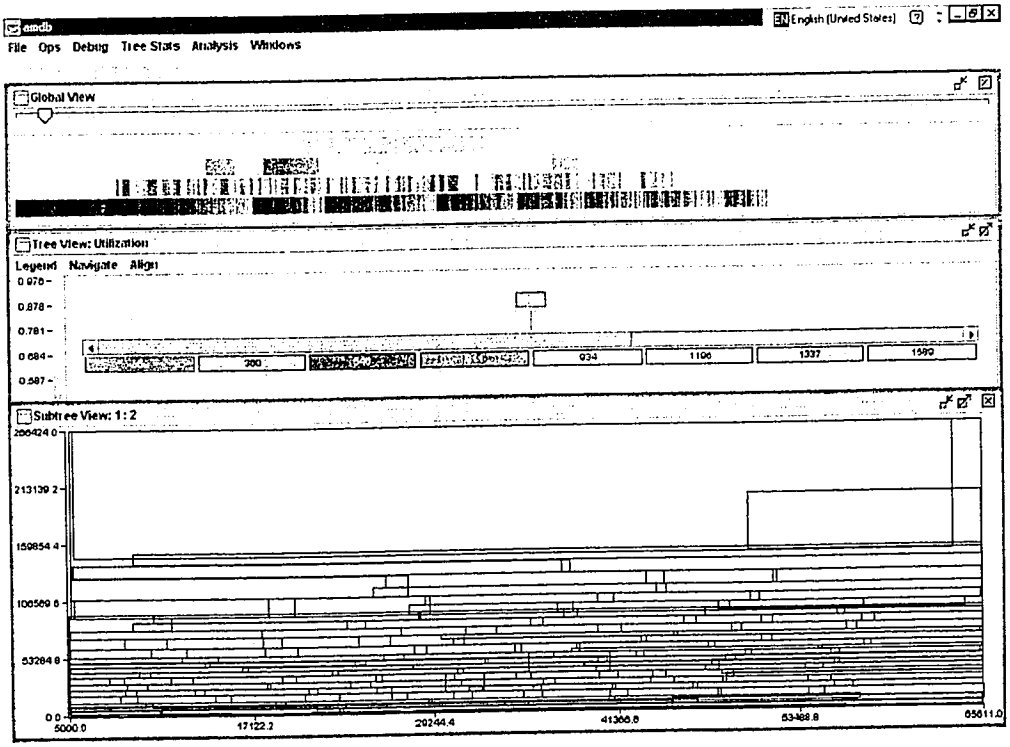


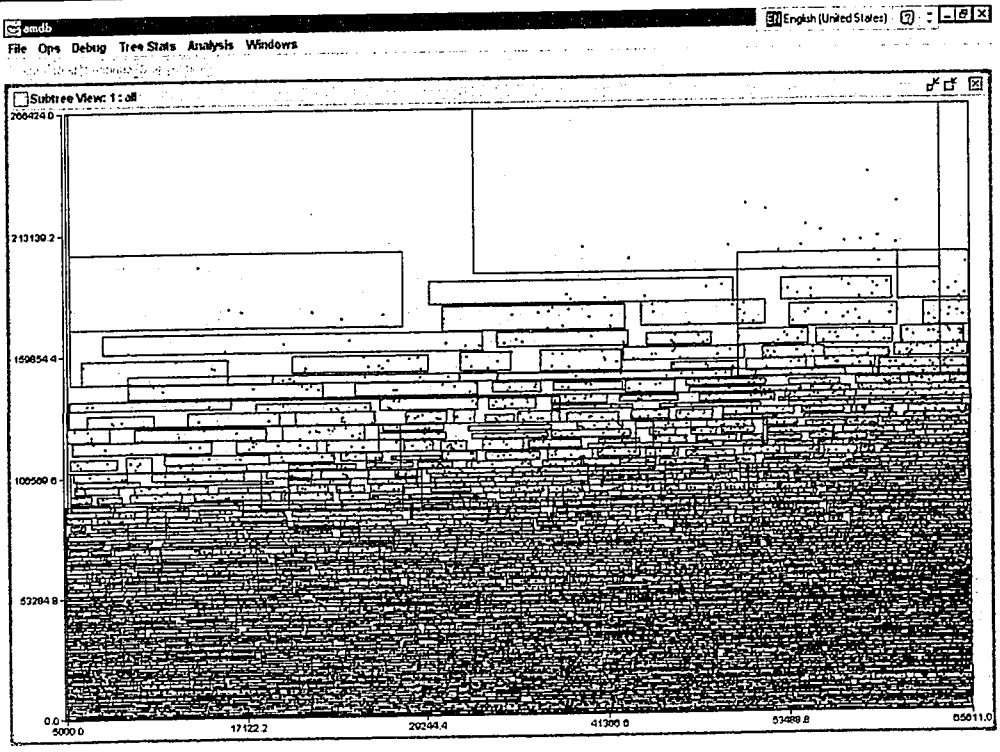
A.28 Dev_1000_R4_point_UcNow





A.29 Dev_50000_R4_point_UcNow




emdb
English (United States)
File Ops Debug Tree Stats Analysis Windows
Subtree View: 1 : all
266424.0
213139.2
159854.4
106569.6
53284.8
0.0
5000.0
17122.2
29244.4
41366.6
53488.8
65611.0