Towards the universal spatial data model based indexing and its implementation in MySQL

Evangelos Katsikaros

Kongens Lyngby 2012
IMM-M.Sc.-2012-97

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-M.Sc.: ISSN XXXX-XXXX

Summary

This thesis deals with spatial indexing and models that are able to abstract the variety of existing spatial index solutions. This research involves a thorough presentation of existing dynamic spatial indexes based on R-trees, investigating abstraction models and implementing such a model in MySQL.

To that end, the relevant theory is presented. A thorough study is performed on the recent and seminal works on spatial index trees and we describe their basic properties and the way search, deletion and insertion are performed on them. During this effort, we encountered details that baffled us, did not make the understanding the core concepts smooth or we thought that could be a source of confusion. We took great care in explaining in depth these details so that the current study can be a useful guide for a number of them.

A selection of these models were later implemented in MySQL. We investigated the way spatial indexing is currently engineered in MySQL and we reveal how search, deletion and insertion are performed. This paves the path to the understanding of our intervention and additions to MySQL's codebase. All of the code produced throughout this research was included in a patch against the RDBMS MariaDB.

Preface

This thesis was prepared at the Department of Informatics and Mathematical Modeling of the Technical University of Denmark, in partial fulfillment of the requirements for acquiring the M. Sc. E. degree in Computer Science and Engineering. This study has been conducted from May 2012 to August 2012 under the supervision of Associate Professor François Anton and the co-supervision of Sergei Golubchick of “Monty Program AB”. It represents a workload of 30 ECTS points.

Lyngby, August 2012

Evangelos Katsikaros

Acknowledgements

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

– *Edsger W. Dijkstra* [15]

This project wouldn't be possible without the help of many people. I would like to thank my parents for their support, and my friends both in Greece and Denmark that assisted in numerous ways. Two people are mainly responsible for this work: François Anton and Sergei Golubchick.

This project wouldn't have been complete without the constant assistance and prodding of François who supervised it. Despite the fact that we were located in different countries during the whole length of the project, he managed to orchestrate everything in the best way.

The implementation part wouldn't have been complete without Sergei who co-supervised the project on behalf of "Monty Program AB". With patience and immense expertize in MySQL server internals, he was an irreplaceable guide through thousands lines of code.

I would also like to thank Yannis Theodoridis and Joseph M. Hellerstein for taking the time to answer questions regarding some of their publications that were vital to the bibliographical research of this project.

Dedicated to Dimitra.

Contents

CHAPTER 1

Introduction

This chapter highlights the background of the thesis and outlines its structure. The chapter is organized as follows: in Section 1.1 we explain why there is a great need for systems that can handle data, and more specifically spatial data, efficiently. In Section 1.2 we continue by explaining why indexes are important in data management. In Section 1.3 we define the goals of the thesis and specify the outcome of our research. In Section 1.4 we present the main literature sources. In Section 1.5 we dive into the major spatial indexing standards, their adoption and different implementations. Finally, in Section 1.6 the organization of the thesis is outlined.

1.1 Data, DBMS and GIS

The wide spread usage of computer devices has induced an explosive growth of the amount of data produced and collected. It is not easy to measure the total volume of data stored, however an International Data Corporation (IDC) estimate puts the size of the “digital universe” at 0.18 zettabytes (10^{21} bytes or 1 billion terrabytes) in 2006, and is forecasting a tenfold growth by 2011 to 1.8 zettabytes [22]. The data sources are countless including machine logs, RFID readers, sensor networks, vehicle GPS traces, financial and commerce transactions, photographs, medical and astronomical images, video and so on.

A DataBase Management System (DBMS) is capable of storing and handling these large data sets. According to [14, p. 5], a DBMS is “a computerized system whose overall purpose is to store information and to allow users to retrieve and update that information on demand”. Applications usually have quite common needs when it comes to storing, retrieving and updating information such as:

- network connectivity;
- the ability to distribute data in many machines, in order to achieve high read/write performance and replication/availability;
- the ability to accommodate a large number of users, that can read and write at the same time; and
- intact recreation of the data even if something goes wrong.

A DBMS handles the basic needs of efficient storage and fast extraction of data, as well as other common trivial and non-trivial tasks. In this way, a DBMS can free an application from the low level details of storing, retrieving and updating information [41, pp. 714–718].

In 1970, Codd presented his seminal work on the relational model [12] that targeted a) *data independence* of the DBMS user or application from the changes in data representation and b) *data consistency*. The relational model gained wide acceptance in the 1980s and is currently dominating database management systems [41, p. 715], with the Relational DataBase Management System (RDBMS) being a very common choice to manage data. At the same time, it became apparent that new applications, like multimedia, Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM), medical, geographical, and unstructured data just to name a few, were not accommodated well by the relational model [41, p. 1929], [42, p. 3].

Efforts to adapt the relation model to some of these challenges, led to an object–oriented approach and the original implementation of PostgreSQL [112], one of the first object–relational DBMS. Moreover, many highly distributed systems like Casandra [39], Hadoop [117] and MongoDB [11] have emerged. These systems deviate from the relational model, by relaxing the data model or the integrity checking or the schema structure, in order to meet very specific needs and handle semi or unstructured data. Over the years, many DBMSes have incorporated, in varying degrees, object–oriented features and functionality, and it is likely that the differences between relational and other types of databases will blur, as features from different models will be incorporated in others [117, p. 6], [48].

Table 1.1: List of Common GIS Analysis Operations [106, p. 3], [4].

Search	Thematic search, search by region, (re)classification
Locational Analysis	Buffer, corridor, overlay, Thiessen/Voronoi
Terrain Analysis	Slope/aspect, catchment, drainage network, viewshed
Flow Analysis	Connectivity, shortest/longest path
Distribution	Change direction, proximity, nearest neighbor
Spatial Analysis	Pattern and indices of similarity, autocorrelation, topology
Measurements	Distance, perimeter, shape, adjacency, direction

One of the challenging areas in databases is geographical applications and spatial data [2, 1], covering data managing for mapping and geographic services, spatial planning in transportation, constructions and other similar areas, and location based services for mobile devices. The special needs of spatial data have created the need of Geographic Information Systems (GIS). According to [106, p. xxi] "GIS is a computer system for assembling, storing, manipulating and displaying data with respect to their locations". Whereas RDBMSes are good in handling alphanumeric data and answer related queries, for example "list the top ten customers, in terms of sales, in the year 1998", spatial queries like "list the top ten customers, within a 5 km radius from branch X" require special abilities. In table 1.1 we present a list of common spatial analysis operations. Usually, a GIS can be built as a front-end of a spatially enabled DBMS, that has the ability to handle spatial data and perform simple spatial analysis.

This introduction to RDBMS and other types of DBMSes emphasized their main characteristics, the ability to store and extract large volumes of information well, and handle common tasks for theses applications. Additionally, several spatial services and GIS operations were described.

1.2 The compulsory need for indexes

Another issue, that arises when databases grow in volume, is the efficiency of retrieving data. According to [41, pp. 1425], an index is "a set of data structures that are constructed from a source document collection with the goal of allowing an information retrieval system to provide timely, efficient response to search queries". The most common type of data structure, used for indexes, are trees and hash structures [41, p. 1433].

One of the most influential works on indexing trees is Comer's B-tree publication [13], that presented the family of binary search trees. B-trees are now a standard part of textbooks on databases and they have grown to a de facto indexing solution for DBMSes and filesystems. Its most well known variant is Knuth's B$^+$-tree [41, p. 1433, p. 3173].

In the same manner that RDBMSes cannot accommodate well some types of applications, such as multimedia and spatial applications, B-trees don't fit well to certain types of data — their original design aimed alphanumeric data like integers, characters and strings. As a consequence, a number of B-tree variations, targeting specific applications, has appeared in the literature [8].

One important family among the newly proposed indexes was the family of R-trees by Guttman in 1984 [28]. It aimed at handling spatial data, including both one-dimensional such as points, and two-dimensional such as polygons and surfaces, three dimensional such as polygons, surfaces, volumes and higher dimensional objects. In the same manner that B-trees became an industry standard indexing solution, R-trees are now common in geographical, spatial, temporal and moving objects applications and databases. R-trees are going to be further analyzed in Chapter 2.

The way data is organized can be different depending on whether data change often or rarely. The two main types of indexes are dynamic and static:

Static indexes are used in cases where changes occur rarely or at specific time intervals, like it is strongly the case with census data and in some cases in cartographic data. In these cases we are interested in optimizing factors like maximum storage utilization, minimum storage overhead, minimization of objects' coverage (in order to improve retrieval performance), or a combination of the above. Since data changes are rare, in the long term, it is efficient to optimize these factors, even if the methods used to achieve this optimization are costly. This is performed by methods known in the literature as *packing* and *bulk inserting* [42, p. 35].

Dynamic indexes are used when objects are inserted in the index in a one–by–one basis. Even if the factors, that interests us in static indexes, apply in this case too, the methods used to achieve performance have to make compromises between cost of tree changes and tree efficiency.

The need for efficient indexing created an explosion of indexing solutions, and as others noticed "trees have grown anywhere" [104], fully justifying the title of Comer's article "The Ubiquitous B-Tree". Over time B-trees and R-trees became standard indexing solutions and are used in a significant number of applications.

1.3 Thesis Specification

In this section we specify the goals and the scope of the thesis. In section 1.3.1 we discuss the reasons we use the RDBMS MySQL and in section 1.3.2 why we decided to collaborate with "Monty Program AB". In section 1.3.3 we present the objectives of the research and finally in section 1.3.4 we summarize the thesis specification.

1.3.1 The RDBMS MySQL

MySQL was first released in 1995. The company and the community around the product grew a lot and in 2008 MySQL AB was acquired by Sun Microsystems [68]. Finally, in 2009 Sun was acquired by Oracle [83].

MySQL is a proven database tool used in heavy-duty production envirnomnents. 16 out of the 20 most frequently visited web sites worldwide use it in larger or smaller part of their infrstructure [82]. Users of MySQL include:

- web sites like Google [27], Facobook [17], Twitter [113], Yahoo [119], Flickr [20] and Etsy [16];
- telecom companies like Virgin Mobile [69], Nokia [81] and Deutsche Telekom [108]; and
- numerous prominent companies in a variety of sectors [67].

Moreover, it's an open-source project. This means that:

- the software can be used without any cost for both academic and industrial pusposes; and
- the code is available so the academic community can use this RDBMS as an implementation sandbox, demonstration and benchmark tool for research projects.

Taking under consideration the above, we chose MySQL because it's a proven RDBMS and our small contibution could potentially benefit a large pool of industry or academic users.

1.3.2 MariaDB and Monty Program AB

The original creator of MySQL Michael "Monty" Widenius left Sun Microsystems in order to create his own company "Monty Program AB". They forked MySQL and created a new database product called MariaDB. "Monty Program AB" turned into a center of engineering excellence for MariaDB, the Aria storage engine, MySQL, and other associated technologies. Most of the company's developers are original core MySQL engineers, and most of the original core MySQL engineers left Sun and later Oracle to join the new company [98].

MariaDB is backwards compatible with MySQL as far as SQL and features are concerned. The application that runs on top of MySQL can keep working with MariaDB without any modifications. Additionaly, the MariaDB server is a binary replacement for the MySQL server. This means that all the software that was compiled and configured to work with MySQL can keep working with MariaDB without recompiling or reconfiguring [37].

Taking under consideration the above, we chose to work on the MariaDB RDBMS. The research was performed with the collaboration of "Monty Program AB", since the company is considered home of world's top MySQL expertise. This research is co-supervised by Sergei Golubchik, one of the first ten employees of the original MySQL company and an expert in the MySQL server code.

In the rest of the thesis when we refer to "MySQL" we refer to the MariaDB codebase or the MariaDB server, because the two terms can be used interchangeably for the purpose of this research. When we refer explicitly to MariaDB we do this to describe a specific version or feature that is available in MariaDB only.

1.3.3 Objectives

The goal of our research is to improve the current spatial indexes available in the RDBMS MySQL. As we will see in the following sections, even if MySQL is a widely used and accepted RDBMS product, its spatial capabilities do not match the capabilities of other DBMS products. There is an ongoing effort to improve the spatial side of MySQL, and this research is a small contribution to this effort.

Our first objective is to improve the way indexing is currently implemented in MySQL, by adding a data structure that abstracts indexing functionality. The data structure we selected was the Generalized Search Tree (GiST) that has already proved useful in PostgreSQL, an RDBMS product widely used for GIS applications (see section 1.5.2.4). Despite their differences, different index trees share similar functionality, and as a consequence, it is possible to create an abstraction data structure that covers these similarities. The obvious benefits of an abstraction level like this are the reduction of redundant code, and the ease to implement new indexes and extend the features of the existing ones. However, great care must be put in the performance overhead that, unavoidably, every abstraction level creates. We are going to investigate the current structure of MySQL's internal and more specifically the indexing code, analyze its behavior, and then implement the abstraction layer.

The second objective is to improve the available spatial indexing capabilities of MySQL. In order to achieve this goal, we investigate the recent bibliography on spatial indexes to gain a broad perspective of the subject and select the most promising options. We then implement the R*-tree in MySQL, using the abstracted data structure (GiST) we have already created.

The research is using a wide range of solutions published in the relevant literature either recently (only last year) or dating more than 15 years ago. Some of these solutions are implemented either only for experimental purposes or in widely used RDBMSes. The originality of our research lies in:

- bringing together all these different abstraction and spatial solutions, under one RDBMS that didn't have this functionality before; and
- trying to push the limits of the GiST data type to investigate the variety of index tree solutions and the extensibility of query types it can accommodate.

1.3.4 Specification Summary

To summarize, this research covers:

- the recent bibliography concerning spatial indexes for low dimensions and more specifically dynamic spatial indexes,

- the recent bibliography concerning ways to abstract implementations of tree indexes, and
- an investigation of the way currently MySQL implements and uses indexes internally.

Moreover, the outcome of this research is:

- a data structure that abstracts index trees; and
- improved dynamic spatial index trees for low dimensions, based on the abstract data structure.

The above mentioned implementations come in the form of a patch against the latest MariaDB development release, and we make sure that it conforms to general good coding practices as well as the MySQL and MariaDB coding standards. The code we delivered can be compiled with both the MySQL and the MariaDB without any issues.

1.4 Main Research Sources

As we have already described in the research area specification (Section 1.3), we investigated literature related to spatial indexes and more specifically R-trees. This bibliography covers a great number of books, conferences and journals, and for this reason we needed some main sources to guide us through the material. These were:

- *Database Management Systems* [99], a book that covers the fundamentals of database systems in great detail;
- *Spatial Databases: A Tour* [106], a book that is considered a standard textbook in spatial data and spatial applications;
- *R-Trees: Theory and Applications* [42], an extensive survey of R-tree–related issues; and
- *Encyclopedia of Database Systems* [41], a comprehensive reference to about 1,400 entries, covering key concepts and terms in the broad field of database systems.

The above mentioned books include a large number of references, a lot of which we further investigated.

Apart from these main sources, we also researched a number of conference proceeding, journals and books for interesting and more recent material.

1.5 Standards for GIS

This section investigates the technical standards concerning GIS and RDBMSes. A big part of our research focuses on GIS systems, so in Section 1.5.1 we investigate the available standards from the Open Geospatial Consortium (OGC), in order to be aware of the widely used practices in this field. Then, in Section 1.5.2, we present how some well-known RDBMSes including MySQL conform to these standards.

Technical standardization is a process that creates a common base for the development of products and services. Well known organizations offering standards include the International Organization for Standardization (ISO), the world's largest developer and publisher of International Standards for various technical areas [32] and the World Wide Web Consortium (W3C), that defines Web technologies [116]. The success of both of these organizations, is based on the participation of a large number of members, from a variety of countries, covering the academic and industrial sectors and bridging the public and private sectors [114, 115, 34, 35].

Widespread adoption of standards is important for business, academia, governments and end–users, enabling the development of interoperable processes and solutions. Suppliers can develop and offer products and services meeting specifications that have wide international acceptance in their sectors as well as perform transactions in the domestic and global marketplace more easily. Finally, end–users have a broad choice of offers that meet well defined criteria [71, 33].

1.5.1 OGC Standards

The Open Geospatial Consortium (OGC) is a standardization organization focusing on interoperable solutions for GIS technologies and GIS web services. According to [73], "OGC is an international industry consortium of 406 companies, government agencies and universities participating in a consensus process to develop publicly available interface standards". OGC has compiled a number of standards including:

- Simple Features SQL [78, 79]: This standard is approved as an ISO standard (the ISO 19125 [78, p. 6], [79, p. viii], [107, p. 1133]) and it evolves as a collaboration and between OGC and ISO [80]. It consists of two parts, under the general title "Geographic information – Simple feature access":
  - Part 1 "Common architecture" [78]: The purpose of this part is strictly to define an architecture for simple geometry. Any implementation details such as ways to define data types and functions, and physical storage in the database are not part of the standard.

    A simple geometry object model and its classes, which correspond to data types, are defined with Unified Modeling Language (UML). The classes include Point, Curve, Surface and GeometryCollection for collections of them (MultiPoint, MultiLineString and MultiPolygon). Moreover, the classes are defined with a number of member functions for:
    * description of the geometric properties of objects, like whether an object is three-dimensional,
    * testing spatial relations between geometric objects, like intersection of objects, and
    * spatial analysis such as distance of objects;
  - Part 2 "SQL option" [79]: This part defines an SQL schema that is used for the management of feature tables, Geometry, and Spatial Reference System information. The purpose of this schema is similar to the role of INFORMATION_SCHEMA, that contains information about the objects defined in a database [95, 50].

    The SQL implementation provides two architectures: one based on primitive data types, for systems that don't have implemented Geometry types, and another based on Geometry types, for systems that have implemented Geometry types. If a database system has implemented Geometry types, then feature tables and Geometry information will be available through INFORMATION_SCHEMA, whereas Spatial Reference System and coordinate dimension are not part of INFORMATION_SCHEMA and cannot be referenced through it.
- KML (formerly Keyhole Markup Language) [76]: Google submitted KML to OGC, so that the OGC consensus handles it evolution, with the goal to become an international standard language for expressing geographic annotation and visualization for web-based and mobile maps (2D) and earth browsers (3D). Under the guidance and the open processes of OGC,

KML has evolved to an important format for the interchange of spatial data [3, p. 144], [89, p. 148].

1.5.2 Industrial Support

A key factor for the success of a standard, both from the industry and the end–user point of view, is its wide adoption. We are going to investigate the adoption of OGC's standards for some of the most well known DB products such as Oracle, MS SQL Server, PostgreSQL and MySQL.

OGC defines two levels of compliance "Implements" and "Compliance" [74]:

Implements This level signifies that the developer of a product has obtained a copy of an OGC standard and has made an attempt to follow its instructions regarding interface or schema syntax and behaviors.

Compliance OGC provides a formal process to test compliance of products with OGC standards. Compliance Testing determines that a specific product implementation complies with all mandatory elements, described in a particular OGC standard, and that these elements operate as described in the standard.

The standard we are interested in is "Simple Features - SQL - Types and Functions v.1.1" that is covered by "Implements" or "Compliance", or unofficially by many database products [75]. We list some of these products alphabetically.

1.5.2.1 MS SQL Server

SQL server is a commercial RDBMS from Microsoft and its latest version added significant spatial support. SQL Server 2008 supports data types and functions according to the OGC standards, even if the product hasn't received a compliance label [53, 51]. It integrates well with other Microsoft products for example with "Visual Earth" [49] for visualization, that is now under the "Bing" product suite [52].

1.5.2.2 MySQL and MariaDB

MySQL is published under dual licence, commercial and GNU GPL, and it is considered the most popular open source RDBMS. It supports spatial extensions for the major storage engines such as `MyISAM`, `InnoDB`, `NDB`, and `ARCHIVE`. The support is not mentioned in any of the OGC product lists. All spatial data types are implemented according to the OGC standard [57]. All the functions are also available, but most of the functions and more importantly, the functions that test spatial relationships between geometries, deviate significantly from the OGC standard. The only, but major difference, is that they operate on Minimum Bounding Rectangles of the geometries, instead of the actual object geometries. The current implementation leaves a lot to be desired and there is an ongoing effort to improve the implementation [56].

MySQL and MariaDB do share the same codebase and are compatible. However, MariaDB offers some advanced features that MySQL doesn't. In contrast with MySQL, MariaDB has support for spatial functions, that operate on the actual geometries and not the MBRs of the geometries.

1.5.2.3 Oracle

Oracle is an advanced and popular commercial RDBMS and offers the extension "Oracle Spatial" which offers spatial data types and functions [85, 84]. Not only this extension has a "Compliance" label from OGC, but Oracle is a member of OGC's Technical Committee [77]. Additionally, Oracle offers further spatial abilities like raster and geo-referenced raster data models, topological data model, medical imaging Digital Imaging and Communications in Medicine (DICOM) data model, and routing solutions.

1.5.2.4 PostgreSQL

PostgreSQL is published under a licence that is similar to the BSD or MIT licenses and is considered the most advanced open source RDBMS. It supports data types for basic geometries [93]. Additionally, PostGIS an extension, published under the GNU GPL licence and developed by Refractions Research, "spatially enables" the PostgreSQL server, allowing it to be used as a back-end spatial database for geographic information systems [92]. PostGIS complies with the data types and functions defined by the OGC standard. It is a leading open-source choice in GIS applications, and among its users there are projects like the EU Joint Research Centre [90] or the French National Geographic Institute (IGN) [91].

Moreover, PostgreSQL plays an central role in the Open Street Maps (OSM) infrastructure [86]. The OSM project has gained a lot of attention for online map solutions. In 2010, one of the most established online map provider, Google, announced that the Google Maps API will no longer be free of charge and that limits will be introduced [26]. This sudden cost increase caused a shift of users and companies to OSM data and related components, including companies like Apple [87] and Flickr [19].

1.6 Outline of the Thesis

The chapters of this thesis are organized in the following way:

- *Chapter 2*: An introduction to R-trees and GiST. The indexes are presented in a detailed way and for each index their properties, and the way search, insertion and deletion are performed, are discussed.
- *Chapter 3*: A number of major R-tree variants are presented. The differences with the original R-tree are discussed and all the details of the original papers are analyzed.
- *Chapter 4*: The discussion about indexes is continued and the MySQL RDBMS is presented in depth. The design of the MySQL the server and the MyISAM storage engine are introduced. Then the current implementation of R*-tree indexes is described in detail.
- *Chapter 5*: The focus is then switched to our own implementation of GiST in MySQL and the design and the challenges of the implementation are discussed.
- *Chapter 6*: This chapter concludes the research by evaluation the project, and suggesting some future improvements.

CHAPTER 2

Preliminaries on R-trees and GiSTs

In this chapter we present background specifically for R-trees and Generalized Search Tree (GiST). The chapter is organized as follows: in Section 2.1 we present the basic properties of Guttman's original R-tree and in Section 2.2 GiSTs are introduced. Finally, in Section 2.3 we summarize the chapter.

2.1 The Original R-tree

The original Guttman's R-tree is described in many textbooks on databases including [106, 99, 121, 24]. However, since the R-tree is central to our research, in this section we are going to briefly recall its basic properties as they are described in [28], [42, pp. 7–12], and [41, pp. 2453–2459].

Guttman proposed the original R-tree in order to solve an organization problem regarding rectangular objects in Very-Large-Scale Integration (VLSI) circuit design. R-trees are hierarchical data structures based on B^+-trees. They are used for the dynamic organization of a set of d–dimensional geometric objects. The property of the objects that is used for the organization is their Minimum Bounding Rectangle. Each node of the R-tree corresponds to the MBR that

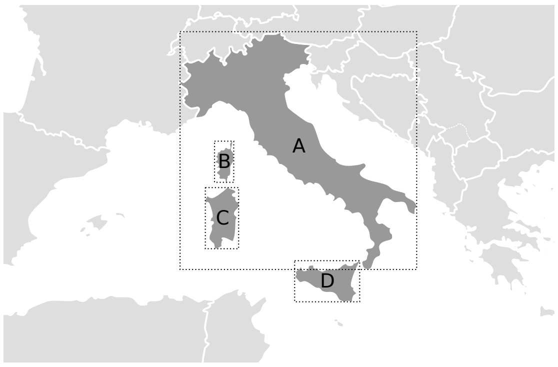


Figure 2.1: The MBRs (dashed rectangles) of the 2–dimensional objects B, C and D intersect with the MBR of object A, whereas the objects themselves do not. Map data from [110].

encloses all its children. Each leaf node, points to one of the objects of the tree.

The R-tree indexing mechanism is used to determine geometric relationships between objects. However, many geometry relationships, such as intersection of complex polygons, can be very demanding computationally, whereas the intersection of rectangles is not a demanding computation. For this reason, R-trees cluster the indexed objects based on the objects' MBRs. It must be noted that MBRs bounding different nodes may overlap, whereas the objects themselves might not overlap. This means that the representation of objects through their MBRs, might result in false positives during search. In order to resolve false positives, the actual geometries of the objects must be examined. Figure 2.1 illustrates such a case where the MBRs of objects B, C and D intersect with the MBR of object A, whereas the objects themselves don't. Therefore, it must be understood that R-trees play the role of a filtering mechanism that reduces the cost of direct examination of geometries.

The rest of the section is organized as follows: in Section 2.1.1, we present the basic properties of the original R-tree. Then, we investigate the details of search in Section 2.1.2, insertion in Section 2.1.3, different splitting methods in Section 2.1.4, and deletion in Section 2.1.5.

2.1.1 Basic Properties

Let M be the maximum number of entries that fit in one node, and let $m \leq \frac{M}{2}$ be a parameter specifying the minimum number of entries in a node. An R-tree

(m, M) satisfies the following properties:

1. Every leaf node contains between m and M entries, unless it is the root.
2. Each entry of a leaf node is of the form (mbr, id), where mbr is the MBR that contains the object and id the object's identifier.
3. Every internal node contains between m and M children, unless it is the root.
4. Each entry of a internal node is of the form (mbr, ptr), where ptr is a pointer to a child of the node and mbr is the MBR that contains all the MBRs, that are contained in this child.
5. The root node has at least two children, unless it is a leaf.
6. All leaf nodes appear on the same level.

The height of a tree is the number of levels within the tree. Let an R-tree containing N entries. Its maximum height is $h_{\max} = \lceil \log_m N \rceil$ [106, p 101].

The maximum number of nodes is $\displaystyle\sum_{i=0}^{h_{\max}} \left\lceil \frac{N}{m^i} \right\rceil$ [42, p 9]

Let an example 2–dimensional R-tree with $(m = 2, M = 3)$, that indexes 17 objects and has three levels. In Figure 2.2 we show the tree structure of the R-tree and in Figure 2.3 we show the spatial representation of the leaf and internal nodes' MBRs. Level three contains the leaf nodes, which hold the identifiers of the indexed objects (numbers 0–16). At the leaf level, 9 leafs are required to store 17 objects if only 2 of the 3 entries are occupied in each leaf node. The MBRs of the indexed objects are represented by the black rectangles 0–16 in Figure 2.3. Levels one and two contain the internal nodes, which hold pointers to children nodes. The MBRs, A1–A3 and B1–B7, of the children nodes are represented by the dashed rectangles in Figure 2.3. In the spatial representation of Figure 2.3, in levels one and two we also represent the MBRs of the leaf nodes, with light gray rectangles, which are obviously not part of these levels. We represent them in order to help the reader relate the internal nodes MBRs with the indexed objects.

2.1.2 Search

The search algorithm descents the tree from the root towards the leaf nodes in a manner similar to a B-tree. However, more than one subtree under one node

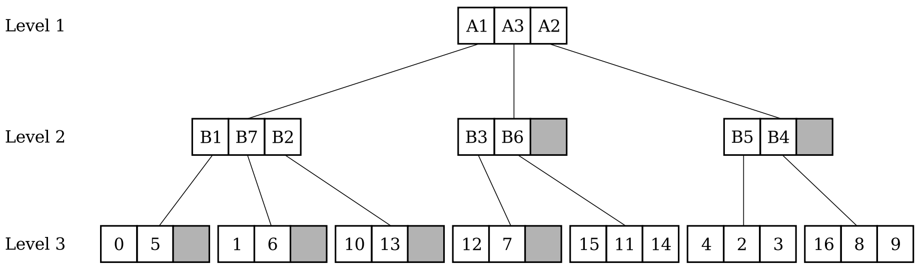


Figure 2.2: Tree structure of the example 2–dimensional R-tree (Section 2.1.1). The spatial representation of the leaf and internal nodes' MBRs are shown in Figure 2.3.

might need to be visited and this consists a problem in the efficiency of the algorithm.

Given an R-tree with root node T and a rectangle S we can form the following query: find all index entries whose MBRs intersect with the search rectangle S. The answer of the query is a set A of objects. This query is called range query and the procedure `RangedSearch`, that processes range queries, is described in Algorithm 2.1.1. The algorithm is called recursively and the initial Node argument is the root node T. All the entries of a node are checked and if an entry's MBR intersects with the search rectangle S, then the algorithm is called on the subtree. As the algorithm descents the tree, if a leaf node is reached all the entries of the leaf node are checked. If an leaf node entry's MBR intersects with S, then the entry is added in the answer set A. For an entry E of a node its MBR of is denoted as $E.mbr$ and the pointer to a child is denoted as $E.ptr$.

2.1.3 Insertion

Insertions in R-trees are handled like insertions in B$^+$-trees. The algorithm descents the tree from the root, in order to locate the appropriate leaf to accommodate the new entry. The new entry is added to the leaf node, and if the node overflows it is split. All the nodes within the path from the root to that leaf are updated recursively.

The method `Insert` handles the insertion and is described in Algorithm 2.1.2. The way a leaf node is found for the new entry (line 1) is handled by `ChooseLeaf` and is described in Algorithm 2.1.3. The overflown nodes are splitted (line 6) with one of the splitting methods presented in Section 2.1.4. Node changes are propagated updards (lines 4, 7) and are handled by `AdjustTree`, described in

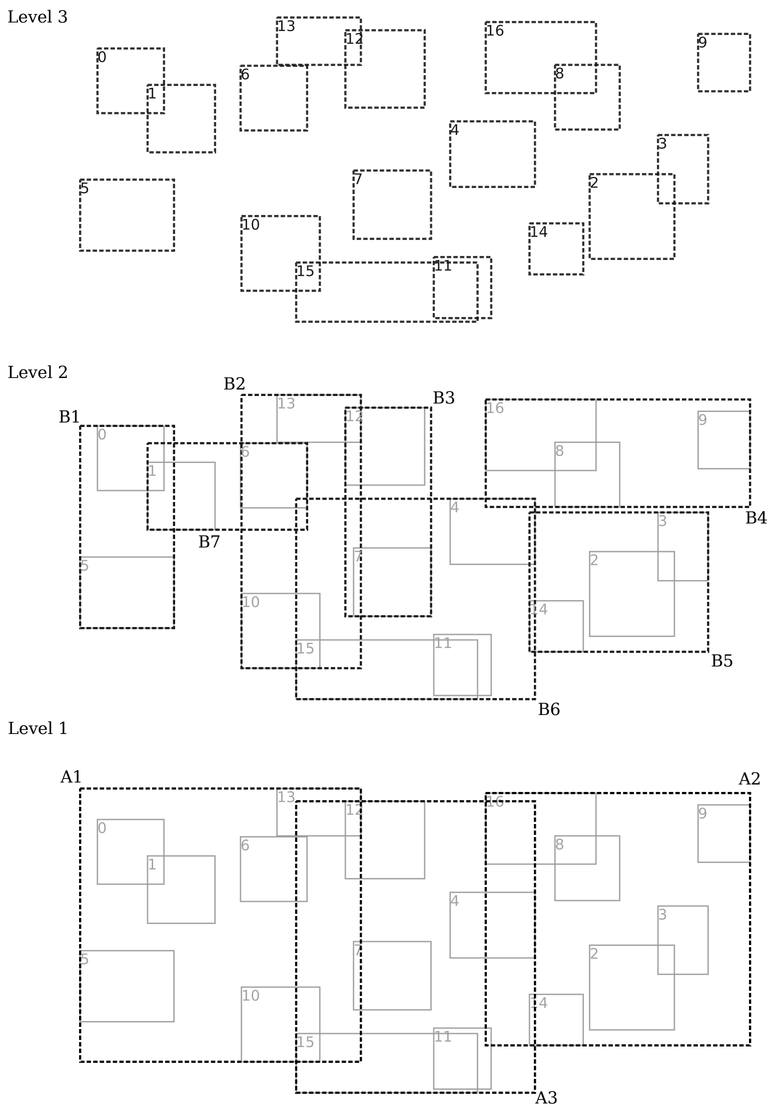


Figure 2.3: Leaf and internal nodes' MBRs spatial representation of the example 2–dimensional R-tree (Section 2.1.1). The tree structure of the example R-tree is shown in Figure 2.2.

```
    **Input**: Node $N$, Rectangle $S$
    **Output**: Set $A$ (index entries whose MBR intersect $S$)
 1  if $N$ is not a leaf node then                    /* Search subtree */
 2      foreach entry $e \in N$ do
 3          if $e.mbr$ that intersects $S$ then
 4              call RangedSearch ($e.ptr$, $S$);
 5  else                                              /* Search leaf node */
 6      foreach entry $e \in N$ do
 7          if $e.mbr$ intersects $S$ then
 8              add $e$ in $A$;
 9
10  return $A$
```

Algorithm 2.1.1: `RangedSearch`(Node N, Rectangle S): R-tree Range Search. Based on the description in [28, p. 49].

Algorithm 2.1.4.

The method `ChooseLeaf`, described in Algorithm 2.1.3, returns the appropriate node N that will accommodate the new entry E. It descents the tree from root to the leaf nodes and in each node finds the entry that requires the minimum area enlargement in order to include $E.mbr$.

During the insertion of a new node two changes can occur: either an overflown node is split, or an entry was added to a leaf node. These changes are propagated upwards by the method `AdjustTree`, described in Algorithm 2.1.4. The method ascends the tree from a leaf or internal node towards the root T and once the root has been reached the algorithm stops. In each level of the tree the MBR of the parent entry of a node N is adjusted to reflect any changes. Then, if a split has occured (line 5), a new entry is added in the parent node of N to accomodate the new node. If there is not enough room in the parent node for a new entry, then the split is propagated upwards (line 11) usinf one of the available splitting methods (Section 2.1.4). Finally, the algorithm prepares to ascend the tree one level.

```
Input: Entry $E$, Node $T$ (root)
Output: Modifies R-tree by adding new entry.

1 $L \leftarrow$ ChooseLeaf ($T$, $E$);   /* Find leaf node for the new entry */
2 if $L$ is not full then                   /* Add entry to leaf node */
3     add $E$ in $L$;
4     AdjustTree ($L, \emptyset$);          /* Propagate changes upwards */
5 else
6     ($L_1$, $L_2$) $\leftarrow$ SplitNode ($L$);
7     AdjustTree ($L_1, L_2$);              /* Propagate changes upwards */
8 if $T$ was split then                     /* Grow tree taller */
9     create new root, and add the old root's split nodes as children;
```

Algorithm 2.1.2: `Insert`(Entry E, Node T): R-tree Insertion. Based on the description in [28, p. 49]

```
Input: Node $N$, Entry $E$
Output: Node $N$ (leaf node where the new entry will be inserted)

1 while $N$ is not leaf node do
2     $K \leftarrow$ entry of $N$ whose $K.mbr$ will require the minimum area
      enlargement in order to include $E.mbr$;
3     Resolve ties by choosing the child whose MBR has the minimum area;
4     $N \leftarrow K.ptr$;
5 return $N$;
```

Algorithm 2.1.3: `ChooseLeaf`(Node N, Entry E): Called by R-tree `Insert` (Algorithm 2.1.2). Based on the description in [28, p. 50].

```
Input: Node $N_1$, Node $N_2$.
Output: Modifies R-tree path starting from the leaf node where a new
        entry was inserted and stopping at the root.

while $N_1$ is not the root $T$ do
    $P \leftarrow$ parent of $N_1$;
    $E_{N_1} \leftarrow N_1$'s entry in $P$;
    Adjust $E_{N_1}.mbr$ so that it tightly encloses all MBRs of $N_1$;

    if split has occurred then                          /* $N_2$ is not $\emptyset$ */
        create new entry $E_{N_2}$, with:
          a) $E_{N_2}.ptr \leftarrow N_2$ and
          b) $E_{N_2}.mbr \leftarrow$ MBR enclosing all MBRs of $N_2$

        if there is room in $P$ then
            add $E_{N_2}$ in P;
        else                                 /* Propagate node split upwards */
            $(K_1, K_2) \leftarrow$ SplitNode ($P$);


    if parent split has occurred then
        $N_1 \leftarrow K_1$;
        $N_2 \leftarrow K_2$;
    else
        $N_1 \leftarrow P$;
        $N_2 \leftarrow \emptyset$;
```

Algorithm 2.1.4: `AdjustTree`(Node N_1, Node N_2): Called by R-tree `Insert` (Algorithm 2.1.2). Based on the description in [28, p. 50].

2.1.4 Node Splitting

Let an R-tree with root node T and a new entry E that needs to be inserted. In order to add a new entry to a full node, that contains M entries, the set of $M + 1$ nodes must be split in two new nodes N_1 and N_2. The objective of the split is to minimize the possibility that the two newly created nodes will both be searched in a future range search query.

During search, the decision whether to visit a node is based on whether the MBR of the node intersects with the search rectangle. This means that after a node is split, the total area of the MBRs of the two new nodes should be minimized. In Figure 2.4 we present an example of bad and good split based on this criterion. Let a 2–dimensional (with $m = 2$, $M = 3$) R-tree and four geometries with MBRs 0–3 (the light gray rectangles). The units used in this example are arbirtary "canvas" units that simply represent the analogies between the lengths. When the fourth geometry is inserted, the root node must be split. The two possible splits are either (0, 1) & (2, 3), or (0, 2) & (1, 3), that corresponds to the (A1, A2) or (B1, B2) MBRs for the the two new nodes. The total area of A1 and A2 is smaller than the one of B1 and B2, meaning that the left split is better than the right one.

Guttman proposed the three following algorithms to handle splits: Exhaustive, Quadratic, and Linear.

Quadratic Split The method `QuadraticSplit`, that describes this splitting technique, is described in Algorithm 2.1.5. Initially, two objects are chosen by `PickSeeds` (Algorithm 2.1.6) as seeds for two new nodes N_1 and N_2, so that these objects create together as much dead space as possible. Let J be the MBR, that bounds both N_1 and N_2, and $N_1.mbr$, $N_2.mbr$ their respective MBRs. Dead space d is the area $d = J - N_1.mbr - N_2.mbr$. For the rest of the remaining objects, the increase $N_1.mbr$ and $N_2.mbr$, if an entry is assigned in one of the nodes, is calculated and the object is assigned to the node, that requires the least enlargement of its MBR.

Method `PickSeeds`, that handles picks two entries of node N based on the dead space, is described in Algorithm 2.1.6. It calculates the inefficiency d of grouping each pair of entries E_1, E_2 of the node, and selects the most wasteful pair.

Linear Split This algorithm is identical to the Quadratic, but uses a different way to select the starting seeds, trying to select two objects that are as far apart as possible from each other. Then, each remaining object is assigned to the

```
Input: Node $N$
Output: Node $N_1$, Node $N_2$
1  $(N_1, N_2)$ ← PickSeeds ($N$);      /* Initialize two nodes with two seeds */
2  while there are unassigned entries do
3      if $N_1$ or $N_2$ has so few entries that the rest must be assigned to it so that it has the required minimum entries then
4          assign them;
5          return
6      foreach unassigned entry $e$ do  /* Select an entry to assign */
7          $d_1$ ← area increase required so that $N_1.mbr$ includes $e$;
8          $d_2$ ← area increase required so that $N_2.mbr$ includes $e$;
9      choose $e$ with maximum difference between $d_1$ and $d_2$;
10     assign it to the node whose MBR has to be least enlarged to include it;
11     Resolve ties by adding the entry:
12        1) to the group with the smaller area;
13        2) to the group with the fewer entries;
14        3) randomly to one of them;
```

Algorithm 2.1.5: QuadraticSplit(Node N): One of the available R-tree splitting methods. Based on the description in [28, p. 52].

```
Input: Node $N$
Output: Node $N_1$, Node $N_2$
   /* Calculate inefficiency of pairs */
1  foreach pair of entries $(E_1, E_2) \in N$ do
2      $J$ ← MBR of $E_1$ and $E_2$;
3      $d$ ← $J.area$ - $E_1.mbr.area$ - $E_2.mbr.area$;
4  choose the pair $(E_1, E_2)$ with the largest $d$;
5  create new empty nodes $N_1$ and $N_2$;
6  $(N_1, N_2) \leftarrow (E_1, E_2)$;
7  return $(N_1, N_2)$
```

Algorithm 2.1.6: PickSeeds(Node N): Called by R-tree QuadraticSplit (Algorithm 2.1.5). Based on the description in [28, p. 52].

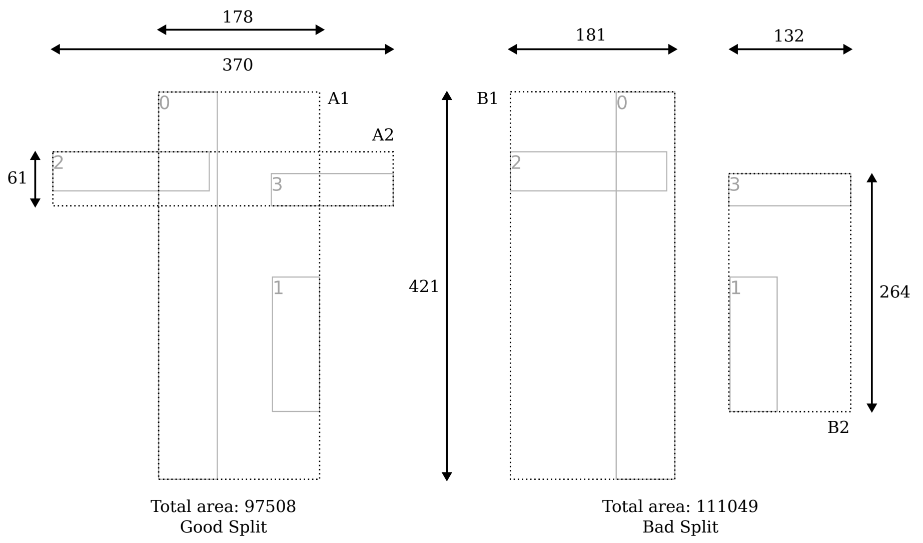


Figure 2.4: The left split is better than the right one because, the total area of the two new nodes is minimized. The units are arbitrary “canvas” units, used for qualitative calculations. Based on [28, p. 51].

node requiring the smallest enlargement of its respective MBR — the order of examination is not important.

Exhaustive Split The most straightforward way to find the minimum area node split is to generate all possible groupings and select the best one. However, the number of possible groupings is 2^{M-1} and with large number of maximum entries in a node the cost of the algorithm becomes prohibiting.

Guttman suggested using the Quadratic algorithm as a good compromise between insertion speed and retrieval performance. Future research and literature on R-trees investigated additional splitting methods and criteria, and in many cases deviated further from the original R-tree.

2.1.5 Deletion

The deletion of an entry from the R-tree is performed by first searching the tree to locate the leaf L that contains the object that needs to be deleted. After the removal of the entry from L, the node may contain fewer entries than m, so the node is underflown.

Handling of underflown nodes is different from B^+-tree, where such an issue is solved by merging two sibling nodes. B^+-trees index one–dimensional data, so two sibling nodes contain “consecutive” entries, whereas R-trees handle multi–dimensional data and this property doesn’t hold. Moreover, merging of nodes is avoided and re-insertion is preferred for the following reasons:

- In order to locate the leaf of the entry that needs deletion, disk was accessed and the path from the root to this leaf might be available in memory. This means that re-insertion might need fewer disk accesses in order to insert the underflown entries.
- The insertion algorithm tries to maintain a good quality of splitting between the nodes. This means that after several deletions, merging of nodes could decrease the quality of the tree, whereas re-insertion ensures it.

Method `Delete` handles the deletion and is described in Algorithm 2.1.7. Finding the leaf containing the entry to delete (line 1) is handled by `FindLeaf` and is described in Algorithm 2.1.8. Underflown nodes (line 5) are handled by `CondenseTree` and the method is described in Algorithm 2.1.9.

```
Input: Entry $E$
Output: Modifies R-tree by removing the specified entry.

1 $L$ ←FindLeaf ($T$, $E$);              /* Find leaf containing entry E */
2 if no match found for $E$ then                /* Entry E not in tree */
3     return 1

4 remove $E$ from $L$;
5 CondenseTree ($L$);                      /* Propagate changes upwards */

6 if $T$ has only 1 child then                          /* Shorten tree */
7     make this child the new root;
```

Algorithm 2.1.7: `Delete`(Node N): R-tree Deletion. Based on the description in [28, p. 50].

Method `FindLeaf`, described in Algorithm 2.1.8, finds the leaf node L that contains the entry E. It descends the tree from the root node T towards the leaf nodes, and is called recursively with initial Node argument the root T. In each level of the tree, the entries of node N are checked and each entry that intersects with $E.mbr$ is checked. In R-trees allow overlapping MBRs for the entries of a node, so more one subtrees of a node might be needed to be checked. Moreover, an entry will be accomodated in only one leaf node, so once we find the entry E, the algorith stops.

```
Input: Entry $E$
Output: Node $L$

1  $L \leftarrow T$;
2  while $L$ is not leaf do                          /* Search subtree */
3      foreach entry $e \in L$ do    /* Entries' MBRs could intersect */
4          if $e.mbr$ intersects $E.mbr$ then
5              $N \leftarrow e.ptr$;
6              FindLeaf ($N$);
7              if FindLeaf returned successfully then
8                  return $L$;
9  foreach entry $e \in L$ do
10     if $e$ matches $E$ then                    /* Found leaf node of E */
11         return $L$;
12 return Null;                         /* Entry E not in this subtree */
```

Algorithm 2.1.8: `FindLeaf`(Node N): Called by R-tree `Delete` (Algorithm 2.1.7). Based on the description in [28, p. 50].

Node elimination is handled by method `CondenseTree`, described in Algorithm 2.1.9. It ascends the tree from a leaf node towards the root T. It propagates upwards key adjustments and underflown node elimination. If root node T is reached the algorithm stops. If N has too few entries, the its entry from the parent node is removed, and it is added to the set of eliminated nodes Q. MBR changes are propagated upwards. Finally, the entries of nodes in Q are re-inserted, at the level of the tree they were removed from, using `Insert` (Algorithm 2.1.2).

```
Input: Node $L$
Output: Modifies R-tree path from root to the leaf where the entry was
        deleted, propagating upwards underflown nodes.

$N \leftarrow L$;
$Q \leftarrow \emptyset$;                              /* Set of eliminated nodes */
while $N$ is not $T$ do
    $P \leftarrow$ parent of $N$;
    $E_N \leftarrow N$'s entry in $P$;

    if $N$ contains less than $m$ entries then
        remove $E_N$ from $P$;
        add $N$ in $Q$;
    if $N$ has not been removed then
        update $E_N.mbr$
    $N \leftarrow P$;

foreach node $q \in Q$ do
    if $q$ was leaf node then
        /* Re-insert leaf nodes normally as leaves                */
        foreach entry $e \in q$ do
            Insert ($e$, $T$);
    else
        /* Re-insert internal nodes as inner nodes                */
        foreach entry $e \in q$ do
            Insert ($e$, $T$, height_flag = True);
```

Algorithm 2.1.9: `CondenseTree`(Node N): Called by R-tree `Delete` (Algorithm 2.1.7). Based on the description in [28, p. 50].

2.2 GiST Trees

In traditional RDBMSes B$^+$-trees are sufficient for the queries posed on alphanumeric data types. On the other hand, new applications, including GIS, multimedia systems and biomedical databases, pushed the research on index trees to accommodate the new challenges. The major approaches are specialized search trees, search trees for extensible data types and abstract search trees.

Specialized search trees Many types of trees were developed to solve specific problems. One such example is R-trees, presented in Section 2.1, that manages to solve spatial range queries well. However, only for R-trees in [42, pp. 4–5] (from 2005) more than 60 variants are reported, meaning that the effort to implement and maintain a good variety of indexing data structures, in an RDBMS, is extremely high.

Search trees for extensible data types An alternative to the creation of new data structures, is to extend the data types they can support [111]. This extension allows the definition of a) new data types, b) new operators for these data types, c) implementation of indexes for these data types and d) instructions, regarding the handling of these data types and indexes, for the query optimizer. In this way for user-defined data types B$^+$-trees can support queries regarding equality and linear range predicates, and R-trees can support queries regarding equality, overlap and containment predicates. However, this method doesn't support the extension of types of queries [29, p. 1] and doesn't solve the difficulty of implementing the new indexes [111, p. 18].

Abstract search trees In [29] and the accompanying technical report [30], Hellerstein, Naughton and Pfeffer presented a third approach for search trees, that extends both the data types and the types of supported queries. This approach uses Generalized Search Tree (GiST), a data structure that provides all the basic search tree logic required by a DBMS, unifying different structures like B$^+$-trees and R-trees.

The rest of the section is based on [29, 30] and is organized as follows: in Section 2.2.1, we present a high altitude view of search trees. In Section 2.2.2, we examine the basic properties of the GiSTs. Then, we investigate the details of search in Section 2.2.3, insertion in Section 2.2.4 and deletion in Section 2.2.5.

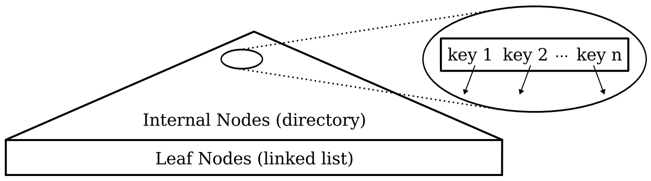


Figure 2.5: Abstraction of a database search tree highlighting its main components: the leaf nodes contain pointers to the actual data, the internal nodes contain pointers to children nodes and keys that hold for each children below the key [29, p. 563].

2.2.1 Abstracting search trees

The idea behind GiSTs is that different search trees can be unified under a single data structure that extends both data types and supported queries. In order to understand this abstraction, it is useful to first review search trees in a simplified manner. The discussion focuses only on the common basic properties of search trees, laying the foundations of a general framework. All the unspecified details will be later filled in, by describing the algorithms of the framework and examples that extend the framework.

A rough abstraction of a search tree is given in Figure 2.5. GiSTs are based on balanced trees with a high fan-out. The leaf nodes contain pointers to the actual data, and they also form a linked list to allow partial or a full sequential scan. The internal nodes are pairs of pointers, to children subtrees, and keys. The way the keys are structured plays a major role in GiSTs. For consistency with [29], the term *predicate* is used as a synonym of the key, implying that something is true or false concerning a quality of the data.

To search for a query predicate q, the search starts at the root node. For each key of a node that, doesn't rule out the possibility that data stored below the pointer matches q, then search traverses this subtree. This property of the key is called *consistency*. The following practical examples will help the understanding of consistency:

- In B^+-trees the queries are in the form of range predicate, like find all the entries e such as $a \leq e \leq b$. In this case the keys dictate whether the data below a pointer match the query. If the query range and a node's key overlap, then the key and query are consistent and the subtree under the node is traversed.
- In R-trees the queries are in the form (in the simple case) of 2-dimensional

range predicate, like find all the entries e such that the region $\big((x_1, y_1), (x_2, y_2)\big)$ intersects with e. The key of a node (the Minimum Bounding Rectangle) dictates that it contains all the keys (the MBRs) of the all children nodes. If the query region and the node's key overlap, the subtree under the node is traversed.

In both these cases the keys are containment attributes, that describe a continuous region in which all the data below the pointer are contained. However, the difference between the two trees is that in R-trees more than one key on the same node may hold simultaneously for a range query. An example can be seen in Figure 2.3, where in level one the MBRs A1 and A3 intersect. If the range query overlaps the intersection of A1 and A3, then both A1 and A3 must be examined.

In GiSTs a key is defined as "any arbitrary predicate that holds for each data below the key" and each subtrees of a GiST represents a partition of data records, but do not necessarily partition the data space itself. In practice a GiST key is a member of a user-defined class, and represents some property that is true of all data items reachable from the pointer associated with the key. The indexed data can be arbitrary data objects. For consistency with [29] we call each indexed datum a *tuple*.

The above ideas form the basis of GiSTs, an abstract data structure for search trees. GiSTs are the base for a framework on search trees, that provides extendibility and a simple way of implementing different trees. The framework exposes methods related to the key definition as well as the handling of overflown and underflown nodes, and the user further defines the inner workings of these methods.

2.2.2 Basic Properties

In this section we present in detail the basic properties of the GiST. It is a balanced tree with a fanout between kM and $\frac{2}{M} \leq k \leq \frac{1}{2}$, where M is the maximum number of elements in a node, and k is the minimum fill factor, a factor defining the minimum number of elements in a node.

A GiST satisfies the following properties:

1. Every node contains between kM and M entries, unless it is the root.

2. Each entry of a leaf node is of the form (p, ptr), where p is a predicate that is used as a search key and *ptr* a pointer the identifier of a tuple in the database. p is true when instantiated with the values from the pointed tuple and this is described as "*p holds* for the tuple".

3. Each entry of an internal node is also of the form (p, ptr), where p is a predicate used as a search key and *ptr* a pointer to a child node. p is true when instantiated with the values of any tuple below *ptr*.

4. The root node has at least two children, unless it is a leaf.

5. All leaf nodes appear on the same level.

Property 3 highlights an important feature of GiSTs. For another entry $E' = (p', ptr')$, below *ptr*, it is simply required that p' and p both hold for all tuples below ptr', whereas the stricter requirement of other trees (like R-tree) $p' \to p$ is not required ($\to$ stands for "implies" in the boolean meaning). An R-tree would require the second because it represents a containment hierarchy.

We should mention here that the original paper, for property 3 states, that it's valid for every tuple *reachable* from *ptr*. We guess that this is a typo and the the authors meant *below ptr*, which is consistent with the rest of the paper.

2.2.2.1 Key Methods

In order to provide to the user a framework to manipulate keys (for insertion, deletion and search), GiSTs provide the key-related methods `Consistent`, `Union`, `Compress`, `Decompress`, `Penalty` and `PickSplit`:

`Consistent`(E, q) given an entry $E = (p, ptr)$ and a query predicate q, this method returns *false* if $p \wedge q$ are definitely unsatisfiable and *true* otherwise. This means that searching the tree can return false positives but never false negatives.

`Union`(P) given a set P of entries $(p_1, ptr_1), \ldots, (p_n, ptr_n)$, this method returns a predicate r that holds for all the tuples stored below $ptr_1, \ldots, ptr_n$. This means that a predicate r that can satisfy all of the predicates $(p_1 \vee \cdots \vee p_n)$ or $(p_1 \vee \cdots \vee p_n) \to r$.

`Compress`(E) given an entry $E = (p, ptr)$, this method returns an entry (p_c, ptr) where p_c is a compressed representation of p.

`Decompress`(E) given an entry $E = (p_c, ptr)$, where $p_c = \texttt{Compress}(p, ptr)$, this method returns an entry (p_d, ptr) so that $p_c \to p_d$. It is not required that $p_c \leftrightarrow p_d$ so the compression method can be a "lossy".

`Penalty`(E_1, E_2) given two entries $E_1 = (p_1, ptr_1)$ and $E_2 = (p_2, ptr_2)$, this method returns a domain specific penalty for inserting E_1 in E_2. This is mainly used to aid the splitting and insertion algorithms, that must have a metric of choosing whether E_1 must be inserted in E_2 or E_3. For example, in R-trees the penalty is the increase in the node's MBR enlargement (see `ChooseLeaf` in Algorithm 2.1.3 and `QuadraticSplit` in Algorithm 2.1.5).

`PickSplit`(P) given a set P of $M + 1$ entries, this method splits P into two sets of entries P_1 and P_2 each of size at least kM. This method is used during the splitting of overflown nodes, orchestrating the Penalty method and the cost of examining the combinations of the $M + 1$ entries.

2.2.2.2 Example

Whereas the previous sections presented the basics of GiSTs, in this section a concrete example is presented. Let an 2D R-tree-based GiST tree, with the MBRs of the indexed data as the key. The key of a node i is represented by the predicate $\texttt{contains}(mbr_i, v)$ where mbr_i the MBR of the node i, and v a free variable.

Let such a tree with an internal node N_{parent} and N_{child} be its child node. In R-trees the organization of the keys is based on a containment hierarchy of the nodes' MBRs. If we recall the properties of an R-tree (Section 2.1.1), properties 2 and 4 dictate that N_{child}'s MBR (p_{child}) must be contained in N_{parent}'s MBR (p_{parent}). This means that $p_{child} \to p_{parent} \Rightarrow \texttt{contains}(mbr_{child}, v) \to \texttt{contains}(mbr_{parent}, v)$.

However, in GiSTs from property 3 (Section 2.2.2) it is simply required that p_{child}, p_{parent} both hold for all nodes N_{below} below N_{child}, or that both $\texttt{contains}$ $(mbr_{child}, mbr_{below})$ and $\texttt{contains}$ $(mbr_{parent}, mbr_{below})$ are true.

R-trees can support many types of predicates and some simple ones include `Contains`, `Equal` and `Overlap`. Also, more complex predicates like the ones mentioned in [88] can be accomodated.

The GiST key methods must be implemented to represent the R-tree properties:

$\texttt{Consistent}(E, q)$ Let an entry $E = (p, ptr)$, q a query predicate on an MBR x, and p the predicate $\texttt{contains}(mbr, v)$ that represents the key of the tree. For any of the query predicates `Contains`, `Equal` and `Overlap` this method returns true if $\texttt{Overlap}(mbr_E, x)$ and false otherwise.

$\texttt{Union}(E_1 \dots E_n)$ returns the MBR of $(E_1 \dots E_n)$.

$\texttt{Compress}(E)$ returns an entry $(E.mbr, E.ptr)$, where $E.mbr$ is the MBR of E.

$\texttt{Decompress}(E)$ in the case of R-trees this method simply returns E. Let x the MBR of E with $x = \texttt{Compress}(E)$. `Decompress` must return an entry (p_d, ptr) so that $x \rightarrow p_d$. The identity function satisfies this property.

$\texttt{Penalty}(E_1, E_2)$ compute $q = \texttt{Union}(E_1, E_2)$ and return $\texttt{area}(q) - \texttt{area}(E_1)$. This is the increase in the node's MBR enlargement (see `ChooseLeaf` in Algorithm 2.1.3 and `QuadraticSplit` in Algorithm 2.1.5).

$\texttt{PickSplit}(P)$ return P splitted in two sets according to `QuadraticSplit` in Algorithm 2.1.5.

2.2.3 Search

GiSTs support two search methods. In Section 2.2.3.1 we present the first search method, that traverses as much of the tree as necessary, descending from the root towards the leaf nodes in a manner similar to a B-tree. In Section 2.2.3.2 we describe the second one, that is useful when the indexed data support linear ordering.

2.2.3.1 General Search

This search method is a general search similar to the search of B-trees and R-trees. Given an GiST with root node T and a predicate q we can form the following query: find all index entries that satisfy q. The predicate q can be either an exact match, or satisfiable by many values in order to support a range query, or even more general predicates not based on contiguous areas in order to support set containment predicates such as all supersets of $\{2, 50, 63\}$. The answer of the query is a set A of objects.

The method `GeneralSearch` is described in Algorithm 2.2.1. It is called recursively, with the the root node T as the initial Node argument. All the entries of a node are checked and if an entry's key p is consistent with with the search predicate q, then the algorithm is called on the subtree. As the algorithm descents the tree, if a leaf node is reached all the entries of the leaf node are checked. If an leaf node entry's if an entry's key p is consistent with with the search predicate q, then the entry is added in the answer set A. For an entry E of a node its key is denoted as $E.p$ and the pointer to a child is denoted as $E.ptr$.

As we have already mentioned, in order to get the final answer of the search the entries of the answer set A must be checked against the predicate q, since GiSTs act as a filtering mechanism. This check can be either performed by the search algorithm or performed by the calling process.

```
Input: Node $N$, Predicate $q$
Output: Set $A$ (index entries that satisfy $q$)

1  if $N$ is not a leaf node then                     /* Search subtree */
2      foreach entry $e \in N$ do
3          if Consistent ($e.p$, $q$) then
4              GeneralSearch ($e.ptr$, $q$);
5  else                                               /* Search leaf node */
6      foreach entry $e$ do
7          if Consistent ($e.p$, $q$) then
8              add $e$ in $A$;
9
10 return $A$
```

Algorithm 2.2.1: `GeneralSearch`(Node N, Predicate q): GiST General Search. Based on the description in [30, pp. 6–8]

2.2.3.2 Linearly Ordered Domains

If the domain of the indexed data offers linear ordering, and queries are usually equality or range containment predicates, then a more efficient search method is possible. The user must make sure the some additional methods and flags (`IsOrdered`, `Compare`, `FindMin`, `Next`) are defined, and that some properties (regarding comparison and overlapping keys) are taken care of:

1. `IsOrdered`: Additional flag, that otherwise defaults to *false*, must be set to *true*. This is a static property of the tree that can only be set during the definition of the tree.
2. `Compare`: Additional method. Given two entries $E_1 = (p_1, ptr_1)$ and $E_2 = (p_2, ptr_2)$, this method returns whether p_1 proceeds, follows or is equally ordered with p_2.
3. `FindMin`: Additional method. It is able to efficiently find the minimum tuple, in the linear order, that satisfies the search predicate q.The method is described in Algorithm 2.2.3.
4. `Next`: Additional method. Returns the next entry on the same level of the tree that satisfies q. The method is described in Algorithm 2.2.4.

Using the functions, flags and properties we mentioned above, equality and range-containment queries can be performed more efficiently with `LinearSearch` than `GeneralSearch` (Algorithm 2.2.1). The method is presented in Algorithm 2.2.2. The search is performed by first using `FindMin`, described in Algorithm 2.2.3, that locates the minimum entry that holds for the search predicate. With this method only one path from root to leaf node will be traversed, unlike `GeneralSearch` that might traverse multiple subtrees. Afterwards, `Next`, presented in Algorithm 2.2.4, is called repeatedly. This method visits only leaf nodes and simply traverses the ordered entries across multiple leaf nodes, until the predicate holds no more.

To find the minimum tuple in linear order, that satisfies the search predicate q, method `FindMin`, described in Algorithm 2.2.3, is used. It descent the leftmost branch of tree and finds the first entry of a leaf node that is `Consistent` with q. It is called recursively and the initial Node argument is root node T. `Consistent` (lines 2 and 8) is described Section 2.2.2.1.

After `FindMin` finds the minimum tuple that satisfies the predicate q, method `Next`, described in Algorithm 2.2.4, is used. This method finds the next entry, in linear order, on the same level of the tree that satisfies q. `Consistent` (lines 3 and 12) is described Section 2.2.2.1.

```
Input: Predicate $q$ (equality and range-containment)
Output: Set $A$ (index entries that satisfy $q$)

$A \leftarrow \emptyset$;
$N \leftarrow$ FindMin ($T$, $q$);            /* First entry that holds for q */
if $N == \emptyset$ then
    return;
add $N$ to $A$;

while true do                               /* All Next entries that hold for q */
    $N \leftarrow$ Next ($N$);
    if $N == \emptyset$ then
        break;
    else
        add $N$ to $A$;
```

Algorithm 2.2.2: `LinearSearch`(Predicate q): GiST Linear Search. Based on the description in [30, pp. 6–8]

```
Input: Node $N$, Predicate $q$
Output: Entry $E$ (minimum leaf node entry that satisfies $q$)

if $N$ is not a leaf node then                          /* Search subtree */
    Find first entry $E$, in linear order, of $N$ so that Consistent ($E$, $q$);
    if such $E$ was found then
        FindMin ($e.ptr$, $q$);
    else
        return $\emptyset$;
else                                                    /* Search leaf node */
    Find first entry $E$, in linear order, of $N$ so that Consistent ($E$, $q$);
    if such $E$ was found then
        return $E$;
    else
        return $\emptyset$;
```

Algorithm 2.2.3: `FindMin`(Node N, Predicate q): Called by GiST `LinearSearch` (Algorithm 2.2.2). Based on the description in [30, pp. 6–8]

```
Input: Node $N$, Predicate $q$, Entry $E$
Output: Entry $E$ (next entry, in linear order, that satisfies $q$)

if $E$ is not the rightmost entry of $N$ then        /* Next on this node */
    $E_{right} \leftarrow$ next entry to the right of $E$;
    if Consistent ($E_{right}$, $q$) then
        return $E_{right}$;
    else
        return $\emptyset$;
else                                                 /* Next on neighboring node */
    $N_{right} \leftarrow$ next node to the right of $N$ on the same tree level;
    if $N_{right} == \emptyset$ then
        return $\emptyset$;

    $E_{right} \leftarrow$ leftmost entry of $N_{right}$;
    if Consistent ($E_{right}$, $q$) then
        return $E_{right}$;
    else
        return $\emptyset$;

```

Algorithm 2.2.4: `Next`(Node N, Predicate q, Entry E): Called by GiST `LinearSearch` (Algorithm 2.2.2). Based on the description in [30, pp. 6–8]

2.2.4 Insert

Insertion in GiSTs is close to the one of R-trees, that resembles the one of B$^+$-trees. It is allowed to insert a node in a specific level of the tree, allowing reuse from other methods. The algorithm descents the tree from root, in order to locate the appropriate leaf to accommodate the new entry. The new entry is added to the leaf node, and if the node overflows it is split. Then upwards from the leaf node, the nodes towards the root are updated.

Let a GiST with root T, a new entry E, a desired tree level l. Moreover, for an entry E, $E.p$ denotes the predicate of the node and $E.ptr$ denotes pointer to the children node. Method `Insert` is described in Algorithm 2.2.5. Finding the lead node that will accommodate the new node (line 1) is handled by method `ChooseSubtree` (Algorithm 2.2.6). For domains that support linear ordering, `Compare` (line 4) can be used (Section 2.2.3.2). Method `Split` (line 8) handles overflown nodes (Algorithm 2.2.7). Finally `AdjustKeys` (line 9) propagates key changes upwards (Algorithm 2.2.8).

```
Input: Node $T$ (root), Entry $E$, Level $l$
Output: Modifies GiST by adding new entry $E$

1 $L \leftarrow$ ChooseSubtree $(T, E, l)$;        /* Find node where E will be inserted */
2 if $L$ is not full then                  /* Add entry to leaf node */
3     if IsOrdered then
4         add $E$ in $L$ according to Compare;
5     else
6         add $E$ in $L$;
7 else
8     Split $(L, E)$;
9 AdjustKeys $(L)$;                        /* Propagate changes upwards */
```

Algorithm 2.2.5: `Insert`(Node N, Entry E, Level l): GiST Insertion. Based on the description in [30, pp. 8–10]

`ChooseSubtree` (Algorithm 2.2.6) descents the tree trying to find the appropriate node that will accommodate the inserted node, by using method `Penalty` (line 5), that is described in Section 2.2.2.1). The method is called recursively and the initial argument is the root node T.

Method `Split`, described in Algorithm 2.2.7, chooses how to split the node N.

```
Input: Node $N$, Entry $E$, Level $l$
Output: Node at level $l$

1 if $N$ is at level $l$ then
2     return $N$;
3 else
4     foreach entry $e \in N$ do
5         Penalty $(e, E)$;
6     $K \leftarrow$ entry $e$ with the minimum penalty;
7     $N \leftarrow$ ChooseSubtree $(K.ptr, E, l)$;
8     return $N$;
```

Algorithm 2.2.6: `ChooseSubtree`(Node N, Entry E, Level l): Called by GiST `Insert` (Algorithm 2.2.5). Based on the description in [30, pp. 8–10]

First, method `PickSplit` (line 1) splits the keys of node N and the new entry E in two nodes. The first node node is put directly in N, and the second is inserted in the parent node. If there is room in the parent node, then an entry pointing to the second node is added. In case the domain is linearly ordered then `Compare` (line 8), described in Section 2.2.3.2, is used for the addition. If the parent node is full, the splitting is propagated upwards. In all the cases a node has changed and the key of its entry in the parent node must be updated (lines 3 xand 14) `Union` is used (described in Section 2.2.2.1).

Method `AdjustKeys`, described in Algorithm 2.2.8, ascends tree from node N and makes all predicates of the nodes accurate characterizations of their subtrees. It stops once the root T is reached or when a predicate is already accurate.Method `Union` (line 5), described in Section 2.2.2.1, is used to calculate the predicate u that holds for all tuples stored under node N.

2.2.5 Delete

The deletion is similar to the one of B$^+$-trees and R-trees. Method `Delete` is presented in Algorithm 2.2.9. It finds and removes the entry to be deleted and propagates upwards key changes and possible elimination of underflown nodes. The entry to be delete is located with a generic or linear `Search` (line 1) presented in Section 2.2.3. Propagation of key changes and handling of underflown is performed by method `CondenseTree` (line 5) described in Algorithm 2.2.10.

`CondenseTree`, described in Algorithm 2.2.10, ascends the tree from node N and

```
Input: Node $N$, Entry $E$
Output: Modifies GiST by splitting $N$ and adding new entry $E$

1  $(N, N') \leftarrow$ PickSplit $(N \cup \{E\})$;
2  $E_{N'} \leftarrow (q, ptr')$ where:
3    $q \leftarrow$ Union $(N')$;
4    $ptr'$ pointer to $N'$;

5  $P \leftarrow$ Parent $(N)$;
6  if there is room in $P$ then              /* Insert $E_{N'}$ in parent node */
7      if IsOrdered then
8          add $E_{N'}$ in $P$ according to Compare;
9      else
10         add $E_{N'}$ in $P$;
11 else
12     Split $(P, E_{N'})$;

13 $K \leftarrow$ entry of $P$, where $K.ptr$ points to $N$;
14 $K.p \leftarrow$ Union $(N)$;
```

Algorithm 2.2.7: `Split`(Node N, Entry E): Called by GiST `Insert` (Algorithm 2.2.5). Based on the description in [30, pp. 8–10]

```
Input: Node $N$
Output: Modifies GiST so that ancestors of $N$ contain correct keys

1  if $N$ is the root then
2      return;
3  else
4      $E \leftarrow$ entry of $P$, where $E.ptr$ points to $N$;
5      $u \leftarrow$Union $(N)$;

6      if $E.p$ is as accurate as $u$ then
7          return;
8      else
9          $E.p \leftarrow u$;
10         AdjustKeys (Parent $(N)$);
11         return;
```

Algorithm 2.2.8: `AdjustKeys`(Node N): Called by GiST `Insert` (Algorithm 2.2.5). Based on the description in [30, pp. 8–10]

```
Input: Entry $E$
Output: Modifies GiST by deleting entry $E$

1 $L \leftarrow$ Search $(T, E.p)$;                         /* Find node */
2 if $L == \emptyset$ then                             /* Entry E not found */
3     return $\emptyset$;

4 Remove $E$ from $L$;
5 CondenseTree $(L)$;

6 if $T$ has only 1 child then                          /* Shorten tree */
7     make this child the new root;
```

Algorithm 2.2.9: `Delete`(Node N): GiST Deletion. Based on the description in [30, pp. 10–11]

makes the predicates of the nodes accurate characterizations of the subtrees. It stops once the root T is reached or when a predicate is already accurate. In the end orphaned entries are re-inserted like in R-trees.

```
Input: Node $N$
Output: Modifies GiST so that ancestors of $N$ contain correct keys

$N \leftarrow L$;
$Q \leftarrow \emptyset$;                                   /* Set of eliminated nodes */
while $N$ is not $T$ do
    $P \leftarrow$ Parent ($N$);
    $E_N \leftarrow$ $N$'s entry in $P$;

    if $N$ contains less than $kM$ entries then
        if IsOrdered then
            $N' \leftarrow$ neighboring node in order;
            if number of entries in $N$ and $N' \geq 2kM$ then      /* Try to
            borrow entries */
                split evenly the entries between $N$ and $N'$;
            else                                        /* Merge with neighbor */
                put entries of $N$ in $N'$;
                remove $E_N$ from $P$;
                AdjustKeys ($N'$);
                AdjustKeys ($P$);

        else                                                    /* Remove Node */
            add $N$ in $Q$;
            remove $E_N$ from $P$;
            AdjustKeys ($P$);

    if $E_N$ was removed from $P$ then
        $N \leftarrow P$;
    else
        AdjustKeys ($N$);
        break;

foreach node $N \in Q$ do                       /* Re-insert orphaned entries */
    foreach entry $e \in N$ do
        Insert ($e$,Level (e));
```

Algorithm 2.2.10: CondenseTree(Node N): Called by GiST Delete (Algorithm 2.2.9). Based on the description in [30, pp. 10–11]

2.2.6 GiSTs in Postgres

Postgres' GiST Application Programming Interface (API) and the functions the user has to implement to use this API is described by the manual in [94]. According to the source file `backend/src/access/gist/README` Postgre's GiST is very close to the original [29]. The implementation has solved concurrency issues and lately improved recovery-related issues. However, as the developers commented in the "pgsql-hackers" mailing list [96] the information in the file is in general correct but might not completely reflect the status of the implementation.

The C API is defined in `src/include/access/gist.h` and implemented in `src/backend/access/gist/`. The functions are registered in the system as built-in SQL functions (`src/include/catalog/pg_proc.h`) and are hooked in `src/backend/access/gist/gist.c`. It will be interesting to investigate this implementation in detail, since it has been in production since 2005 [97] but due to time constraints we couldn't go in the source code.

According to the R-tree implementation provided by Postgres (`src/backend/access/gist/gistproc.c`) the following functions are defined in order to use the GiST API:

- `same` returns true if the 2 input geometries are equal. This function is not mentioned in the original GiST framework, but is needed widely in the implementation.
- `consistent` for a query predicate (or as named in the source code "query operator") this function returns false if for all the data indexed below an entry if the qury predicate is false.
- `union` Given a set of entries, this function generates a new index entry that represents all the given entries.
- `penalty` calculates the cost of inserting the new entry in an a node.
- `picksplit` a method to split an overflown node.
- `compress` prepares the physical storage of the key in an index page. In the case of R-trees the MBR of the indexed datum is the key and is already considered as "compressed".
- `decompress` converts the stored representation of the data item into a format that can be manipulated by the database. In the case of R-trees the key of the indexed datum is its MBR and the system is already capable of handling the data structure, so it doens't need "decompression".

2.3 Summary

In this chapter we presented the spatial index R-tree and the abstract search tree GiST. For both indexing solutions we first discussed their basic properties. Then we described how search, insertion and deletion are performed and the details of the algorithms that drive these actions. Moreover, we took a look at the splitting of tree nodes that are full and joining tree nodes that are filled below their fill threshold.

CHAPTER 3

Dynamic R-tree versions

The R-tree data structure is a major spatial indexing solution. A survey from Gaede and Guenther [21] and the one of the book [42], that serves as one of our main sources of reference, describe a large number of dynamic R-tree variants. This chapter focuses on a number of these dynamic variants where the spatial objects are inserted on a one-by-one basis. For each, their structure, indexing, splitting and querying techniques are examined in detail.

Six variations of the original R-tree are investigated. In Section 3.1, the R$^+$-tree variant is presented. Then, we present the R*-tree variant in Section 3.2, and the Hilbert R-tree in Section 3.3. Two splitting algorithms are then introduced, the linear splitting in Section 3.4, and the optimal splitting in Section 3.5. Finally, VoR-Tree a variant for nearest neighbor queries is described in Section 3.6.

3.1 R$^+$-tree

The original R-tree based its search performance on two factors, that could easily create performance problems:

- *minimal overlap*: during insertion a new node is inserted in the path that causes the minimum area enlargement. This factor the most critical.

- *minimal coverage*: during split of overflown nodes, the two new nodes should have as much as empty space between them as possible.

Moreover, if only a few large rectangles are inserted, the overlap of internal can increase significantly and decrease search performance.

Sellis, Roussopoulos and Faloutsos proposed the R$^+$-tree data structure in [103], whose major goal was to provide not just minimal, but zero overlap. In the R-tree structure each entry is accommodated in only one node, whereas the R$^+$-tree allows the splitting of a node, in order to avoid overlap of internal nodes.

An example of the main idea behind the R$^+$-tree is given in Figure 3.1. Let four example objects (the gray rectangles 0–3) that are inserted in a $(2,3)$ R-tree (left column) and a $(2,3)$ R$^+$-tree (right column). The dashed rectangles (A1, A2, B1, B2) represent the MBRs of the internal nodes of each tree. For consistency with [103], the term *data rectangle* is used to "denote a rectangle that is the MBR of an object" as opposed to rectangles that correspond to the intermediate nodes of the tree. Whenever a data rectangle overlaps with a rectangle of a higher level, it is decomposed in non-overlapping sub-rectangles. The union of these sub-rectangles is the original rectangle. In our example, object 3 causes a problem in the minimum overlap factor of the R-tree, making nodes A1 and A2 to overlap. In order to have zero overlap between the nodes, object 3 is decomposed in two sub-rectangles B1 and B2 that have zero overlap. In the R$^+$-tree the data rectangle of object 3 is located in two leaf nodes.

R$^+$-trees are balanced trees and their leaf and intermediate nodes have the same form as in R-trees. They satisfy the following properties:

1. Each entry of an intermediate node is of the form (mbr, ptr), where ptr is a pointer to a child node and mbr is the MBR that contains *completely* all the MBRs of this child.

2. For two entries (mbr_1, ptr_1) and (mbr_2, ptr_2), of an intermediate node, there is zero overlap between mbr_1 and mbr_2.

3. Each entry of an leaf node is of the form (mbr, id), where mbr is the MBR that contains the object and id the object's identifier. The leaf's entry mbr is *not* required to be *completely* contained in the parent's entry mbr.

4. The root node has at least two children, unless it is a leaf.

5. All leaf nodes appear on the same level.

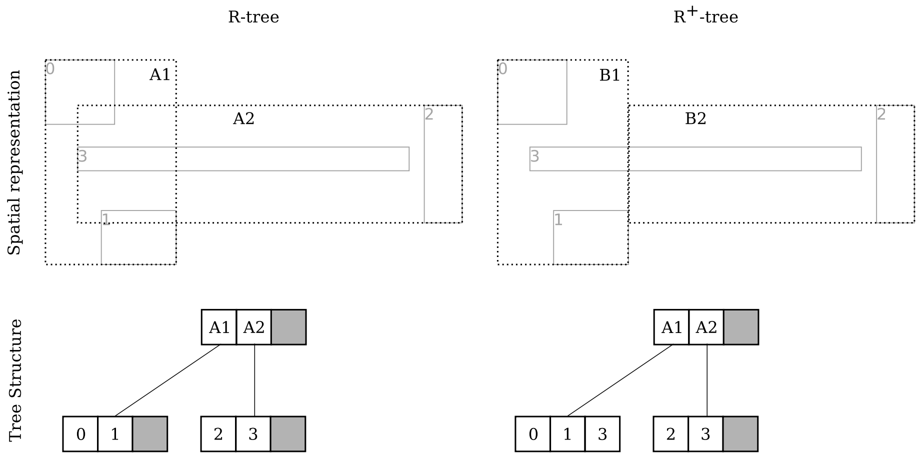


Figure 3.1: R-tree overlapping and R$^+$-tree decomposition of MBRs. Top row: Leaf and internal nodes' MBRs spatial representation of trees. Bottom row: tree structure of the tree above. Left column: R-tree. Right column: R$^+$-tree.

The rest of the section is organized as follows: in Section 3.1.1, we present how search is performed. In Section 3.1.2, insertion is described. In Section 3.1.3, splitting is presented, and in Section 3.1.4, partitioning is introduced. In Section 3.1.5, packing is discussed briefly. Finally, in Section 3.1.6, we outline the basics of deletion.

We should note that the authors of the paper made an error in the definition of the the format of nodes [103, p. 511]. They mention the "form of the leaf nodes" and the "form of the internal node", but instead they mean the form of *entries* of the leaf and internal nodes accordingly. These terms, node and entry of a node, also get confused in the definition of the `SplitNode` algorithm in [103, pp. 513–514].

3.1.1 Search

The search is described in Algorithm 3.1.1. The space is **already** decomposed in disjoint sub-regions. The method descents the tree from root to leaf nodes and in each level checks the subtrees of the entries, whose MBRs intersect with the search area S. It is called recursively with initial Node argument the root T. The procedure differs to the insertion of R-trees (Algorithm 2.1.1) in line 4, where only the search area is clipped as the algorithm goes to the level below. Also in line 8, duplicates must be eliminated from the answer set either in this method or by the caller of the search.

```
Input: Node $N$, Rectangle $S$
Output: Set $A$ (index entries whose MBR intersect $S$)

1  if $N$ is not a leaf node then                        /* Search subtree */
2      foreach entry $e \in N$ do
3          if $e.mbr$ that intersects $S$ then
4              call Search ($e.ptr, S \cap e.mbr$);
5  else                                                  /* Search leaf node */
6      foreach entry $e \in N$ do
7          if $e.mbr$ intersects $S$ then
8              add $e$ in $A$ ;                          /* Avoid duplicates */
9
10 return $A$;
```

Algorithm 3.1.1: Search(Node N, Rectangle S): R$^+$-tree Search. Based on description in [103, p. 512].

3.1.2 Insert

Insertion is handled by method `Insert` described in Algorithm 3.1.2. A new entry E is inserted in an R$^+$-tree, by performing a recursive search on the tree and adding the entry in the leaf nodes. The initial node argument is the root node T. Unlike the case of an R-tree, the new entry might be added in more than one leaf nodes and the MBR of the new entry is decomposed in sub-regions in the internal nodes. Method `SplitNode` (line 8) handles overflown nodes by re-organizing the tree. Splitting is described in Section 3.1.3.

Moreover, we should note that the `if` clause in line 3 doesn't have a corresponding `else` clause, even if a new entry could not intersect with existing node's MBRs. This implies a decomposition of the whole space, during the creation of the tree similar to the K-D-B-trees [100].

3.1.3 Split

Method `SplitNode`, presented in Algorithm 3.1.3, handles overflown nodes by re-organizing the tree. In line 2 method `Partition`, described in Algorithm 3.1.4,

```
**Input**: Entry $E$, Node $N$ (root)
**Output**: Modifies R$^+$-tree by adding new entry.
1 if $N$ is not a leaf node then                    /* Search subtree */
2     foreach entry $e \in N$ do
3         if $e.mbr$ intersects $S$ then
4             call Insert ($e.ptr, E.mbr$);
5 else                                              /* Search leaf node */
6     add $E$ in $N$;
7     if $N$ has $M + 1$ entries then
8         SplitNode ($N$);                          /* Re-organize tree */
9
```

Algorithm 3.1.2: `Insert`(Entry E, Node N): R$^+$-tree Insertion. Based on description in [103, p. 512].

is used to find two mutually disjoint partitions for the node N. Even if the method returns a Node and a Set of entries, both returned data structures are used as sets of entries. Their MBRs are used to initialize two new empty nodes, and then their entries are then divided to the node that covers them completely (lines 8 and 10). If an entry intersects with both partitions then if the algorithm is on a leaf node the entry is placed in both nodes. Otherwise the splitting is propagated downwards `SplitNode` on the subtree. In the end, node splitting changes are propagated upwards.

Downwards propagation of splitting is required due to the property 1 of R$^+$-trees (Section 3.1), as children nodes might need to be split. Such a case is demonstrated in Figure 3.2. Node A1 is the parent of node A2, and A2 is the parent of Node A3. The tree structure is presented on the right and the spatial representation of the nodes on the left. If node A1 has to be split, then its children might also need to be split. In this example, if the partition line crosses all three children, then all of them need to be checked for splitting.

3.1.4 Partition

Partitioning is used to decompose the space of a node in non-overlapping sub-regions. In this section we present the algorithms for two dimensions, however their generalization is straight-forward.

```
Input: Node $N$
Output: Modifies R$^+$-tree by splitting overflown nodes.

$S \leftarrow$ set of all entries in $N$;
$(K, S') =$ Partition $(S, f)$;
$S_1, S_2 \leftarrow$ 1st, 2nd sub-regions of partition;
$(N_1, N_2) \leftarrow (\emptyset, \emptyset)$ ;                     /* New empty nodes */
$E_{N_1} \leftarrow (N.mbr \cap S_1.mbr, N_1)$ ;        /* Entries pointing to them */
$E_{N_2} \leftarrow (N.mbr \cap S_2.mbr, N_2)$ ;        /* with initialized MBRs    */

foreach entry $e$ of $N$ do
    if $e.mbr$ completely in $E_{N_1}.mbr$ then
        add $e$ in $N_1$;
    else if $e.mbr$ completely in $E_{N_2}.mbr$ then
        add $e$ in $N_2$;
    else                              /* Partially in either of them */
        if $N$ is leaf node then
            add $e$ in both nodes;
        else                                        /* Internal node */
            $(K_1, K_2) \leftarrow$ SplitNode $(e.ptr)$ ;         /* Split subtree */
            add $K_1$ and $K_2$ as children in nodes $N_1$ and $N_2$, depending in
            which of $N_1$ and $N_2$ they are included completely;

if $N == T$ then                          /* Propagate changes upwards */
    create new root with children $N_1$ and $N_2$;
else
    $P \leftarrow$ parent node of $N$;
    $e_p \leftarrow$ entry of $N$ in $P$;
    remove $e_p$ from $P$;
    add entries pointing to $N_1$ and $N_2$;
    if $P$ has more then $M$ entries then
        SplitNode $(P)$;
```

Algorithm 3.1.3: SplitNode(Entry E, Node N): R$^+$-tree Splitting. Called by Insert described in (Algorithm 3.1.2). Based on description in [103, p. 513].

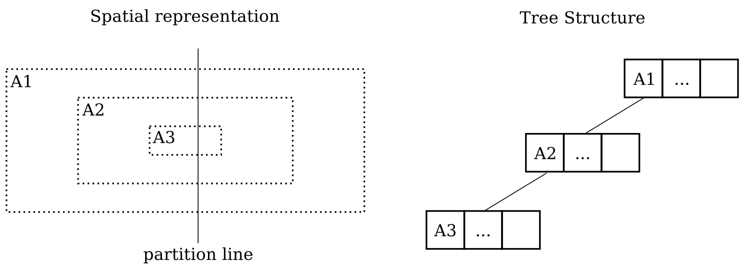


Figure 3.2: R$^+$-tree downwards propagation example [103].

The 2-dimensional space is divided in two sub-regions using one of the available axis. The criteria on which the axis is chosen are:

1. nearest neighbors,
2. minimal total axis displacement,
3. minimal total space coverage due to the new sub-regions, and
4. minimal number of entry splits.

The first three criteria help search performance by reducing coverage of dead space, whereas the fourth limits the tree height.

The method that handles partitioning is `Partition` described in Algorithm 3.1.4. Beginning from the lowest point of the set (l_x, l_y), `Sweep` (line 6) scans each of the available axes. This method is described in Algorithm 3.1.5 and returns the cost of splitting each axis. The overall minimum cost is calculated according to one or a combination of the above mentioned criteria, and the axis that has this cost is used for the portioning. The two sub-regions define one node and one set, each containing all the nodes of N that fall in each sub-region.

`Sweep`, described in Algorithm 3.1.5, scans an axis a to find the partitioning point *cut*. It begins scanning the axis from the point l and it collects the first f elements from the given set of rectangles S. The authors mention that the set S is sorted, but they don't define how this sorting is performed, so we assume that they mean ordering by the value on the axis a. The value, on axis a, of the last element that is inserted is the point *cut* (line 3).

Another error that appears in the paper is that the authors mention that *cut* gets the largest value, from one of the axes, of the f entries. We believe they mean the largest value of the axis that is *currently* scanned, since the partitioning of

```
Input: Set of rectangles $S$, FillFactor $f$
Output: Node $N$, Set of rectangles $S'$

1  $N \leftarrow \emptyset$;
2  if $S$ contains $\leq f$ elements then        /* No partition required */
3      add all elements of $S$ in $N$;
4      return $(N, \emptyset)$;
5  $(l_x, l_y) \leftarrow$ Lowest $x$ and $y$ coordinates of all elements of $S$;
6  $(C_x, cut_x) \leftarrow$ Sweep $(x, l_x, f, S)$  $(C_y, cut_y) \leftarrow$ Sweep $(y, l_y, f, S)$;
7  $(C_{min}, cut_{min}) \leftarrow$ smallest cost and the corresponding cut point;
   /* Now $cut_{min}$ divides S is two sub-regions                    */
8  $N \leftarrow$ all elements of $S$ that fall in 1st sub-region;
9  $S' \leftarrow$ set of all elements of $S$ that fall in 2nd sub-region;
10 return $(N, S')$;
```

Algorithm 3.1.4: `Partition`(Set of rectangles S, FillFactor f): R$^+$-tree Partitioning. Called by `SplitNode` (Algorithm 3.1.3). Based on description in [103, p. 514].

the node selects one axis and *cut* is the point where the partitioning is performed. The value of other axis might be outside the range of values available for the axis that is currently scanned.

`Cost` (line 3) calculates the cost C for this partitioning point, according to one or a combination of the above mentioned criteria. This implementation is not presented and is left for the implementation of the tree.

3.1.5 Pack

The packing algorithm re-creates the tree, in order to improve its search performance, that could degrade, as nodes are inserted and deleted. The interested reader can find its description in [103], as well as in [101] that discusses this packing method in detail.

```
Input: Axis $a$, Point $l$, FillFactor $f$, Set of rectangles $S$
Output: Cost $C$, Point $cut$
1 $G \leftarrow \emptyset$ ;                                  /* Set of first f elements */
2 starting from point $l$, add to $G$ the $f$ largest elements of $S$ on axis $a$;
3 $cut \leftarrow$ the value of the largest element added in $G$, for axis $a$  $C \leftarrow$ Cost
  $(G)$ ;                                 /* Cost of measured property */
4 return $(C, cut)$;
```

Algorithm 3.1.5: `Sweep`(Set of rectangles S, FillFactor f): R$^+$-tree Partitioning. Called by `Partition` described in (Algorithm 3.1.4). Based on description in [103, p. 515].

3.1.6 Delete

The deletion algorithm is similar to the one of R-trees. The difference is that an indexed object might be present in more than one leaf nodes, so it has to be removed by all of them. The algorithm is described in [42, p. 17]

3.2 R*-tree

In 1990, Beckmann, Kriegel, Schneider and Seeger proposed an R-tree variant the R*-tree [7]. It is very close to Guttman's data structure (Section 2.1 and [28]), but offers a more engineered approach when it comes to choosing the insertion path and the splitting procedure. The algorithm is currently implemented in Oracle [118] and SQLite [109], and is still considered in the literature as a "prevailing performance-wise structure often used as a basis for performance comparisons" [42, p. 18]. In Section 3.2.1 choosing the appropriate insertion path is described, in Section 3.1.3 the splitting of overflown nodes is presented and finally in Section 3.2.3 the re-insertion procedure is analyzed.

The criteria considered for insertion path choosing and reinsertion are the following:

- Minimization of the area covered by MBRs: this factor is the only one also considered in the original R-tree. The goal is to minimize the dead space, the area of an node's MBR that is not covered by its children nodes

MBRs.

- Minimization of overlap covered by MBRs: the goal is to minimize the expected number of paths followed by a range query.

- Minimization of MBR margins: margin is defined as the sum of the lengths of the edges of an MBR. The goal is to shape the MBRs as quadratic as possible. This also improves the packing of the nodes making the MBRs of upper levels of the tree smaller, thus achieving indirectly minimization of the area.

- Maximization of node utilization: the higher the node utilization the less nodes will be read from disk during query processing.

In their paper they state that they tested different combinations of the above mentioned criteria to find which one is the preferable to choose an appropriate insertion path. They concluded that the best results are given when the overlap is defined when minimization of the area covered by MBRs is taken into account [7, p. 325].

3.2.1 Insertion path

Since R-tree is a dynamic data structure, the insertion of new entries plays an important role in its performance. The first issue Beckmann, Kriegel, Schneider and Seeger try to improve is the insertion strategy and the way the appropriate insertion path is chosen.

Method `ChooseSubtree`, described in Algorithm 3.2.1 returns the appropriate node N that will accommodate the new entry E. It descents the tree from root to the leaf nodes and it is similar to `ChooseLeaf`, described in Algorithm 2.1.3. The main difference is that it uses different methods to determine the insertion path. If the node N, that is currently examined, has children that are leaf nodes (line 2), the method finds the entry that requires the minimum *overlap* enlargement in order to include $E.mbr$. If the node N, that is currently examined, has children that are non leaf nodes (line 5), the method finds the entry that requires the minimum *area* enlargement in order to include $E.mbr$.

Moreover, in their paper, the authors offer a method of finding the *nearly* minimum overlap for trees with a large number of entries per node, in order to achieve smaller CPU cost.

```
Input: Node $N$, Entry $E$
Output: Node $N$ (leaf node where the new entry will be inserted)

1 while $N$ is not leaf node do
2     if children of $N$ entries are leaf nodes then    /* determine the minimum overlap */
3         $K \leftarrow$ entry of $N$ whose $K.mbr$ will require the minimum overlap enlargement in order to include $E.mbr$;
4         Resolve ties by choosing the child whose MBR has the minimum area;
5     else                        /* determine the minimum area cost */
6         $K \leftarrow$ entry of $N$ whose $K.mbr$ will require the minimum area enlargement in order to include $E.mbr$;
7         Resolve ties by choosing the child whose MBR has the minimum area;
8     $N \leftarrow K.ptr$;
9 return $N$;
```

Algorithm 3.2.1: `ChooseSubtree`(Node N, Entry E): Called by R-tree and R*-tree `Insert` (Algorithm 2.1.2 - `ChooseLeaf`). Based on description in [7, p. 324].

3.2.2 Splitting

The splitting method tries to split an overflown node in 2 new nodes in a good way. In order to decide the where the split will occur it examines the different grouping of all the entries of the node. We should remind the notation used to describe the properties of R-trees: M is the maximum entries a node can hold, and m, with $2 \leq m \leq M$, is the minimum entries a node can hold. The grouping is performed by sorting the entries, and creating $M-2m+2$ distributions of two groups. For the k-th distribution the first group contains the first $(m-1)+k$ sorted entries and the second the rest entries.

Let an example 2-dimensional R*-tree with $(m = 2, M = 5)$, with an overflown node of $M + 1 = 6$ entries. The spatial representation of the entries' MBRs and the sorting of the entries by upper and lower value for axis X are shown in Figure 3.3. For each sorting there are three distributions. The distributions, for the sorting of upper values for axis X, are also shown in Figure 3.3. In the first distribution the first group contains the first 2 entries and the second group the remaining, in the second distribution the first group contains the first 3 entries and the second group the remaining, and in the third distribution the first group contains the first 4 entries and the second group the remaining.

The method `ChooseSubtree` implements the splitting of an overflown node. It first calls `ChooseSplitAxis` (line 1) to determine the axis on which the split will occur and then calls `ChooseSplitIndex` (line 2) to determine the two new groups that will be created.

```
Input: Node $N$ (the overflown node)
Output: Node $A$, Node $B$ (the result of the spitting)

1 $axis$ ← ChooseSplitAxis;
2 $A, B$ ← ChooseSplitIndex ($axis$);
3 Distribute the entries in two groups $A, B$;
4 return $A, B$;
```

Algorithm 3.2.2: `Split`(Node N): R*-tree splitting. Called by `OverflowTreatment` (Algorithm 3.2.7). Based on description in [7, p. 326].

The method `ChooseSplitAxis` is called by `Split` (Algorithm 3.2.2) and picks the axis perpendicular to which the split will occur. It examines all the available axis and creates two sortings: by lower and by upper value of this axis (lines 2–3). Then it finds all the available distributions of the entries of the node (line 4), , as it was explained in the introduction of this section (3.2.2 - page 58). Finally it picks the axis where the sum of margins of all its distributions is minimum.

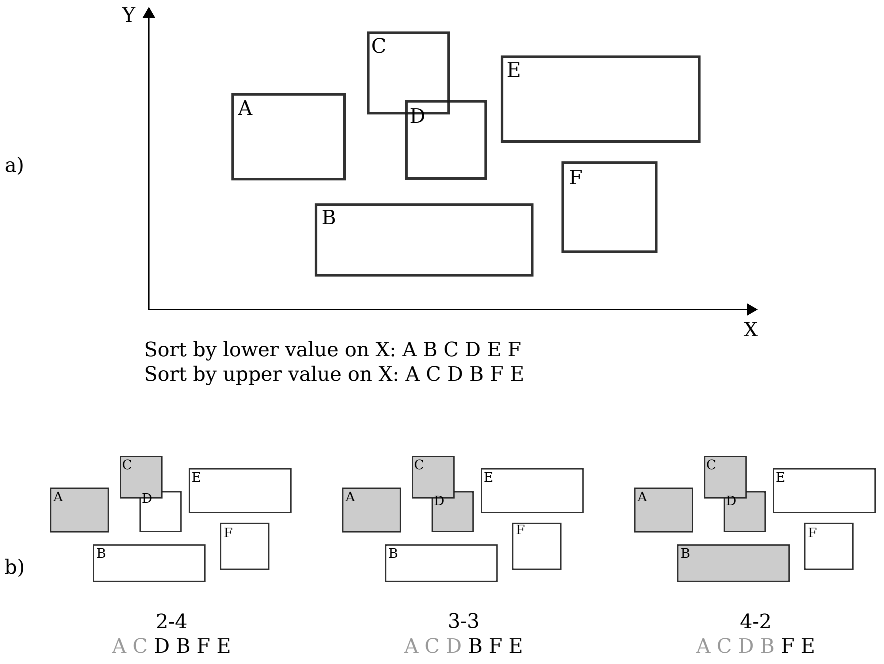

a)
Y
X
A
B
C
D
E
F
Sort by lower value on X: A B C D E F
Sort by upper value on X: A C D B F E
b)
2-4
A C D B F E
3-3
A C D B F E
4-2
A C D B F E

Figure 3.3: a) example of an overflown R*-tree node and b) its entries distributions during splitting for upper values of axis X.

Input: Node N
Output: Axis $axis$

```
1 foreach $axis$ do
2     $A$ = sort entries by lower value on axis;
3     $B$ = sort entries by upper value on axis;
4     Determine all the distributions of $A, B$ (as described in text);
5     foreach $distribution$ do
6         find the sum $s$ of margins for both groups of the distribution;
7     find the sum $S$ of all the $s$ for each distribution;
8 $axis \leftarrow$ the axis with the minimum $S$;
9 return $axis$;
```

Algorithm 3.2.3: `ChooseSplitAxis`(Node N): R*-tree splitting. Called by `ChooseSplit` (Algorithm 3.2.2). Based on description in [7, p. 326].

The method `ChooseSplitIndex` is called by `Split` (Algorithm 3.2.2) and selects the two groups in which the overflown node will be split. After the axis of the split is selected (`ChooseSplit` Algorithm 3.2.3), the selected axis is examined. It creates two sortings: by lower and by upper value of this axis (lines 1–2). Then, it finds all the available distributions of the entries of the node (line 3), as it was explained in the introduction of this section (3.2.2 - page 58). For each distribution the overlap value and the area value are calculated. The distribution with the minimum overlap value is selected. Ties are resolved by choosing the distribution with the minimum area value.

```
Input: Axis $axis$
Output: Group $A$, Group $B$ (the result of the spitting)

1 $A$ = sort entries by lower value on $axis$;
2 $B$ = sort entries by lower value on $axis$;
3 Determine all the distributions of $A, B$ (as described in text);
4 foreach $distribution$ do
5     compute overlap value $O$ of both groups of the distribution;
6     compute area value $A$ of both groups of the distribution;
7 pick the distribution with minimum $O$;
8 resolve ties by choosing the minimum $A$;
9 return $A, B$;
```

Algorithm 3.2.4: `ChooseSplitIndex`(Axis *axis*): R*-tree splitting. Called by `ChooseSplit` (Algorithm 3.2.2). Based on description in [7, p. 326].

3.2.3 Reinsert

Since R-tree is a dynamic index data structure, different sequences of the same insertions will lead to a different indexing. Moreover, the way old entries were inserted in the tree might not reflect the current status of the indexed data, leading to a bad retrieval performance. In their paper, Beckmann, Kriegel, Schneider and Seeger [7] examined the performance effect that the reinsertion of old entries in the tree would have. The results showed a performance improvement of 20% to 50% depending on the type of queries [7, p. 326]. This is the reason why R*-tree dynamically reorganizes itself during the insertion of new entries.

The insertion of new entries is similar to the one described for the original R-tree (Algorithm 2.1.2), except the overflow treatment that will be presented in the rest of the section. Method `InsertData` (Algorithm 3.2.5) is a simple wrapper around the main insertion method `Insert` (Algorithm 3.2.6). It initiates the procedure of inserting a new entry in the tree, and calls `Insert` (Algorithm 3.2.6) with the level of the leaf nodes as argument.

```
Input: Entry $E$
1 $l \leftarrow$ leaf level of the tree;
2 Insert ($E$, $l$);
```

Algorithm 3.2.5: `InsertData`(Node N): R*-tree Insertion. Based on description in [7, p. 327].

Method `Insert`, presented in Algorithm 3.2.6, is responsible for performing the insertion of new entries in the appropriate level of the tree. The first time it is called the level argument is the level of the leaf nodes. It calls `ChooseSubtree` (Algorithm 3.2.1) to find the node N that will accommodate the new entry. If N has enough room the new entry is added to the node. Otherwise `OverflowTreatment` (Algorithm 3.2.7) is called in order to perform either a re-insertion or a split of the node. Next, if `OverflowTreatment` splitted a node, `OverflowTreatment` is propagated upwards, and if a splitting of the root occurs, a new root is created. Finally all the MBRs are adjusted to reflect the changes of the tree.

```
Input: Entry $N$, Level $l$
1  $N \leftarrow$ ChooseSubtree ($E$, $l$);
2  if $N$ is not full then                          /* Add entry to node */
3      add $E$ in $N$;
4  else
5      add $E$ in $N$ ;        /* split and others expect M+1 entries */
6      OverflowTreatment ($N, l$);

7  if OverflowTreatment was called and split was performed then
8      propagate OverflowTreatment upwards if necessary;

9  if root was split then                          /* Grow tree taller */
10     create new root, and add the old root's split nodes as children;
11 adjust all MBRs in the insertion path;
```

Algorithm 3.2.6: `Insert`(Node N, Level l): R*-tree Insertion. Based on description in [7, p. 327].

Method `OverflowTreatment`, described in Algorithm 3.2.7, decides how an overflown node will be handled. If `OverflowTreatment` is called for the first time in this level, some of the entries of the node will be re-inserted by `ReInsert` (Algorithm 3.2.8). Otherwise, the node is split by `Split` (Algorithm 3.2.2).

```
   Input: Node $N$, Level $l$

1  if $l$ is not root level and this is the first call of OverflowTreatment then
2      ReInsert ($N$);
3  else
4      Split ($N$);
```

Algorithm 3.2.7: `OverflowTreatment`(Node N, Level l): R*-tree Insertion. Called by `Insert` (Algorithm 3.2.6). Based on description in [7, p. 327].

Method `ReInsert`, shown in Algorithm 3.2.8, is responsible for re-organizing the tree by re-inserting some of the overflown node's entries. It calculates the distance of the center of each entry from the center of the node. The p entries of the node that have the largest distance are removed from the node and re-inserted (to the leaf nodes) by calling `Insert` for each of them.

```
   Input: Node $N$

1  foreach entry $e$ of $N$ do
2      compute distance $d$ between center of $e.mbr$ to the center of $N.mbr$;
3  sort $ds$ in descending order;
4  remove the first $p$ entries from $N$;
5  adjust $N.mbr$;

6  foreach of the removed $p$ entries $e$ of $N$, keeping the sorting order do
7      Insert ($e$) ;                    /* call Insert to reinsert them */
```

Algorithm 3.2.8: `ReInsert`(Node N): R*-tree Insertion. Called by `OverflowTreatment` (Algorithm 3.2.7). Based on description in [7, p. 327].

3.3 Hilbert R-tree

In [36], Kamel and Faloutsos propose the Hilbert R-tree, a hybrid structure between the R-tree and B$^+$-tree. The way the splitting of overflown nodes is handled, involves the usage of the Hilbert filling curve, which serves as the ordering criterion of a node's entries. In Section 3.3.1 the Hilbert curve and its properties are presented, in Section 3.3.2 the basic properties of the data structure are given. In Section 3.3.3, insertion is presented; in Section 3.3.4, the way the splitting of overflown nodes is handled is explained; and finally, in Section 3.3.5, the deletion is described.

3.3.1 The Hilbert curve

Space filling curves are paths than can be applied in a 2-dimensional grid. Such a path visits all the points of the grid, exactly once without crossing itself and joins each point of the grid with a vertex. A path has two free ends, a start and an end that can be joined with other paths. These curves are usually constructed recursively, by defining a basic curve of order 1. Then, to derive the curve of order i, each vertex is replaced by the curve of order $i-1$ which could be rotated and reflected to fit the new curve [18]. The construction of the curves can of course be generalized for higher dimensions.

The Hilbert curve, proposed by David Hilbert [31] in 1891, is a space filling curve that can be constructed by Algorithm 3.3.1. The algorithm is recursive and its initial arguments is the order of the curve and a default value of 90 degrees. The algorithm is presented in Logo style where a "pen" moves on a canvas for a defined length, and while it moves it draws a straight line on the canvas. When it stops moving we can change the direction of the next straight line. The drawing point of the order 1 curve turns right, moves forward, turns left, moves forward, turns left, moves forward, and turns right. Higher order curves recursively call the drawing of the lower order curves. Figure 3.4 shows 4 Hilbert curves (black path) of order one, two, three and four. All curves have the same "move forward" length and for each curve the grid (light gray) they fill is shown.

In [18], the spatial distance-preserving mappings ability of various filling curves is investigated. More specifically, the performance of a distance preserving mapping under range and nearest neighbor queries is benchmarked and the results show that Hilbert curve behaves better because it avoids long jumps between points. This is the reason why the Hilbert curve is used as the ordering criterion in Hilbert R-tree.

```
Input: Level level, Angle angle
Output: Hilbert Curve Drawing
if $level == 0$ then
    return
/* Always move forward by a predefined length */
turn right ($angle$);
Hilbert ($level - 1$, $-angle$);
move forward;
turn left ($angle$);
Hilbert ($level - 1$, $angle$);
move forward;
Hilbert ($level - 1$, $angle$);
turn left ($angle$);
move forward;
Hilbert ($level - 1$, -$angle$);
turn right ($angle$);
```

Algorithm 3.3.1: 2-dimensional Hilbert curve construction (Logo style). Initial *angle* argument is 90 degrees.

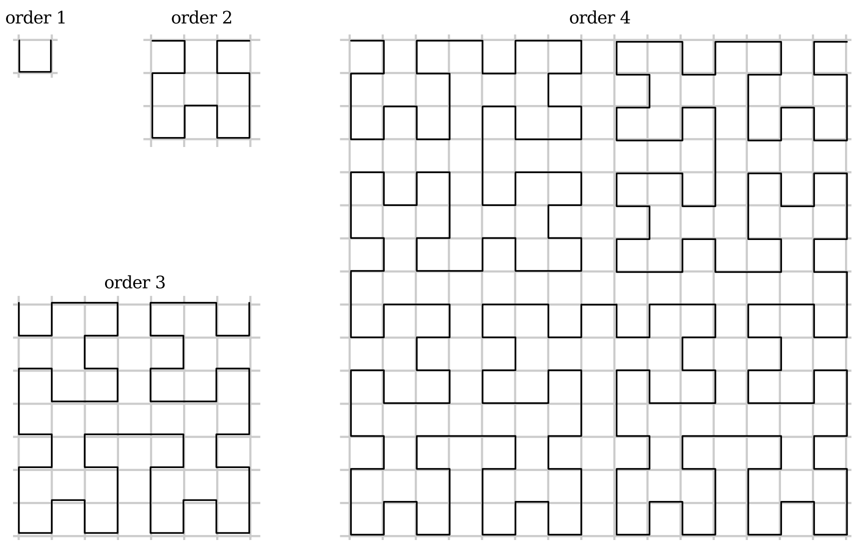


Figure 3.4: 2-dimensional Hilbert curves of order 1, 2, 3 and 4.

3.3.2 Basic Properties

Hilbert R-trees differ only slightly from the original R-tree. The leaf nodes have the same structure but the internal nodes are of the form (mbr, id, lhv), where mbr is the MBR that contains the object and id the object's identifier, like in the R-tree. Additionally lhv stores the largest Hilbert value of all the entries below the node. The largest Hilbert value lhv is used as the primary key on which the entries of the tree are sorted, and this is where lies the resemblance with B^+-trees.

3.3.3 Insertion

Insertion of Hilbert R-tree is similar to the one of R-trees (Section 2.1.3). Method `Insert` handles the insertion and is described in Algorithm 3.3.2. The way a leaf node is found for the new entry (line 1) is handled by `ChooseLeaf`. The overflown nodes are balanced with `HandleOverflow` described in Algorithm 3.3.4 (line 6) using other sibling nodes, as presented in Section 3.3.4, and a split is performed if needed. Node changes are propagated upwards (line 8) and are handled by `AdjustTree`.

```
Input: Entry $E$, Node $T$ (root)
Output: Modifies R-tree by adding new entry.
1  $L \leftarrow$ ChooseLeaf ($T$, $E$);   /* Find leaf node for the new entry */

2  if $L$ is not full then                         /* Add entry to leaf node */
3      add $E$ in $L$, ordered by Hilbert value;
4      return
5  else
6      $L_1 \leftarrow$ HandleOverflow ($L$);

7  $S \leftarrow$ set containing $L$, the cooperating siblings, and $L_1$;
8  AdjustTree ($S$);                         /* Propagate changes upwards */

9  if $T$ was split then                               /* Grow tree taller */
10     create new root, and add the old root's split nodes as children;
```

Algorithm 3.3.2: `Insert`(Entry E, Node T): Hilbert R-tree Insertion. Based on description in [36, pp. 502–504].

Method `ChooseLeaf` is similar to the one of R-trees (Algorithm 2.1.3). The difference that the largest Hilbert value of the node that is examined is used to select the next node of the insertion path. Method `AdjustTree` is also similar to R-tree's Algorithm 2.1.4, where both the MBRs and the largest Hilbert values of the sibling and upper nodes is adjusted.

3.3.4 Overflown Nodes

Method `HandleOverflow`, presented in Algorithm 3.3.3, handles overflown nodes of Hilbert R-trees. Suppose that when the overflow occurs the level has s nodes. The method, first tries to move some the overlfown's node entries to the other s sibling nodes (line 3). If that fails, the entries of the s nodes are distributed among $s+1$ nodes (line 6).

Since the largest Hilbert value of the entries, represented an ordering, it is possible to perform both the moving (line 3) and the distribution (line 6) of entries.

```
Input: Entry $E$, Node $N$
Output: Node $N$ ($\emptyset$ if no split occurred)
$S \leftarrow$ set containing all entries from $N$ and it's cooperating sibling nodes;
add $E$ to $S$;

if one of the sibling nodes is not full then
    distribute $S$ evenly among the $s$ nodes according to Hilbert value;
    return $\emptyset$;
else                                   /* All sibling nodes are full */
    create a new node $N'$ ;
    distribute $S$ evenly among the $s + 1$ nodes according to Hilbert value;
    return $N'$;
```

Algorithm 3.3.3: `HandleOverflow`(Entry E, Node T): Hilbert R-tree Overflown node handling. Based on description in [36, p. 504].

3.3.5 Deletion

Deletion is slightly different from the other R-tree variants we have encountered so far. It doesn't follow a re-insert procedure, but tries to compact the entries in the available nodes.

Suppose that when the overflow occurs the level has $s+1$ nodes. Method `Delete`, presented in Algorithm 3.3.4 first locates the leaf node, where resides the node to be deleted, and deletes it. If the node is underfull then entries from the other s nodes (line 4) are borrowed, but if all the other s nodes are in the verge of being underfull, the $s+1$ nodes are merged into s nodes (line 6).

The largest Hilbert value of the entries, represents an ordering, that makes possible to both the borrowing and the merging of entries.

3.4 Linear Node Splitting

As we described in Section 2.1.4, the original R-tree has three splitting techniques to handle overflown nodes. In [5] Ang and Tan proposed an additional splitting algorithm of linear time. The goal of the method is first to distribute the entries, of the overflown node in two nodes, as evenly as possible and second

```
   Input: Entry $E$
   Output: Modifies R-tree by deleting entry.
1  $L \leftarrow$ Search ($E$);          /* Find leaf node containing entry E */
2  Remove $E$ from $L$;
3  if $L$ is underfull then
4      borrow entries from the other $s$ nodes;
5      if all $s$ nodes are ready to underfull then
6          merge $s+1$ nodes to $s$;
7          adjust the resulting nodes;
8  $S \leftarrow L$;
9  if underflow occurred then
10     $S \leftarrow S \cup$ cooperating siblings;
11 AdjustTree ($S$);                   /* Propagate changes upwards */
```

Algorithm 3.3.4: `Delete`(Entry E): Hilbert R-tree Deletion. Based on description in [36, p. 504].

to minimize the overlap between them. Finally, the last goal is to minimize total coverage.

The method of the new linear splitting is described in Algorithm 3.4.1. Four lists L_L, L_B, L_R, L_T (line 1) hold the entries e of node N, that are closer to the left, bottom, right and top of $N.mbr$ (lines 2-10). These lists represent two partitionings since each entry can be part of L_L or L_R and L_B or L_T. The decision of the axis, on which the split is performed, depends on the vertical and horizontal distribution of the entries. The metric used is the number of elements in the L_L, L_R (horizontal) and L_B, L_T (vertical) lists.

In Figure 3.5 an example node with 11 entries, of a 2-dimensional R-tree, is given. The rectangles with the black line are the MBRs of the entries, and the dotted rectangle is the MBR of the node. The numbers in parenthesis is the number of elements in each list. The spatial distribution of the nodes is selected on purpose, so that it is easy to find, without calculations, the list in which each node entry belongs to. Qualitative, we see that the maximum number of elements of the horizontal lists is 6, whereas the the maximum number of elements of the vertical lists is 7. This means that the entries are distributed more evenly horizontally, and that the splitting axis will be X.

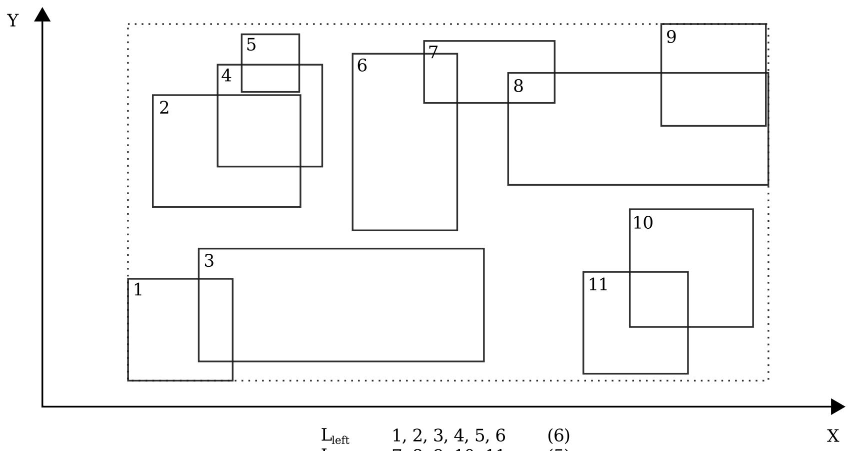
```
Input: Node $N$
Output: Node $N_1$, Node $N_2$

/* initialize lists for left, bottom, right, top                */
$L_L \leftarrow L_B \leftarrow L_R \leftarrow L_T \leftarrow \emptyset$ ;

/* fill lists                                                   */
foreach entry $e \in N$ do
    /* N.mbr = ($L$, $B$, $R$, $T$) - left, bottom, right, top         */
    /* e.mbr = ($x_l$, $y_l$, $x_h$, $x_h$) - left, bottom, right, top */
    if $x_l - L < R - x_h$ then
        $L_L \leftarrow L_L \cup e$;
    else
        $L_R \leftarrow L_R \cup e$;
    if $y_l - B < T - y_h$ then
        $L_B \leftarrow L_B \cup e$;
    else
        $L_T \leftarrow L_T \cup e$;

/* choose split axis                                            */
if max $(|L_L|, |L_R|) <$ max $(|L_B|, |L_T|)$ then
    spit along X axis;
else if max $(|L_L|, |L_R|) >$ max $(|L_B|, |L_T|)$ then
    spit along Y axis;
else                                                    /* tie */
    if overlap $(L_L, L_R) <$ overlap $(L_B, L_T)$ then
        spit along X axis;
    else if overlap $(L_L, L_R) >$ overlap $(L_B, L_T)$ then
        spit along Y axis;
    else                                                /* tie */
        split along axis with smallest total overage;
```

Algorithm 3.4.1: `NewLinear`(Node N): Additional R-tree node splitting method. Based on [5, p. 5]



L_{left}	1, 2, 3, 4, 5, 6	(6)
L_{right}	7, 8, 9, 10, 11	(5)
L_{top}	2, 4, 5, 6, 7, 8, 9	(7)
L_{bottom}	1, 3, 11, 10	(4)

Figure 3.5: Example distribution of a node's entries in the left, right, bottom and top lists.

3.5 optimal split

In [23] the authors presented an optimal node splitting algorithm that is described in [42, pp. 24-25]. When we tried to read the actual paper we couldn't find it, even if it's a relatively new paper from VLDB '98. In one of author's site we read that "there is an error in this paper, a corrected version will appear" [40]. However, we couldn't find a new version of the paper either, so we omit this splitting algorithm.

3.6 VoR-Tree

Sharifzadeh and Shahabi present in [105] the VoR-Tree, an R-tree variant that performs very well for nearest neighbor queries by using Voronoi diagrams.

In Section 3.6.1 we introduce the Voronoi diagram and in Section 3.6.2 the Delaunay graph. In Section 3.6.3 the VoR-Tree data structure is presented and finally in Section 3.6.4 a quick reference to the maintenance of the index is given.

3.6.1 Voronoi diagrams

Let a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^d$. The *Voronoi diagram* of P partitions the $\mathbb{R}^d$ space in n regions. Given a distance metric D, each region includes all the points in $\mathbb{R}^d$ that fulfill the following:

$$\forall p' \in P, p \neq p', D(q,p) \leq D(q,p')$$

We call *Voronoi cell* $V(p)$, the region containing the point p, and all the points that are closet to p than all the other points of P. Finally we call *Voronoi neighbors* of p the points of P with which p has a common Voronoi edge. In Figure 3.6a, we present a set P of eleven points and the corresponding Voronoi diagram for $\mathbb{R}^2$ and Euclidean D. For point p we show, in gray, its the Voronoi cell $V(p)$. Additionally we note one of its Voronoi edges, one of its Voronoi vertexes and one of its neighbors.

For $\mathbb{R}^2$ and Euclidean distance as the distance metric D, Voronoi cells are convex hulls. Each edge of the polygon is a line segment of the perpendicular bisector of the line connecting p to another point of P. We call each of these edges *Voronoi edge.* we call each of its end points, which are also the vertices of the polygon, *Voronoi vertex.*

3.6.2 Delaunay graph

Let an undirected graph $DG(P) = G(V,E)$ with the set of vertices $V = P$. The edges that connect the points:

$$\forall p, p' \in V \text{ and } p \text{ is neighbor of } p'$$

form the *Delaunay graph.* In Figure 3.6b we represent the Delaunay graph (black line) of the set of points P, that is shown in Figure 3.6a. The dotted diagram is the Voronoi diagram of the same set.

3.6.3 VoR-Tree Structure

The structure of the VoR-Tree augments the original R-tree with Voronoi diagram and Delaunay graph information. More specifically, the internal nodes are structured like the ones of R-tree, but the leaf nodes store Voronoi information.

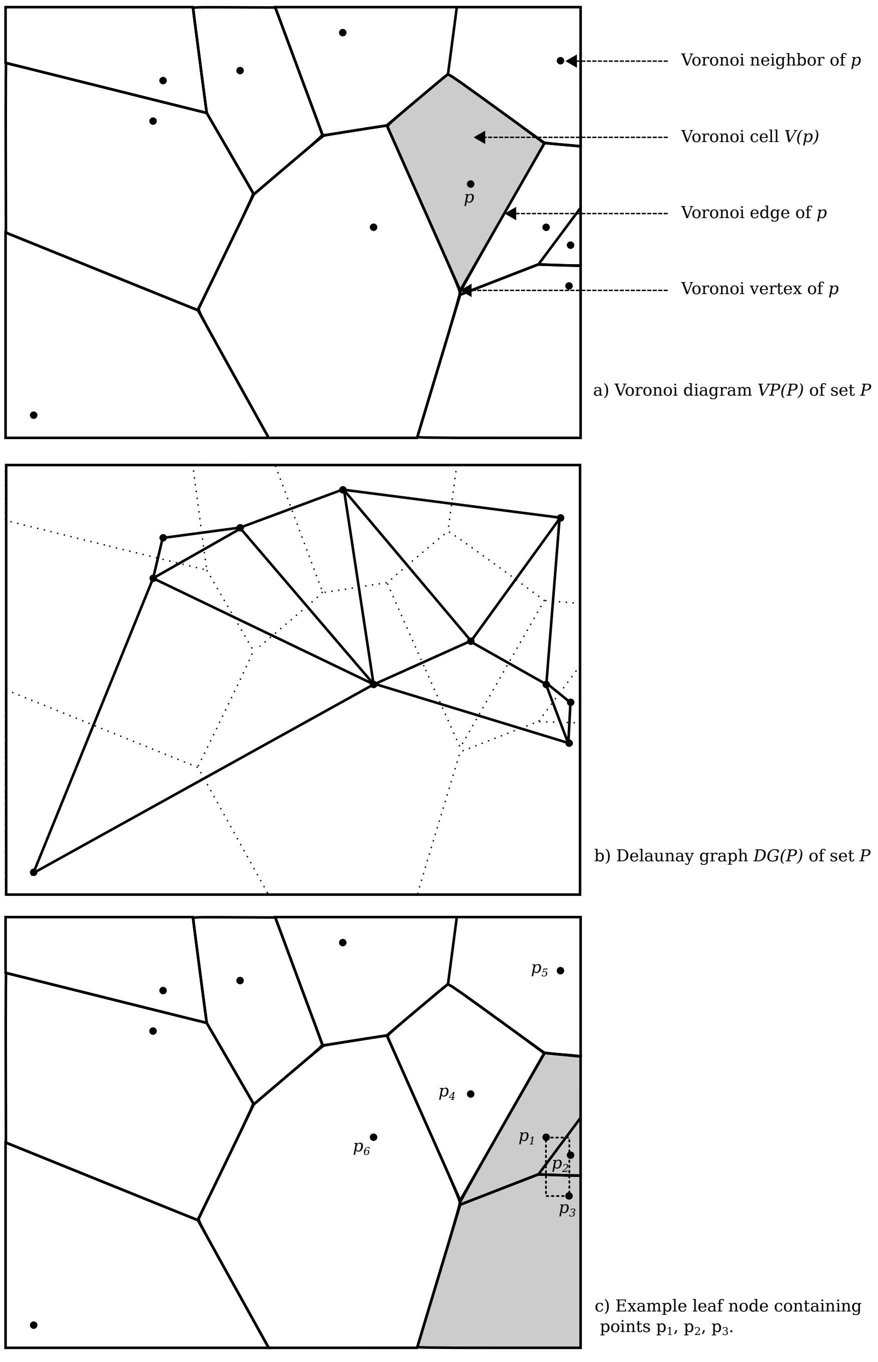

Voronoi neighbor of *p*
Voronoi cell *V(p)*
p
Voronoi edge of *p*
Voronoi vertex of *p*
a) Voronoi diagram *VP(P)* of set *P*
b) Delaunay graph *DG(P)* of set *P*
p_5
p_4
p_6
p_1
p_2
p_3
c) Example leaf node containing points p_1, p_2, p_3.

Figure 3.6: a) Voronoi Diagram, b) Delaunay graph for $\mathbb{R}^2$ and Euclidean D for a set of 11 points and c) example leaf node containing 3 points.

Let a set of points P and the corresponding Voronoi diagram $V(P)$. The leaf node stores everything stored in an R-tree, a set of points P_S, subset of P. Additionally, for each point $p \in P_S$, it stores the pointer to the location of each Voronoi neighbor of p ($VN(P)$) and also the vertices of the Voronoi cell of p ($V(p)$).

Let a leaf node of the points shown in Figure 3.6c, that contains the points p_1, p_2, p_3. The corresponding Voronoi cells have a gray fill, and the MBR of the node is shown in a dashed rectangle. For each point the node contains the following information:

$VN(p_1) = \{p_2, p_3, p_6, p_4, p_5\}$

$V(p_1) = \{$ vertices of p_1's Voronoi cell $\}$

$VN(p_2) = \{p_3, p_1\}$

$V(p_2) = \{$ vertices of p_2's Voronoi cell $\}$

$VN(p_3) = \{p_6, p_4, p_1, p_2\}]$

$V(p_3) = \{$ vertices of p_3's Voronoi cell $\}$

3.6.4 Insertion, Deletion and Querying

The maintenance of the VoR-Tree is described in detail in the authors' paper. Moreover, the algorithms are given in a clear code-like form, so we don't feel the need to further explain them here.

3.7 Conclusion

In this Section, six variants of the R-tree we presented in detail. We encountered a variety both in the algorithmic approach, and in the domain each variant tries to solve. Also, it's intersting that even recently, almost thirty years after the introduction of the original R-tree, there is active research going on in the field of low level spatial indexing solutions. Finally, their common characteristics would benefit from an common spatial data structure that could be used for the implementation of all these R-tree variants.

Chapter 4

MySQL Internals

This chapter focuses on MySQL internals and the way the server performs operations behind the scenes. We begin with section 4.1 where we define which code we work with. Then, in section 4.2, a bird's eye description of MySQL's architecture is given. In Section 4.3 the storage engine pluggable architecture is presented and in Section 4.4 we intordouce the MyISAM storage engine. The core of this chapter is found in section 4.5, where we dive in the details of the way spatial indexing is performed with MySQL and the MyISAM storage engine. Finally, we conclude in section 4.6.

We should note that throughout the whole chapter any mentioned directory and files, that belong to the MySQL codebase, are paths relevant to the directory of the source code. For example the directory `storage/myisam` and the file `storage/myisam/ha_myisam.cc` are both relative paths to the directory of the codebase.

4.1 Codebase details

The software where we performed the implementation of this research is MariaDB (see Section 1.3). As we already mentioned in Section 1.3.2, MariaDB is a fork of MySQL and its code and features are synchronized with the changes of the MySQL code. This means it's a backward compatible, drop-in replacement of MySQL. So, when we refer to "MySQL" we refer to the MariaDB codebase or the MariaDB server, because the code we discuss in the next chapters is common and everything that we discuss applies both to the RDBMSes MySQL and MariaDB.

MariaDB is an open source project so it can be downloaded and used under the terms of the GPL v2 license. Installation instructions are given in the documentation of the software which can be found in [47]. The development source code is available through the publicly available repository [46] and detailed instructions can be found in [45].

The version we worked on is 5.5 (more specifically 5.5.27). The same changes can, with extremely few modifications, be applied in the MariaDB 5.3, as well as the MySQL code.

4.2 MySQL Architecture

The MySQL online manual [59] includes a wealth of information about MySQL in different levels of detail. MySQL offers different storage engines [60] each trying to solve different needs. Storage engines are plugins to the server and implement the actual physical storage of the tables and data. Some of the available storage engines are briefly described to demonstrate the range of different needs that MySQL can handle:

- **InnoDB**: a transactional ACID-compliant [41, pp. 19–21] storage engine, that provides crash-safe data storage [61].
- **MyISAM**: non-transactional, simple but not crash safe. Can index spatial data [62], [102, pp. 17–19].
- **Archive**: stores large amounts of data without indexes, compressed so that they have a very small footprint [64].
- **Memory**: stores contents only in memory. It's very fast, but not crash safe [63].
- **InfiniDB**: column-oriented storage engine for data warehouse solutions. The product is distributed separately [10].
- **SphinxSE**: provides an SQL interface to the Sphinx fulltext search server [120, 72].

Figure 4.1 is borrowed from [102] and gives an abstraction of the MySQL server internals. MySQL follows the client/server architecture and the server is implemented in such a way so that the query handling and the actual reading or writing of data is separated:

- The core server handles the queries and requests data from the storage engines.
- The storage engines, that are plugins to the server, perform the actual reading and writing of data and reply to the requests of the core server.

The flow of a query throughout the server can be observed in Figure 4.1.

1. Clients connect to the server (component 1) and send queries. In this level the server handles network, threads, authentication and security.
2. Then the client's query is transfered to the parser (component 2) that parses the SQL. In this level, all the functionality that spans across all storage engines is handled. These include triggers, stored procedures and built in functions (date, time, string, math, encryption, etc). For queries that only read data, the parser checks whether the query's resultset should be fetched from a MySQL internal cache (component 3), or if the resultset should be read from the database. If the server decides that the read or write query needs to be executed, then the SQL optimizer (component 4) finds an optimal execution plan and initiates the execution of the query.
3. In order to execute the query the server requests from the table's storage engine to read or write data. The storage engine replies back to the server and then the server performs final operations on the returned data.

4.3 Storage engine implementation overview

The pluggable architecture of MySQL is discussed in length in [66, 25]. The storage engine plugins are implemented through two main structures [25, pp 161-162] that are found in `sql/handler.h` and `sql/handler.cc`:

- **handler** is a class. It is the interface for dynamically loadable storage engines and there can be many objects of this class. It provides the methods that work on a single table which includes, among others, operations like opening a table, reading from an index and writing a row.

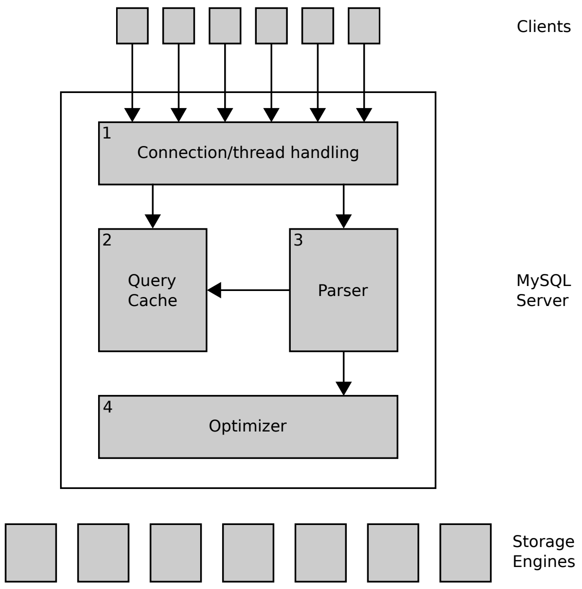


Figure 4.1: A logical view of the MySQL server architecture. Source [102]

- **handlerton** is a singleton structure. There is one instance per storage engine and provides access to the storage engine's functionality that affect the whole of the storage engine. This includes, among others, operations like committing and aborting transactions, and showing the status of the storage engine.

4.4 MyISAM storage engine

MyISAM is one of the main storage engines of MySQL and its properties are discussed in length in [60, 102], where the interested reader can find extensive examples and design ideas. MySQL was originally built around MyISAM-like storage and multiple pluggable storage engines were added later. This legacy is still reflected, even if the core server vs. storage engine separation is clear, by the fact that some functionality is still tied to the core server and engineered having in mind the way MyISAM is designed.

MyISAM provides a large list of features including compression, full-text search indexing, spatial functions and spatial indexing. Some of the features it's missing are transactions, row-level locking and crash safety. However, MyISAM suits very well certain workloads and specifications and is used in production system with success.

For MyISAM, the code that is responsible for implementing the interface with the pluggable storage engines (see Section 4.3) is found in `storage/myisam/ha_myisam.cc` and `storage/myisam/ha_myisam.h`. The class `ha_myisam` inherits from `handler` and several functions implement methods of `handlerton`.

4.5 R-trees in MyISAM

In this section the way R-trees are handled in MySQL is discussed. Insertion is presented in Section 4.5.1, deletion in Section 4.5.2 and search in Section 4.5.3. Finally, a summary is given in Section 4.5.4.

MySQL implements the R*-tree variant. MySQL's R-tree index has the structure of the original R-tree (which is the identical to the R*-tree). The tree has levels, and in each level there are several nodes. Each node is either an internal node or a leaf. Each node has many keys, and each key is a data structure with two members:

- a pointer to a node down (for internal nodes), or to data (for leaf nodes).
- a rectangle that represents the MBR of the data the pointer points to. For internal nodes it is the MBR of the child node, and for leaf nodes it is the MBR of the data.

In the sections below, the term "node" is equivalent to the term "disk page". Each node of the tree has the size of one disk page. Modifying a tree node means that a disk page is modified, and writing a node to disk means that a disk page is written.

4.5.1 Insertion

In this section, we describe the insertion flow for MySQL's R-tree keys. In Section 4.5.1.1, the algorithm is presented in an abstract way and then, in Section 4.5.1.2, it is described with more details. Finally, in Section 4.5.1.3, the differences with the original R-tree algorithm are discussed.

4.5.1.1 Abstract description

The code that is associated with the R-tree insertion resides in the source files `storage/myisam/rt_index.c` and `storage/myisam/rt_key.c`. A high level view of the insertion flow is presented in Algorithms 4.5.1 and 4.5.2 and the most important methods are:

- `rtree_insert_level`
- `rtree_insert_req`
- `rtree_add_key`

The method `rtree_insert_level` (Algorithm 4.5.1) is called from the root of the tree and calls `rtree_insert_req`. When `rtree_insert_req` returns, the new key has been added in the leaf level and all the nodes below the root have been adjusted. Then the root node is adjusted and the insertion finishes.

The method `rtree_insert_req` (Algorithm 4.5.2) is called recursively and descends the tree towards the leaf nodes. If an internal node is encountered then `rtree_insert_req` is called (line 3) to descend down one level with arguments the child node and the increased level. When it returns, the current level is adjusted and if it's needed it is split. When the leaf node is encountered the key is added (line 8) and if necessary the node is split.

```
Input: key
1 begin
2     rtree_insert_req (key, 0);
3     Adjust root if needed;
4 end
```

Algorithm 4.5.1: `rtree_insert_level` abstract: MyISAM R-tree insertion abstract.

4.5.1.2 Detailed description

This section describes MySQL's R-tree insertion flow in detail. More specifically the following methods are presented:

- `rtree_insert` (Algorithm 4.5.3)

```
Input: key, level
1  begin
2      if can go one level down then
3          rtree_insert_req (key, level + 1);
4          Adjust key if child node was modified;
5          Split node if necessary;
6          return
7      else
8          rtree_add_key;
9          return
10 end
```

Algorithm 4.5.2: `rtree_insert_req` abstract: MyISAM R-tree insertion abstract.

- `rtree_insert_level` (Algorithm 4.5.4)
- `rtree_insert_req` (Algorithm 4.5.5)
- `rtree_add_key` (Algorithm 4.5.6)

Even if we do provide enough details to understand how insertions are performed, some details fall outside the scope of the description. The description focuses on the fact that somehow, the key information can be read, updated and saved, and that nodes can be read and saved permanently, but doesn't mention how this is performed. These are important but lower level MyISAM operations and the interested reader can check directly in the source code files.

rtree_insert The method is described in Algorithm 4.5.3 and is the single point of entry for the insertion of keys in MySQL's R-trees. It modifies the index by inserting one key and returns 0 for success and 1 if something went wrong. It is a wrapper around `rtree_insert_level` (line 1) that is described in Algorithm 4.5.4. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. key: the new leaf key that will be inserted in the tree
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

```
Input: $info$, $keynr$, $key$, $key\_length$
Output: Modifies R-tree: 1 for Error, 0 for OK
1 $res \leftarrow$ rtree_insert_level ($info$, $keynr$, $key$, $key\_length$, $-1$);
2 return $res$;
```

Algorithm 4.5.3: `rtree_insert`: MyISAM R-tree insertion.

rtree_insert_level The method is described in Algorithm 4.5.4. It modifies the index by calling `rtree_insert_req` to insert the key. Returns 0 if the root was not split, 1 if it was split and -1 if something went wrong. It is called either during insertion by `rtree_insert` (Algorithm 4.5.3) or during deletion at the re-insertion stage (Section 4.5.2.2, Algorithm 4.5.9). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. key: the new leaf key that will be inserted in the tree
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. ins_level: the level at which the key is going to be insert. To insert a leaf node (like from an SQL Insert command) -1 is used. To insert a key during delete reinsertion (Section 4.5.2.2, Algorithm 4.5.9) the level of the key is used.

First, the root of the tree and information regarding the table's keys are taken from info. Afterwards, an empty new node is created in memory, in case it's needed further down the algorithm. Then a check for the existence of the root node is performed (line 4). If it doesn't exist it's created and the key is added to the empty root. If the root does exist, then `rtree_insert_req` is called (line 11). It recursively calls itself, in order to insert the key to the leaf node and adjust all the associated internal nodes. It returns with either an error or success. If the root was split during the process, a new root is created and keys are added there.

```
Input: $info$, $keynr$, $key$, $key\_length$, $ins\_level$
Output: Modifies R-tree: $-1$ for Error, 0 if root was *not* split, 1 if root
        was split

$keyinfo \leftarrow$ take key information from $info$;
$new\_page \leftarrow$ new empty node;
$old\_root \leftarrow$ take root node from $info$;

if Root doesn't exist then
    Create new root;
    if error during new root creation then
        return $-1$;
    else
        $res \leftarrow$ rtree_add_key;        /* add key to the empty node */
        return $res$;

$res \leftarrow$ rtree_insert_req ($info$, $keyinfo$, $key$, $key\_length$, $old\_root$,
$new\_page$ , $ins\_level$, 0);

if $res == 0$ then                             /* Root was not split */
    return 0
else if $res == 1$ then                           /* Root was split */
    Create new root and add keys there;
    if error during new root creation then
        return $-1$
    return 1
else
    return $-1$
```

Algorithm 4.5.4: `rtree_insert_level`: MyISAM R-tree insertion. Called from the root of the tree.

rtree_insert_req The method, described in in Algorithm 4.5.5, is called recursively in order to modify one level of the tree. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.

2. keyinfo: data structure that includes information about the key associated with the insertion.

3. key: is the new leaf key that will be inserted in the tree

4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

5. new_page: an new empty node in memory. It is a place holder to insert new keys if needed.

6. ins_level: the level at which the key is going to be insert. To insert a leaf node (like from an SQL Insert command) -1 is used. To insert a key during delete reinsertion (Section 4.5.2.2, Algorithm 4.5.9) the level of the key is used.

7. level: the current level of the tree. When `rtree_insert_req` descends one level down then this argument is increased by one.

First, the algorithm decides if the recursion should go one level down towards the leaf nodes (line 1). In case `rtree_insert_req` was called by `rtree_insert_level`, in order to insert a new key in the tree, then the recursion continues until the leaf nodes are reached. In case the `rtree_insert_req` was called by `rtree_delete` during the deletion of a key, in order to re-insert a node that became filled less than the fill factor, the recursion continues until the level of the re-inserted node is reached.

If the algorithm must go one level down (line 1), then one key is picked up from the available keys of the node (line 2). The child of this key is the node where the algorithm will descend into (line 4). Then `rtree_insert_req` is called for this key. Once it returns, the key has been added somewhere below and all the nodes below the current level have been adjusted. If the child node was not split (line 5), then the current node is adjusted. If the child was split, (line 11), then a new key points to the new child node. Afterwards, the new key and the old key are adjusted, the new key is added to the node (line 14) and the method returns the result of `rtree_add_key` or -1 if something went wrong.

If the algorithm must not go one level down (line 21), then the key is added to the node (line 22) and the method returns the result of `rtree_add_key` or -1 if something went wrong.

rtree_add_key The method handles adding the key to a node and it is presented in Algorithm 4.5.6. The input arguments of this method are the following:

```
Input: $info$, $keyinfo$, $key$, $key\_length$, $page$, $new\_page$, $ins\_level$, $level$
Output: Modifies one level in the R-tree: $-1$ for Error, 0 if child was not
        split, 1 if child was split
1  if go down one level then
2      $k \leftarrow$ rtree_pick_key                 /* will insert into entry $k$ */
3      $p \leftarrow$ node where $k$ points to (internal node or data);
4      $res \leftarrow$ rtree_insert_req ($info$, $keyinfo$, $key$, $key\_length$, $p$,
       $new\_page$ , $ins\_level$, $level + 1$);
5      if $res == 0$ then                          /* Child was not split */
6          rtree_combine_rect ($k$, $key$);      /* add $key$ MBR to $k$ MBR */
7          save node;
8          if error then
9              return $-1$
10         return 0;
11     else if $res == 1$ then                          /* Child was split */
12         $new\_key \leftarrow$ new child node;
           /* calculate & store new and existing key MBRs        */
13         rtree_set_key_mbr ($k$); rtree_set_key_mbr ($new\_key$);
           /* add new key to current node                         */
14         $res \leftarrow$ rtree_add_key ($new\_key$);
15         save current node;
16         if error during the above then
17             return $-1$
18         return $res$
19     else
20         return $-1$
21 else    /* Node is leaf or we don't have to go further down */
22     $res \leftarrow$ rtree_add_key ($key$) ;
23     save node;
24     if error during write then
25         return $-1$ ;
26     else
27         return $res$;
28
```

Algorithm 4.5.5: `rtree_insert_req`: MyISAM R-tree insertion. Called recurcively on each level of the tree.

1. info: data structure that includes information about the database table associated with the insertion.
2. keyinfo: data structure that includes information about the key associated with the insertion.
3. key: is the new leaf key that will be inserted in the tree
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. new_page: a new empty node.

If the node has enough free space for one more key, then the key is added (line 1). If the node is a leaf then the key points to the data stored. If the node is internal then the key points to a child node. The method returns 0 indicating that the node was not split.

If the node does not have enough space for one more key, then the node is split and the new node is written in new_page (line 7). The method returns -1 on error or 1 on success indicating that the node was split.

```
   Input: info, keyinfo, key, key_length, new_page
   Output: Modifies key node: −1 for Error, 0 for no split, 1 for split

 1 if node has enough free space to hold one more key then
       /* modify key's pointer                                  */
 2     if node is not leaf then
 3         add the child node link to the key;
 4     else
 5         add the data record link to the key;
 6     return 0;
 7 res ← rtree_split_page;
 8 if res == 1 then
 9     return −1;
10 else
11     return 1;
```

Algorithm 4.5.6: `rtree_add_key`: MyISAM R*-tree insertion. Add key to node

4.5.1.3 Comparison with original R*-tree insertion

The insertion algorithm closely follows the original R*-tree. The one and major difference with the original algorithm is that the nodes don't keep the information of their parent node. This means that changes cannot be adjusted after the insertion has finished. Each level is adjusted right after the insertion of the node has been finished in its child node. This doesn't affect the logic of the algorithm, it simply makes the code to perform better as far as IO time is concerned.

Another interesting option concerns the criteria used for finding the correct insertion path. In Section 3.2, we presented the criteria tested by Beckmann et *al.* which include among others minimization of area (that the authors chose as the preferable method) and margin of MBRs. The functionality to use either one of these criteria is available in the code and it can be compiled accordingly, with the default being the area.

4.5.2 Deletion

In this section, we describe how deletion is performed in MySQL's R-tree keys. In Section 4.5.2.1, the algorithm is presented in an abstract way and then, in Section 4.5.2.2, it is described with more details. Finally, in Section 4.5.2.3 ,the differences with the original R-tree deletion algorithm are discussed.

4.5.2.1 Abstract description

The code that is associated with the R-tree deletion resides in the source files `storage/myisam/rt_index.c` and `storage/myisam/rt_key.c`. A high level view of the deletion flow is presented in Algorithm 4.5.7.

The method `rtree_delete` is called from the root of the tree and then it calls `rtree_delete_req`. This method recursively calls itself until the proper leaf node is reached and the key is deleted (line 2). During this process some nodes might require reinsertion. This is performed after `rtree_delete_req` has returned (line 3). Reinserting is required when some of the nodes become filled less than their minumum fill factor during the deletion process,.

```
Input: key
begin
    rtree_delete_req (key);
    Reinsert deleted nodes;
end
```

Algorithm 4.5.7: `rtree_delete` abstract: MyISAM R-tree deletion abstract.

4.5.2.2 Detailed description

This section describes MySQL's R-tree deletion flow in detail. More specifically the following methods are presented:

- `rtree_delete` (Algorithm 4.5.8)
- `rtree_delete_req` (Algorithm 4.5.9)
- `rtree_delete_key`

Even if we do provide enough details to understand how deletion is performed, some details fall outside the scope of the description. The description focuses on the fact that somehow the key information can be read, updated and saved, and that nodes can be read and saved permanently, but doesn't mention how this is performed. These are important but lower level MyISAM operations and the interested reader can check directly in the source code files.

rtree_delete The method is described in Algorithm 4.5.8, and it is the single point of entry for deleting a key from the index. It modifies the index by deleting one key and returns `0` for success and `-1` if something went wrong (same as `rtree_insert` in Algorithm 4.5.3). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the deletion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. key: is the leaf key that will be deleted in the tree

4. key_length: is the key length. Keys can have different lengths because they can be of columns of data types with different size.

First, the key's information are taken from the table data structure (line 1) as well as the root of the tree. Also, an empty list, that can accommodate nodes that will be re-inserted, is created (line 3). Then `rtree_delete_req` is called. This method calls itself recursively and descends the tree until the leaf nodes are reached. Then, it deletes the keys. During this process, some nodes might become filled less than the fill factor and must be re-inserted. They are deleted from the tree and they are appended to ReinsertList. Once method `rtree_delete_req` returns, the re-insertion takes place (line 6). The method `rtree_insert_level` (line 9), described in Algorithm 4.5.4, is called to insert either leaf nodes or internal nodes. For internal nodes it reinserts the keys of the internal nodes. The subtrees of the internal node's keys are left untouched.

rtree_delete_req The method, described in Algorithm 4.5.9, is called recursively in order to modify one level of the tree. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the deletion.
2. keyinfo: data structure that includes information about the key associated with the insertion.
3. key: is the leaf key that will be deleted from the tree
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. page: the current page that is operated.
6. page_size: total size of keys on the current page.
7. ReinsertList: the list of nodes that might require to be re-inserted after deletion has finished.
8. level: the current level of the tree. When `rtree_delete_req` descends one level down then this argument is increased by one.

First, for each node that is visited all the keys are checked (line 1) in a loop. If the node is internal (line 2) and if the key to delete MBR in inside the node's

```
Input: $info$, $keynr$, $key$, $key\_length$
Output: Modifies R-tree: $-1$ for Error, 0 if key was deleted

keyinfo ← take key information from info;
old_root ← take root node from info;
ReinsertList ← empty list of pages;

res ← rtree_delete_req (info, keyinfo, key, key_length, old_root,
  page_size, ReinsertList, 0);

if res == 0 then                                    /* not split */
    foreach page i ∈ ReinsertList do
        foreach key k ∈ ReinsertList.[i] do
            l ← ReinsertList.pages.[i].[k].level;
            rtree_insert_level (info, keynr, k, key_length, l);
            if root was split and tree grew one level then
                ∀ remaing pages and keys increase by one the re-insertion
                level;
            if any error during the above then
                return −1;
    return 0;
else if res == 1 then                               /* key not found */
    return −1;
else if res == 2 then                               /* tree is now empty */
    return 0;
else
    return −1;
```

Algorithm 4.5.8: `rtree_delete`: MyISAM R-tree deletion. Called from the root of the tree.

key MBR (line 3), then `rtree_delete_req` is called for the child node (line 5). Otherwise the loop visits the next key of the node.

Once `rtree_delete_req` returns, the algorithm takes different actions depending on the returned value. If the deletion was successful (returned 0 - line 6), the fill of the page is checked (line 7) and if it is below the fill factor the node is appended to the ReinsertList and `rtree_delete_key` is called to delete the key (line 11). If the key for deletion was not in the subtree just checked (returned 1 - line 15) visit the next key of the node (line 1). If the child node was is empty and the subtree is no longer needed (returned 2 - line 17) the key is deleted.

When the algorithm finishes with the current key (lines 3 - 23), the next key of the node is visited until all keys of the current node have been checked. We do need to visit all the keys of the node even if `rtree_delete_req` has been called for one of them, because the node MBRs might overlap. This means that even if the MBR of the key we want to delete is inside one of the MBR of the keys of the node (line 3), the subtree of this key might not have the key we want to delete.

If the node's key is a leaf node (line 24) and the node's key matches exactly the search key and refers to the same data (line 25), then `rtree_delete_key` is called to delete the key (line 26). If the page is now empty 2 is returned, if it is not empty 0 is returned and if something went wrong during the deletion -1 is returned.

rtree_delete_key This method deletes a key from a node. An algorithm for this method is not presented because the actions it performs are extremely simple: a node is given and a specific key is deleted from the node. The deletion of a key is much simpler than the method `rtree_add_key` (Section 4.5.1) that needs to perform a series of operations and checks.

4.5.2.3 Comparison with original R*-tree deletion

The deletion algorithm closely follows the original R*-tree. As with the search algorithm the one and major difference with the original algorithm is that the nodes don't keep information about which node is their parent.

```
Input: info, keyinfo, key, key_length, page, page_size, ReinsertList,
       level
Output: Modifies one level in the R-tree: $-1$ for Error, 0 if key was
        deleted, 1 if key was not found, 2 if the leaf is empty
foreach key $k \in$ node do                    /* loop the keys of the node */
    if node is internal then
        if key within k then                              /* rtree_key_cmp  */
            child $\leftarrow$ child page of k;
            res $\leftarrow$ rtree_delete_req (info, keyinfo, key, key_length,
            child, page_size, ReinsertList, level $+ 1$);
            if res $== 0$ then
                if page is adequatly filled then
                    rtree_set_key_mbr (k);               /* store key MBR */
                else
                    add k's child to ReinsertList;
                    rtree_delete_key (k) ;
                if error during the above then
                    return $-1$
                return res
            else if res $== 1$ then                      /* key not found */
                continue the loop and check other keys;
            else if res $== 2$ then              /* last key in leaf page */
                rtree_delete_key;
                if any error during the above then
                    return $-1$;
                return 0;
            else
                return $-1$;
    else                                                     /* Leaf node */
        if key MBR is equal to k and refers to the same data then
        /* rtree_key_cmp  */
            rtree_delete_key;
            if page is now empty then
                return 2;
            else
                return 0;
            if any error during the above then
                return $-1$;
    return 1;
```

Algorithm 4.5.9: `rtree_delete_req`: MyISAM R-tree deletion. Called recursively on each level of the tree.

4.5.3 Search

In this section, we describe how indexes are used during search and specifically how search is performed with MySQL's R-tree keys. First, in Section 4.5.3.1, we present some information about the way indexes are used by the MyISAM storage engine during search operations. Then, we continue with the R-tree specific parts of the storage engine and in Section 4.5.3.2, the algorithms are presented in an abstract way. Then, in Section 4.5.3.3, we dive into more details. Finally, in Section 4.5.3.4, the differences with the original R-tree search algorithm are discussed.

4.5.3.1 MyISAM index search

The search is a bit more complex than the deletion and insertion (Sections 4.5.2 and 4.5.1). The reason for this is that the MySQL classifies the `SELECT` queries into many search modes; 13 in total and 5 of them concern spatial searches (defined in `include/my_base.h`). The API of the storage engines with the core MySQL server, for both deletion and insertion, has a single point of entry. On the contrary, handling searching the data using indexes involves more than 20 storage engine API functions [25, pp. 203–239].

Interface `handler` implementation The MyISAM `handler` implementation is in `storage/myisam/ha_myisam.cc`. The most important storage engine API methods, that are needed to understand the way search is performed, are the following:

- `index_read`: It takes as argument a key and its length and is used to search in the index.
- `index_read_map` This function works like `index_read` but takes as argument a key and bitmap of keys. For example, if a key is created over `KEY(a,b,c,d,e,f)` and the search is performed using for 3 columns only (`WHERE a=1 AND b=2 AND c=3`) the bitmap argument is 000111 (in binary).
- `index_read_idx_map`: The only difference between the `index_read_idx_map` method and the `index_read_map` method is that it takes the index number as an argument. The `handler` class implements this method by converting it into a sequence of `index_read_map`.

- `index_next`: This method can be called after `index_read`, when we want to get the next value of the index, after the last one found. This is used in index scans or for getting all matching values from a non-unique index.

- `index_next_same`: This method is similar to `index_next`, but next row is returned only if it has exactly the same key as the one that was searched for. On the other hand, `index_next` returns the next row independent of its key. The `handler` class implements this method by calling `index_next` and comparing the key of the returned row.

MyISAM methods using indexes The MyISAM storage engine `handler` methods call the MyISAM specific functions that handle the lower level operations, and they are the following:

- `mi_rkey`: reads a row using a key (defined in `storage/myisam/mi_rkey.c`).

- `mi_rnext`: reads the next row with the same key as the previous read (defined in `storage/myisam/mi_rnext.c`).

- `mi_rnext_same`: same as `mi_rnext` but aborts reading if the key has changed (defined in `storage/myisam/mi_rnext_same.c`).

R-tree index methods Finally, the following the methods, that are described in Section 4.5.3.2 in detail, are the index specific methods to access the R-tree index:

- `rtree_find_first`

- `rtree_find_next`

- `rtree_get_first`

- `rtree_get_next`

In Figure 4.2 we summarize the caller graph for all the above mentioned methods.

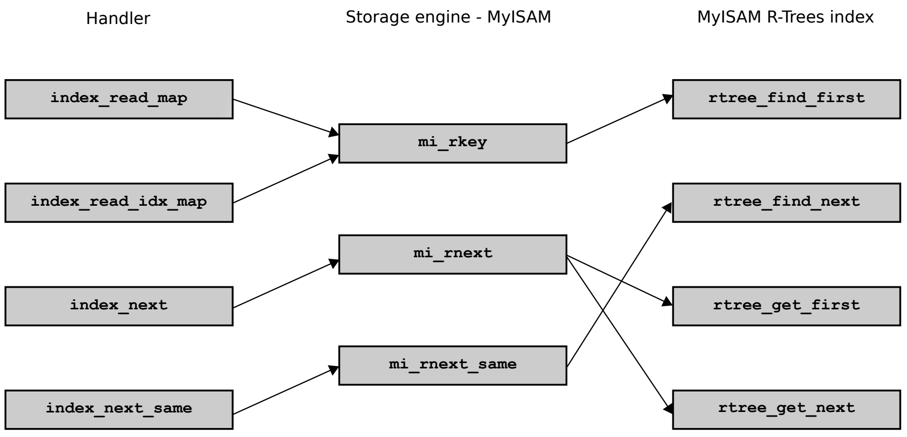


Figure 4.2: Caller graph for the main methods used to search indexes in MyISAM.

4.5.3.2 Abstract description

The code that is associated with the R-tree search is found in the source files `storage/myisam/rt_index.c` and `storage/myisam/rt_key.c`. A high level view of the search is given by the methods:

- `rtree_find_first` presented in Algorithm 4.5.10.
- `rtree_find_next` presented in Algorithm 4.5.11.

The method `rtree_find_first` is called from the root of the tree when index search is used in order to find the first match. It calls `rtree_find_req`. This method recursively calls itself until the proper leaf node is reached and the data is found.

```
Input: key
1 begin
2     return rtree_find_req (key);
3 end
```

Algorithm 4.5.10: `rtree_find_first` abstract: MyISAM R-tree search abstract.

The method `rtree_find_next` is called when index search is used to find the next first match. If the table has changed since the last read, then `rtree_find_first` is called to find the next match (line 3). If the key of the last row can be used and it matches the search criteria, then it returns 0 (line 5). If the next key satisfies the search criteria, the algorithm updates the cursor and returns (line 6).

```
   Input: key
1  begin
2      if table was changed since the last read then
3          return rtree_find_first (key);
4      if key of last row can be used then
5          return 0;
6      return rtree_find_req (key);
7  end
```

Algorithm 4.5.11: `rtree_find_next` abstract: MyISAM R-tree search abstract.

All the methods `rtree_find_first`, `rtree_find_next` and `rtree_find_req` have a second variant. The names of the variants are the same but the "`find`" part of the name is replaced with "`get`". For example the equivalent of `rtree_find_first` is `rtree_get_first`.

These methods have the same input and output and the same flow. The difference between the two variants is that the "`get`" ones are used for index full scans and traverse the index without doing any comparisson at the nodes, whereas the "`find`" variants traverse the index and compare the keys of the nodes with the key currently being searched.

4.5.3.3 Detailed description

This section describes MySQL's R-tree search flow in detail. More specifically the following methods are presented:

- `rtree_find_first` (Algorithm 4.5.12)
- `rtree_find_next` (Algorithm 4.5.13)
- `rtree_find_req` (Algorithm 4.5.14)
- `rtree_get_first` (Algorithm 4.5.15)

- `rtree_get_next` (Algorithm 4.5.16)
- `rtree_get_req` (Algorithm 4.5.17)

Even if we do provide enough details to understand how search is performed, some details fall outside the scope of the description. The description focuses on the fact that somehow the key information can be read, updated and saved, and that nodes can be read and saved permanently but doesn't mention how this is performed. These are important but lower level MyISAM operations and the interested reader can check them directly in the source code files.

rtree_find_first The method is described in Algorithm 4.5.12. It finds the first occurrence of the data that matches the search criteria by calling `rtree_find_req`.

The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. key: key to search for
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. search_flag: flag related to search properties.

Lines 1 to 4 initialize the variables needed for the search. The structure info contains a temporary storage (buff) for the keys which can be used by `mi_rnext`. This function reads the next row after the last row read, using the current index. In line 4 the flag, that marks that reusing the key of the previous row read, is set.

Finally, a recursive search on the tree begins (line 6) and the result of `rtree_find_req` is returned.

rtree_find_next The method is described in Algorithm 4.5.13 and finds the next key during a search. The input arguments of this method are the following:

```
Input: $info$, $keynr$, $key$, $key\_length$, $search\_flag$
Output: $-1$ for Error, 0 if found, 1 if not found
1 $keyinfo \leftarrow$ information from $info$ regarding the index used in search;
2 $info.last\_rkey\_length \leftarrow key\_length$;
3 $info.rtree\_recursion\_depth \leftarrow -1$;
  /* $info.buff$ is a temporary storage for keys */
4 $info.buff\_used \leftarrow 1$;     /* $buff$ has to be reread for rnext */
5 $nod\_cmp\_flag \leftarrow$ MBR_INTERSECT;
6 return rtree_find_req ($info$, $keyinfo$, $search\_flag$, $nod\_cmp\_flag$,
  $root$, 0);
```

Algorithm 4.5.12: `rtree_find_first`: MyISAM R-tree search.

1. info: data structure that includes information about the database table associated with the insertion.

2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. search_flag: flags the describes the search criteria that comes from the MyISAM engine.

First, a check whether the table has changed is performed (line 1). When reading the next row of data the key used for the previous search could be reused. If the table has changed since the last read, then the key must be found again from scratch, by calling `rtree_find_req` (line 2) and the algorithm ends here.

If the table hasn't changed and the key from the previous search can be used to find the next key, then the next keys of the page will be read in a loop (lines 5-9). If a key matches the search criteria (line 6), then this key is used and 0 is returned. Otherwise, the next key of the page is checked (line 9).

If the method has not returned yet, it means that the table has not changed (so the next key of the same page could have be used) but all the keys of the node were checked. So `rtree_find_req` is called to get the next node (line 12).

rtree_find_req The method, described in in Algorithm 4.5.14, is called recursively. It descends the tree towards the leaf nodes in order to find a match. The input arguments of this method are the following:

```
Input: $info$, $keynr$, $search\_flag$
Output: $-1$ for Error, 0 if found, 1 if not found

1  if table has changed and the change was a deletion then
       /* find again the last key                                  */
2      return rtree_find_first ($info$, $keynr$, $lastkey$, $lastkey\_length$,
       $search\_flag$);
3
4  if temporary storage of the key can be reread (for rnext) then
5      while not at end of page do
6          if key matches the search criteria then    /* rtree_key_cmp */
7              $info.lastpos$ ← position of next data;
8              return 0;
9          $key$ ← next key in page;
10
11 $nod\_cmp\_flag$ ← MBR_INTERSECT;
12 return rtree_find_req ($info$, $keyinfo$, $search\_flag$, $nod\_cmp\_flag$,
   $root$, 0);
```

Algorithm 4.5.13: `rtree_find_next`: MyISAM R-tree search.

1. info: data structure that includes information about the database table associated with the insertion.
2. keyinfo: data structure that includes information about the key associated with the insertion.
3. key: is the new leaf key that will be inserted in the tree
4. search_flag: flags the describes the search criteria that comes from the MyISAM engine. It's used for the internal nodes only.
5. nod_cmp_flag: same as `search_flag` but used for the leaf nodes only.
6. page: position of the node in the index.
7. level: the current level of the tree. When `rtree_find_req` descends one level down then this argument is increased by one.

The algorithm loops through all the keys of the node it is currently on. If the node is internal (lines 2–13), the key is matched against the search criteria (line 3). If it does not match the loop continues to the next key. If it matches the search criteria then `rtree_find_req` is called to descend one level down the tree and the result of `rtree_find_req` is checked and the recursion ends here for the current level.

If the node is leaf (line 14), the key is matched against the search criteria (line 15). If it does not match the loop continues to the next key. If it matches the search criteria then the key is saved for later usage and 0 is returned.

Finally, if the loop has finished without a match (line 20), the algorithm returns 1 for failure.

```
   Input: info, keyinfo, search_flag, nod_cmp_flag, page, level
   Output: −1 for Error, 0 if found, 1 if not found
1  foreach key k ∈ page do
2      if node is internal then                           /* page is internal */
3          if k matches the search criteria then          /* rtree_key_cmp    */
4              res ← rtree_find_req;                      /* go one level down */
5              if res == 0 then                           /* found, break recursion */
6                  return res;
7              else if res == 1 then                      /* not found, continue */
8                  info.rtree_recursion_state ← level;
9                  break;
10             if error then
11                 return −1
12
13
14     else                                               /* page is leaf */
15         if k matches the search criteria then          /* rtree_key_cmp    */
16             save position and lenth of next key to info;
17             info.rtree_recursion_state ← level;
18
19
   /* loop finished and match wasn't found                                  */
20 return 1;
```

Algorithm 4.5.14: `rtree_find_req`: MyISAM R-tree search. Called recurcively on each level of the tree.

rtree_get_first The method is described in Algorithm 4.5.15 and it flows similar to `rtree_find_first` (Algorithm 4.5.12). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.

2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

Lines 1 to 3 initialize the variables needed for the search. The structure info contains a temporary storage (buff) for the keys which can be used by `mi_rnext`. This function reads the next row after the last row read, using the current index. In line 3 the flag, that marks that reusing the key of the previous row read, is set.

Finally, a recursive search on the tree begins (line 4) and the result of `rtree_get_req` is returned.

```
Input: $info$, $keynr$, $key\_length$
Output: $-1$ for Error, 0 if found, 1 if not found
1 $keyinfo \leftarrow$ information from $info$ regarding the index used in search;
2 $info.rtree\_recursion\_depth \leftarrow -1$;
  /* $info.buff$ is a temporary storage for keys */
3 $info.buff\_used \leftarrow 1$;      /* $buff$ has to be reread for rnext */
4 return rtree_get_req ($info$, $keyinfo$, $key\_length$, $root$, 0);
```

Algorithm 4.5.15: `rtree_get_first`: MyISAM R-tree search.

rtree_get_next The method is described in Algorithm 4.5.16 and its flow similar to `rtree_find_next` (Algorithm 4.5.13). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.

2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

The method checks if the next key is on the same page and if the page has not changed (line 1). If one of the two is not valid, then `rtree_get_req` is called to find the next key (line 5). If both are valid, then the position of the next data is stored and the method returns successfully (line 3).

```
Input: $info$, $keyinfo$, $key\_length$, $page$, $level$
Output: $-1$ for Error, 0 if found, 1 if not found

1 if next key is on the same page and page has not changed then
2     $info.lastpos \leftarrow$ position of next data;
3     return 0;
4
5 return rtree_get_req ($info$, $keyinfo$, $key_length$, $root$, 0);
```

Algorithm 4.5.16: `rtree_get_next`: MyISAM R-tree search.

rtree_get_req The method, described in in Algorithm 4.5.17, is called recursively and its flow similar to `rtree_find_req` (Algorithm 4.5.14). It descends the tree towards the leaf nodes in order to find the next row of the search based on information about the last row read and key used. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.

2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

4. page: position of the node in the index.

5. level: the current level of the tree. When `rtree_get_req` descends one level down then this argument is increased by one.

The method scans in a loop all the keys of a node (lines 1–15). If the node is internal (lines 2–11), the algorithm descends one level down the tree by calling `rtree_get_req`. If a node was found (line 4), the result is returned. If a node was not found, the algorithm continues to the next key in the loop (line 6). Finally if an error occurred the algorithm terminates (line 9).

If the node is a leaf (lines 12–15), the position of the next row is saved and the algorithm returns.

Finally, if the loop has examined all the keys of the node it returns 1 (line 16).

```
   Input: info, keyinfo, key_length, page, level
   Output: −1 for Error, 0 if found, 1 if not found
1  foreach key k ∈ page do
2      if node is internal then                     /* page is internal */
3          res ← rtree_get_req;                     /* go one level down */
4          if res == 0 then      /* node was found, break recursion */
5              return res;
6          else if res == 1 then                  /* not found, continue */
7              info.rtree_recursion_state ← level;
8              break;
9          if error then
10             return −1
11
12     else                                              /* page is leaf */
13         save position and lenth of next key to info;
14         info.rtree_recursion_state ← level;
15
   /* loop finished and all keys were examined                        */
16 return 1;
```

Algorithm 4.5.17: `rtree_get_req`: MyISAM R-tree search. Called recurcively on each level of the tree.

4.5.3.4 Comparison with original R*-tree search algorithm

The search algorithm is very close to the original R*-tree and R-tree search algorithms. Method `rtree_find_req` (Algorithm 4.5.14) is quite similar to the method `RangedSearch` (Algorithm 2.1.1), that follows closely the way search is performed in B-trees. The rest of the search methods are wrappers around `rtree_find_req` and facilitate the way the core MySQL server performs search using indexes. `rtree_find_first` finds the first row occurrence, and `rtree_find_next` finds the next row to read, both using `rtree_find_req` when needed.

```
$ wc -l storage/myisam/ha_myisam.cc storage/myisam/ha_myisam.h \
 storage/myisam/rt_index.c storage/myisam/rt_index.h storage/myisam/rt_key.c \
 storage/myisam/rt_key.h storage/myisam/rt_mbr.h storage/myisam/mi_search.c \
 storage/myisam/mi_delete.c storage/myisam/mi_write.c storage/myisam/mi_open.c \
 storage/myisam/mi_rkey.c storage/myisam/mi_rnext.c  \
 storage/myisam/mi_rnext_same.c   include/my_base.h

  2412 storage/myisam/ha_myisam.cc
   179 storage/myisam/ha_myisam.h
  1126 storage/myisam/rt_index.c     *
    45 storage/myisam/rt_index.h     *
   106 storage/myisam/rt_key.c       *
    31 storage/myisam/rt_key.h       *
    36 storage/myisam/rt_mbr.h
  1925 storage/myisam/mi_search.c
   894 storage/myisam/mi_delete.c
  1050 storage/myisam/mi_write.c
  1366 storage/myisam/mi_open.c
   266 storage/myisam/mi_rkey.c
   157 storage/myisam/mi_rnext.c
   127 storage/myisam/mi_rnext_same.c
   599 include/my_base.h
 10319 total
```

Figure 4.3: Files investigated for the reasearch of Section 4.5.

4.5.4 MyISAM R-tree summary

In this section we summarize the way R-trees are handled in the MyISAM storage engine. As we discussed the search, deletion and insertion methods resemble the original R*-tree methods. The main differences are found in the fact that the keys don't keep information about their parent keys. This forces the tree operation to be performed in a clear recursive way and the changes that must be performed to a tree level due to changes to lower levels are done immediately after the method returns from the lower level. Moroever, the search is wrapped around a method that follows R-tree closely, in order to handle the many search modes of the MySQL core server.

In Figure 4.3 we present the source code files were read in order to perform the research of the Section 4.5. First the Linux bash command `wc`, that counts the lines of the files given as arguments is given. Then follows a list and in each row there is the number of lines in the file (including whitespace and comments) and the path of the file. The asterisc (∗) marks the files where most of the Algorithms presented in the Section 4.5 are found. The last line of the list shows the sum of lines in all the files we investigated (around 10K lines).

4.6 Summary

This chapter was a thorough introduction to MySQL internals, and we begun this introduction by defining the codebase we worked with. A high level overview of the MySQL server's architecture was given, as well as the path an SQL query follows from the moment it reaches the server until data is read from the storage. MyISAM, one of MySQL's main storage engines and the storage engine we used for our implementation, was then presented. Finally, the way MySQL and MyISAM currently perform spatial indexing was extensively discussed.

Chapter 5

GiST Implementation

This chapter presents the implementation part of the research. Based on the knowledge discussed in Chapter 4, we implemented our own GiST-based index solution for the MyISAM storage engine of MySQL. In Section 5.1, we begin by describing the changes needed to make the MySQL server GiST aware. Then, in Section 5.2 we discuss the core implementation of the indexes and in Section 5.3 we dive into the details of the index algorithms. Finally, we conclude in Section 5.6.

For a complete and working GiST implementation both the code of Sections 5.1 and 5.2 is required. The implementation process itself was split in these two steps, so it made sense for the presentation to follow the same logic.

The code was based on the latest 5.5 version (currently 5.5.27). The source code of MariaDB is required in order to follow the description of the patches. Directions and details for downloading and compiling the source for Linux Debian based systems are given in Appendix A.

5.1 Making MySQL GiST-aware

In this section we discuss the changes that we performed to the codebase in order to make the server "aware" of GiSTs. After these modifications are applied, the GiST indexes are hooked in the MySQL server and the MyISAM storage engine. However, the indexes are only skeleton implementations and their full implementation is discussed in Section 5.2.

First, in Section 5.1.1 the changes necessary in the build infrastructure are presented. Then in Section 5.1.2 the changes needed to extend the SQL parser are shown. In Section 5.1.3 we discuss the changes required in the MySQL core server and finally in Section 5.1.4 the changes to the MyISAM storage engine. Finally, in Section 5.1.5 we present the changes required for a GiST skeleton implementation.

All the code changes discussed in this Section can be found in Section B.1 in a diff format. The paths of all the files are relative to the directory of the source code.

5.1.1 Changes in the build infrastructure

In this section we present the changes to the build infrastructure. MariaDB uses `cmake` for building and we made it aware of the new files and generic flags required to build the MariaDB server with GiST enabled.

storage/myisam/CMakeLists.txt We added the new files required to build MyISAM with GiSTs. The `gist-*` files include the index implementation but it's a skeleton one, and used only to keep the compiler and the linker happy. In Section 5.2 these files are enhanced to include the full implementation.

config.h.cmake We added the C preprocessor flag `HAVE_GIST_KEYS` that is used to wrap the GiST-related code. Figure 5.1, presents two examples of using such a preprocessor flag. In the first one, the code used to call a feature, is called only if the `HAVE_SOMETHING` has been defined. In the second example, the code used to call a feature is wrapped in the true part of the `ifdef` and if the `HAVE_SOMETHING` has not been defined, an exception is thrown.

```
#ifdef HAVE_SOMETHING
    call_feature_something();
#endif


#ifdef HAVE_SOMETHING
    call_feature_something();
#else
    throw a debug assertion
#endif
```

Figure 5.1: Examples of using a C preprocessor flag in the code

5.1.2 Changes in the SQL parser

The SQL parser of MySQL is implemented with the Bison parser generator, that in turn is compatible with Yacc. The Bison generator accepts as input the definition of a grammar as well as hooks for specific actions, and produces a C program that can parse the given grammar and execute the defined hooks [9].

Parsing of SQL for the creation of keys occurs in two SQL commands: `CREATE TABLE` and `CREATE INDEX` [55, 54].

We have extended the current SQL syntax to accept two new types of indexes: a GiST for the R*-tree index and GiST for the original R-tree. Both types of indexes belong to the `SPATIAL` index type.

In Figure 5.2 we present the changes in the syntax of the `CREATE INDEX` SQL command. Lines 1–16 show the current syntax and lines 18–33 the new one. The command has been extended (lines 11 and 28) to accept the new types of indexes. The same change was applied for the `CREATE TABLE` command.

After the changes the `CREATE TABLE` and `CREATE INDEX` SQL commands of Figure 5.3 are valid.

sql/lex.h In this file we only define the two new SQL keywords `GIST_RSTAR` and `GIST_RGUT83`.

sql/sql_yacc.yy This file describes the syntax of the SQL language that MySQL can parse. We changed the parser for the SQL commands `CREATE`

```
# current syntax
CREATE [ONLINE|OFFLINE] [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
    [index_type]
    ON tbl_name (index_col_name,...)
    [index_option] ...

index_col_name:
    col_name [(length)] [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name

# new syntax
CREATE [ONLINE|OFFLINE] [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
    [index_type]
    ON tbl_name (index_col_name,...)
    [index_option] ...

index_col_name:
    col_name [(length)] [ASC | DESC]

index_type:
    USING {BTREE | HASH | GIST_RSTAR | GIST_RGUT83}

index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
```

Figure 5.2: Valid `CREATE TABLE` and `CREATE INDEX` SQL commands with GiST index types

```
CREATE TABLE ‘t1‘ (
  ‘c1‘ geometry NOT NULL,
  SPATIAL KEY ‘idx1‘ (‘c1‘) USING GIST_RSTAR
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE SPATIAL INDEX ‘idx2‘ ON t1 (c1) USING GIST_RGUT83;
```

Figure 5.3: Valid `CREATE TABLE` and `CREATE INDEX` SQL commands with GiST index types

TABLE and CREATE INDEX, so that they can accept the new index types after the SQL keyword USING.

client/mysql.cc We define the existence of two new SQL keywords GIST_RSTAR and GIST_RGUT83 to the mysql command line client tool.

5.1.3 Changes in the MySQL core server

In this section we present the changes required to the core MySQL server, in order to enable various aspects of the GiST indexes.

include/maria.h A simple comment change to remind us of the presence of GiST indexes.

include/myisam.h A simple comment change to remind us of the presence of GiST indexes.

include/my_base.h This file is used for a number of server-wide data structures. In the C enumeration ha_key_alg we added the two new types of indexes GIST_RSTAR and GIST_RGUT83 that are mapped to the internal values HA_KEY_ALG_GIST_RSTAR and HA_KEY_ALG_GIST_RGUT83 accordingly. Moreover, the flag HA_GIST_INDEX is used to mark the indexes as of type GiST.

sql/handler.h In the handler interface we added a preprocessor macro that returns 1 if the storage engine is capable of GiST indexes.

sql/sql_show.cc The methods in the this file are responsible for the SQL command SHOW TABLE. It returns a string that is the CREATE TABLE command that corresponds to this table. We added code that handles the presence of GiST indexes.

In the next three set of changes, we added code that deals with server variables. These are variable that the user can use to interact with the server and configure it [58]. For the server-user interaction SQL or configuration files are used.

sql/mysqld.cc We added the definition of the server variable `have_gist_keys`.

sql/set_var.h We added the definition of the server variable `have_gist_keys` for the SQL command `SET variable`. We noticed that in these values are the same ones in the file `sql/mysqld.cc` and that are redefined. An improvement for the MySQL codebase would be to use one common place for these definitions.

sql/sys_vars.cc We added the code that returns the value of the server variable `have_gist_keys`.

5.1.4 Changes in the MyISAM storage engine

In this section we present the changes required to add the GiST indexes in MyISAM. The MyISAM storage engine supports, B-tree, fulltext and spatial indexes and there is already code that checks the type of the index and performs the proper operations. We added checks to this code that handle the GiST index type.

storage/myisam/mi_check.c In the function that checks if the key type matches the data record we added code that handles the GiST indexes.

storage/myisam/mi_create.c We simply added debugging code.

storage/myisam/mi_open.c The methods in this file are responsible for the proper opening of the table. We added code to check the type of the index, mark the presence of GiST indexes and map the GiST functions with the key's data structure.

storage/myisam/myisamdef.h This file defines several data structures used by MyISAM. We add in the `MYISAM_INFO` data structure two fields that store the depth and the state of the recursion when traversing the GiST tree.

storage/myisam/ha_myisam.cc We added code that informs the server via the handler API that the MyISAM storage engine can handle GiST indexes.

In the next three set of changes, we added code to check the existence of GiST indexes during search related operations.

storage/myisam/mi_rkey.c The methods in this file are responsible for reading a data record using a key.

storage/myisam/mi_rnext.c The methods in this file are responsible for reading the next row (in the index order) after a successful index read.

storage/myisam/mi_rnext_same. The methods in this file are responsible for reading the next row (in the index order) with the same key as previous read.

5.1.5 Changes for a GiST skeleton implementation

In this section we present the changes required to the core MySQL server, in order to implement a skeleton version of the GiST indexes. The skeleton version doesn't provide any indexing functionality like insertion, deletion or searching. However, it was implemented for the following reasons:

- It allows for the creation of tables and indexes with GiST indexes.
- It provides all the necessary points and hooks in the code to start implementing the actual functionality of the index.
- While providing the above point, it keeps the compiler and linker happy in order to build the MySQL server.

The files we added were the following:

- `storage/myisam/gist_index.c`
- `storage/myisam/gist_index.h`
- `storage/myisam/gist_key.c`
- `storage/myisam/gist_key.h`

5.2 GiST implementation

In this section we discuss the changes that we performed to the codebase in order to implement the GiST functionality. In Section 5.2.1 we show the changes to the build infrastructure, and in Section 5.2.2 the changes to the MyISAM storage engine. Then in Section 5.2.3 we describe the debugging code we added to some methods. Finally, in Section 5.2.4 we present the changes for the core GiST implementation and in Section 5.2.5 the tests we added.

All the code changes are presented in Section B.2 in a diff format. The paths of all the files are relative to the directory of the source code.

5.2.1 Changes in the build infrastructure

File `storage/myisam/CMakeLists.txt` was modified in order to accommodate the new GiST-related files.

5.2.2 Changes in the MyISAM storage engine

File `storage/myisam/mi_range.c` contains code that gives an estimation for the number of records that exist between two keys. We added code that handles the GiST indexes.

5.2.3 Changes for debugging information

In this section we present the files where we added debug code:

- `storage/myisam/ha_myisam.cc`
- `storage/myisam/mi_check.c`
- `storage/myisam/mi_open.c`
- `storage/myisam/mi_rkey.c`
- `storage/myisam/mi_rnext.c`
- `storage/myisam/mi_rnext_same.c`

- `storage/myisam/mi_search.c`
- `storage/myisam/mi_dynrec.c`
- `storage/myisam/mi_write.c`
- `storage/myisam/mi_key.c`

The reason we added more debug code is that we wanted to be able to monitor the operations on the indexes through the trace file MySQL produces when it runs with debug code enable. Adding debug code doesn't cause any performance penalty at all in the non-debug version of the server, since the debug code is implemented only with C preprocessor code, that can turn on and off the presence of debug code.

5.2.4 Changes for the GiST indexes

In this section we present the changes we performed for the core of the GiST implementation. We show how we stripped the files related to R-tree from any code that could be re-used in other indexes and moved it in common files, and what kind of functionality was added in the GiST-related files. A detailed analysis of the GiST tree algorithms follows in Section 5.3.

In the following two sets of changes we describe the code that could be re-used for other indexes from `rt_index` files.

storage/myisam/rt_index.c We removed code that defined data structures used for the reinsertion of nodes during deletion, since this code could be re-used from other indexes too. We also added debug code.

storage/myisam/rt_index.h We redefined some functions from being `static` (in the C meaning) to non-static functions so that they are accessible for other R-tree-like indexes.

In the following four sets of changes we describe the re-usable code that was moved from the `rt_index` files.

storage/myisam/sp_reinsert.h Definitions of methods related to re-insertion of nodes was moved from `rt_index` files into a separated header file.

storage/myisam/gist_functions.c We moved here code that is related to splitting nodes and adjusting the node's keys. For the moment this is a simple wrapper around the existing `rtree` functionality.

storage/myisam/gist_functions.h The definitions of the methods in file `storage/myisam/gist_functions.c`.

In the following four set of changes we describe the functionality that was added to the files related closely to the GiST implementation.

storage/myisam/gist_index.c In this file the code related to the insertion, deletion and search of GiST trees was added. A detailed analysis of the algorithms is found in Section 5.3.

storage/myisam/gist_index.h The definitions of the methods in file `storage/myisam/gist_index.c`.

storage/myisam/gist_key.c In this file we added code that is related to adding, deleting and comparing nodes of the GiST tree.

storage/myisam/gist_key.h The definitions of the methods in file `storage/myisam/gist_key.h`.

5.2.5 Changes for testing the GiST implementation

In order to test the implementation of the GiST index we used the MySQL testing suite. The testing procedure is described in detail in Section 5.5. The test we added was the file `mysql-test/t/gis-gist.test`.

5.3 Analysis of the GiST algorithms

In this section we present the details of the algorithms that we implemented. The files where the GiST implementation is found are the files `storage/myisam/gist_*` (as already described in Section 5.2.4).

The basic idea behind the implementation is to wrap the GiST functionality around the existing R-tree. In Sections 4.5.3.4, 4.5.1.3 and 4.5.2.3 we have already noticed the similarity of MySQL's R-tree implementation with the original R*-tree algorithms. Moreover, as we have already noticed from Sections 2.2.3, 2.2.4 and 2.2.5 the GiST algorithms are similar to the algorithms of B-tree and R-tree algorithms.

These similarities have driven our implementation and the reader will notice a similarity in the search, deletion and insertion algorithms between the existing R*-tree implementation (presented in Section 4.5) and our new implementation GiST presented here. Additionally, we kept the same naming conventions in order to make following browsing the code easier to a reader experienced with the MyISAM codebase. Last but not least, the GiST implementation has the same interface with the rest of the MyISAM code, as the existing R-tree has. This helps the implementation itself, since the changes in non-related places are kept to a minimum.

In Section 5.3.1 we present the search functionality, in Section 5.3.2 the deletion functionality and finally in Section 5.3.3 the insertion.

5.3.1 GiST search

In this section, we describe how searching is performed. The reader will notice a similarity between the search algorithm in this section and the existing R-tree MySQL indexes (Section 4.5.3).

First, in Section 5.3.1.1 we describe in an abstract level the way GiST search operates. Then, in Section 5.3.1.2, take a closer look to the details. Finally, in Section 5.3.1.3, the differences with the original GiST search algorithm are discussed.

5.3.1.1 Abstract description

The code associated with the GiST search is found in the source files `storage/myisam/gist_*`. A high level view of the search is given by the methods:

- `gist_find_first` presented in Algorithm 5.3.1.
- `gist_find_next` presented in Algorithm 5.3.2.

The method `gist_find_first` is called from the root of the tree in order to find the first match and it calls `gist_find_req`. This method recursively calls itself until the correct leaf node is reached and the data is found.

```
Input: key
1 begin
2     return gist_find_req (key);
3 end
```

Algorithm 5.3.1: `gist_find_first` abstract: MyISAM GiST search abstract.

The method `gist_find_next` is called to find the next match. If the table has changed since the last read, then `gist_find_first` is called to find the next match (line 3). If the key of the last row can be used and it matches the search criteria, then it returns 0 (line 5). If the next key satisfies the search criteria, the algorithm updates the cursor and returns (line 6).

```
Input: key
1 begin
2     if table was changed since the last read then
3         return gist_find_first (key);
4     if key of last row can be used then
5         return 0;
6     return gist_find_req (key);
7 end
```

Algorithm 5.3.2: `gist_find_next` abstract: MyISAM GiST search abstract.

All the methods `gist_find_first`, `gist_find_next` and `gist_find_req` have a second variant, as their `rtree_*` equivalents do. The names of the variants are the same but the "`find`" part of the name is replaced with "`get`". For example the equivalent of `gist_find_first` is `gist_get_first`.

These methods have the same input and output and the same flow. The difference between the two variants is that the "`get`" ones are used for index full scans and traverse the index without doing any comparisson at the nodes, whereas the "`find`" variants traverse the index and compare the keys of the nodes with the key currently being searched.

5.3.1.2 Detailed description

This section describes MySQL's R-tree search flow in detail. More specifically the following methods are presented:

- `gist_find_first` (Algorithm 5.3.3)
- `gist_find_next` (Algorithm 5.3.4)
- `gist_find_req` (Algorithm 5.3.5)
- `gist_get_first` (Algorithm 5.3.6)
- `gist_get_next` (Algorithm 5.3.7)
- `gist_get_req` (Algorithm 5.3.8)

Even if we do provide enough details to understand how search is performed, some details fall outside the scope of the description. The description focuses on the fact that somehow the key information can be read, updated and saved, and that nodes can be read and saved permanently but doesn't mention how this is performed. These are important but lower level MyISAM operations and the interested reader can check them directly in the source code files.

gist_find_first The method is described in Algorithm 5.3.3. It finds the first occurrence of the data that matches the search criteria by calling `gist_find_req`.

The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. key: key to search for
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. search_flag: flag related to search properties.

Lines 1 to 4 initialize the variables needed for the search. The structure info contains a temporary storage (buff) for the keys which can be used by `mi_rnext`. This function reads the next row after the last row read, using the current index. In line 4 the flag, that marks that reusing the key of the previous row read, is set.

Finally, a recursive search on the tree begins (line 6) and the result of `gist_find_req` is returned.

```
Input: $info$, $keynr$, $key$, $key\_length$, $search\_flag$
Output: $-1$ for Error, 0 if found, 1 if not found

1 $keyinfo \leftarrow$ information from $info$ regarding the index used in search;
2 $info.last\_rkey\_length \leftarrow key\_length$;
3 $info.gist\_recursion\_depth \leftarrow -1$;
  /* $info.buff$ is a temporary storage for keys */
4 $info.buff\_used \leftarrow 1$;        /* $buff$ has to be reread for rnext */
5 $nod\_cmp\_flag \leftarrow$ MBR_INTERSECT;
6 return gist_find_req ($info$, $keyinfo$, $search\_flag$, $nod\_cmp\_flag$, $root$,
  0);
```

Algorithm 5.3.3: `gist_find_first`: MyISAM GiST search.

gist_find_next The method is described in Algorithm 5.3.4 and finds the next key during a search. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. search_flag: flags the describes the search criteria that comes from the MyISAM engine.

First, a check whether the table has changed is performed (line 1). When reading the next row of data the key used for the previous search could be reused. If the table has changed since the last read, then the key must be found again from scratch, by calling `gist_find_req` (line 2) and the algorithm ends here.

If the table hasn't changed and the key from the previous search can be used to find the next key, then the next keys of the page will be read in a loop (lines 5-

9). If a key matches the search criteria (line 6), then this key is used and 0 is returned. Otherwise, the next key of the page is checked (line 9).

If the method has not returned yet, it means that the table has not changed (so the next key of the same page could have be used) but all the keys of the node were checked. So `gist_find_req` is called to get the next node (line 12).

```
Input: $info$, $keynr$, $search\_flag$
Output: $-1$ for Error, 0 if found, 1 if not found

1  if table has changed and the change was a deletion then
       /* find again the last key                                    */
2      return gist_find_first ($info$, $keynr$, $lastkey$, $lastkey\_length$,
       $search\_flag$);
3
4  if temporary storage of the key can be reread (for rnext) then
5      while not at end of page do
6          if key matches the search criteria then     /* gist_key_cmp  */
7              $info.lastpos \leftarrow$ position of next data;
8              return 0;
9          $key \leftarrow$ next key in page;
10
11 $nod\_cmp\_flag \leftarrow$ MBR_INTERSECT;
12 return gist_find_req ($info$, $keyinfo$, $search\_flag$, $nod\_cmp\_flag$, $root$,
   0);
```
Algorithm 5.3.4: `gist_find_next`: MyISAM GiST search.

gist_find_req The method, described in in Algorithm 5.3.5, is called recursively. It descends the tree towards the leaf nodes in order to find a match. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keyinfo: data structure that includes information about the key associated with the insertion.
3. key: is the new leaf key that will be inserted in the tree
4. search_flag: flags the describes the search criteria that comes from the MyISAM engine. It's used for the internal nodes only.

5. nod_cmp_flag: same as `search_flag` but used for the leaf nodes only.
6. page: position of the node in the index.
7. level: the current level of the tree. When `gist_find_req` descends one level down then this argument is increased by one.

The algorithm loops through all the keys of the node it is currently on. If the node is internal (lines 2–13), the key is matched against the search criteria (line 3). If it does not match the loop continues to the next key. If it matches the search criteria then `gist_find_req` is called to descend one level down the tree and the result of `gist_find_req` is checked and the recursion ends here for the current level.

If the node is leaf (line 14), the key is matched against the search criteria (line 15). If it does not match the loop continues to the next key. If it matches the search criteria then the key is saved for later usage and 0 is returned.

Finally, if the loop has finished without a match (line 20), the algorithm returns 1 for failure.

gist_get_first The method is described in Algorithm 5.3.6 and it flows similar to `gist_find_first` (Algorithm 5.3.3). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

Lines 1 to 3 initialize the variables needed for the search. The structure info contains a temporary storage (buff) for the keys which can be used by `mi_rnext`. This function reads the next row after the last row read, using the current index. In line 3 the flag, that marks that reusing the key of the previous row read, is set.

Finally, a recursive search on the tree begins (line 4) and the result of `gist_get_req` is returned.

```
Input: info, keyinfo, search_flag, nod_cmp_flag, page, level
Output: −1 for Error, 0 if found, 1 if not found
1  foreach key k ∈ page do
2      if node is internal then                          /* page is internal */
3          if k matches the search criteria then         /* gist_key_cmp  */
4              res ← gist_find_req;                      /* go one level down */
5              if res == 0 then                          /* found, break recursion */
6                  return res;
7              else if res == 1 then                     /* not found, continue */
8                  info.gist_recursion_state ← level;
9                  break;
10             if error then
11                 return −1
12
13
14     else                                              /* page is leaf */
15         if k matches the search criteria then         /* gist_key_cmp  */
16             save position and lenth of next key to info;
17             info.gist_recursion_state ← level;
18
19
   /* loop finished and match wasn't found                                    */
20 return 1;
```

Algorithm 5.3.5: `gist_find_req`: MyISAM GiST search. Called recurcively on each level of the tree.

```
Input: info, keynr, key_length
Output: −1 for Error, 0 if found, 1 if not found
1  keyinfo ← information from info regarding the index used in search;
2  info.gist_recursion_depth ← −1;
   /* info.buff is a temporary storage for keys                               */
3  info.buff_used ← 1;          /* buff has to be reread for rnext */
4  return gist_get_req (info, keyinfo, key_length, root, 0);
```

Algorithm 5.3.6: `gist_get_first`: MyISAM GiST search.

gist_get_next The method is described in Algorithm 5.3.7 and its flow similar to `gist_find_next` (Algorithm 5.3.4). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

The method checks if the next key is on the same page and if the page has not changed (line 1). If one of the two is not valid, then `gist_get_req` is called to find the next key (line 5). If both are valid, then the position of the next data is stored and the method returns successfully (line 3).

```
Input: $info$, $keyinfo$, $key\_length$, $page$, $level$
Output: $-1$ for Error, 0 if found, 1 if not found

1 if next key is on the same page and page has not changed then
2     $info.lastpos \leftarrow$ position of next data;
3     return 0;
4
5 return gist_get_req ($info$, $keyinfo$, $key_length$, $root$, 0);
```

Algorithm 5.3.7: `gist_get_next`: MyISAM GiST search.

gist_get_req The method, described in in Algorithm 5.3.8, is called recursively and its flow similar to `gist_find_req` (Algorithm 5.3.5). It descends the tree towards the leaf nodes in order to find the next row of the search based on information about the last row read and key used. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

4. page: position of the node in the index.

5. level: the current level of the tree. When `gist_get_req` descends one level down then this argument is increased by one.

The method scans in a loop all the keys of a node (lines 1–15). If the node is internal (lines 2–11), the algorithm descends one level down the tree by calling `gist_get_req`. If a node was found (line 4), the result is returned. If a node was not found, the algorithm continues to the next key in the loop (line 6). Finally if an error occurred the algorithm terminates (line 9).

If the node is a leaf (lines 12–15), the position of the next row is saved and the algorithm returns.

Finally, if the loop has examined all the keys of the node it returns 1 (line 16).

```
   Input: info, keyinfo, key_length, page, level
   Output: −1 for Error, 0 if found, 1 if not found
1  foreach key k ∈ page do
2      if node is internal then                              /* page is internal */
3          res ← gist_get_req;                               /* go one level down */
4          if res == 0 then             /* node was found, break recursion */
5              return res;
6          else if res == 1 then                         /* not found, continue */
7              info.gist_recursion_state ← level;
8              break;
9          if error then
10             return −1
11
12     else                                                      /* page is leaf */
13         save position and lenth of next key to info;
14         info.gist_recursion_state ← level;
15
   /* loop finished and all keys were examined                                */
16 return 1;
```

Algorithm 5.3.8: `gist_get_req`: MyISAM GiST search. Called recursively on each level of the tree.

5.3.1.3 Comparison with original R*-tree search algorithm

The search algorithm is very close to the original GiST search algorithm. However, we haven't implemented the full abstraction that GiST provides. The important missing part is the `Union` and `Compare` functionality. Even if it is currently not implemented, when we do implement them, the flow of the algorithm will not change. The changes will have to be performed to the lower level method `gist_key_cmp`. This will simply call the appropriate `Compare` methods for the specific variant the GiST index abstracts.

5.3.2 GiST deletion

In this section, we describe the algorithm for the deletion of GiST keys. The reader will notice a similarity between the deletion algorithms in this section and the existing R-tree MySQL indexes (Section 4.5.2).

In Section 5.3.2, the algorithm is presented in an abstract way and then, in Section 5.3.2.2 with more details. Finally, in Section 5.3.2.3, the differences with the original GiST deletion algorithm are discussed.

5.3.2.1 Abstract description

The code that is associated with the GiST deletion is found in the source files `storage/myisam/gist_*`. A high level view of the deletion flow is presented in Algorithm 5.3.9.

The method `gist_delete` is called from the root of the tree and then it calls `gist_delete_req`. This method recursively calls itself until the proper leaf node is reached and the key is deleted (line 2). During this process some nodes might require reinsertion. This is performed after `gist_delete_req` has returned (line 3). Reinserting is required when some of the nodes become filled less than their minimum fill factor during the deletion process,.

5.3.2.2 Detailed description

This section describes MySQL's GiST deletion flow in detail. More specifically the following methods are presented:

```
Input: key
begin
    gist_delete_req (key);
    Reinsert deleted nodes;
end
```

Algorithm 5.3.9: `gist_delete` abstract: MyISAM GiST deletion abstract.

- `gist_delete` (Algorithm 5.3.10)
- `gist_delete_req` (Algorithm 5.3.11)
- `gist_delete_key`

Even if we do provide enough details to understand how deletion is performed, some details fall outside the scope of the description. The description focuses on the fact that somehow the key information can be read, updated and saved, and that nodes can be read and saved permanently, but doesn't mention how this is performed. These are important but lower level MyISAM operations and the interested reader can check directly in the source code files.

gist_delete The method is described in Algorithm 5.3.10, and it is the single point of entry for deleting a key from the index. It modifies the index by deleting one key and returns `0` for success and `-1` if something went wrong (same as `gist_insert` in Algorithm 5.3.14). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the deletion.
2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.
3. key: is the leaf key that will be deleted in the tree
4. key_length: is the key length. Keys can have different lengths because they can be of columns of data types with different size.

First, the key's information are taken from the table data structure (line 1) as well as the root of the tree. Also, an empty list, that can accommodate nodes that will be re-inserted, is created (line 3). Then `gist_delete_req` is called. This method calls itself recursively and descends the tree until the leaf nodes are reached. Then, it deletes the keys. During this process, some nodes might become filled less than the fill factor and must be re-inserted. They are deleted from the tree and they are appended to ReinsertList. Once method `gist_delete_req` returns, the re-insertion takes place (line 6). The method `gist_insert_level` (line 9), described in Algorithm 5.3.15, is called to insert either leaf nodes or internal nodes. For internal nodes it reinserts the keys of the internal nodes. The subtrees of the internal node's keys are left untouched.

```
   Input: $info$, $keynr$, $key$, $key\_length$
   Output: Modifies GiST: $-1$ for Error, $0$ if key was deleted

 1 $keyinfo \leftarrow$ take key information from $info$;
 2 $old\_root \leftarrow$ take root node from $info$;
 3 $ReinsertList \leftarrow$ empty list of pages;

 4 $res \leftarrow$ gist_delete_req ($info$, $keyinfo$, $key$, $key\_length$, $old\_root$,
   $page\_size$, $ReinsertList$, 0);

 5 if $res == 0$ then                                        /* not split */
 6     foreach $page\ i \in ReinsertList$ do
 7         foreach $key\ k \in ReinsertList.[i]$ do
 8             $l \leftarrow ReinsertList.pages.[i].[k].level$;
 9             gist_insert_level ($info$, $keynr$, $k$, $key\_length$, $l$);
10             if $root\ was\ split\ and\ tree\ grew\ one\ level$ then
11                 $\forall$ remaing pages and keys increase by one the re-insertion
                   level;
12             if $any\ error\ during\ the\ above$ then
13                 return $-1$;
14     return 0;
15 else if $res == 1$ then                                   /* key not found */
16     return $-1$;
17 else if $res == 2$ then                                   /* tree is now empty */
18     return 0;
19 else
20     return $-1$;
```

Algorithm 5.3.10: `gist_delete`: MyISAM GiST deletion. Called from the root of the tree.

gist_delete_req The method, described in Algorithm 5.3.11, is called recursively in order to modify one level of the tree. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the deletion.
2. keyinfo: data structure that includes information about the key associated with the insertion.
3. key: is the leaf key that will be deleted from the tree
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. page: the current page that is operated.
6. page_size: total size of keys on the current page.
7. ReinsertList: the list of nodes that might require to be re-inserted after deletion has finished.
8. level: the current level of the tree. When `gist_delete_req` descends one level down then this argument is increased by one.

First, for each node that is visited all the keys are checked (line 1) in a loop. If the node is internal (line 2) and if the key to delete MBR in inside the node's key MBR (line 3), then `gist_delete_req` is called for the child node (line 5). Otherwise the loop visits the next key of the node.

Once `gist_delete_req` returns, the algorithm takes different actions depending on the returned value. If the deletion was successful (returned 0 - line 6), the fill of the page is checked (line 7) and if it is below the fill factor the node is appended to the ReinsertList and `gist_delete_key` is called to delete the key (line 11). If the key for deletion was not in the subtree just checked (returned 1 - line 15) visit the next key of the node (line 1). If the child node was is empty and the subtree is no longer needed (returned 2 - line 17) the key is deleted.

When the algorithm finishes with the current key (lines 3 - 23), the next key of the node is visited until all keys of the current node have been checked. We do need to visit all the keys of the node even if `gist_delete_req` has been called for one of them, because the node MBRs might overlap. This means that even if the MBR of the key we want to delete is inside one of the MBR of the keys of the node (line 3), the subtree of this key might not have the key we want to delete.

If the node's key is a leaf node (line 24) and the node's key matches exactly the search key and refers to the same data (line 25), then `gist_delete_key` is called to delete the key (line 26). If the page is now empty 2 is returned, if it is not empty 0 is returned and if something went wrong during the deletion −1 is returned.

gist_delete_key This method deletes a key from a node. An algorithm for this method is not presented because the actions it performs are extremely simple: a node is given and a specific key is deleted from the node. The deletion of a key is much simpler than the method `gist_add_key` (Section 5.3.3) that needs to perform a series of operations and checks.

5.3.2.3 Comparison with original GiST deletion

The deletion algorithm closely follows the original GiST. The major difference with the original algorithm is that we haven't implemented the full abstraction that GiST provides. Even if it is currently not fully implemented, when we do implement them, the flow of the algorithm will not change. The changes will have to be performed to the lower level methods

- `gist_key_cmp`
- `gist_delete_key`
- `gist_set_key_mbr`

that currently use the existing R-tree functionality. The same applies for the methods that are responsible for compacting the nodes (called by `gist_delete_key`). All these will simply call the appropriate `Compare` and `Union` methods for the specific variant the GiST index abstracts.

5.3.3 GiST insertion

In this section, we describe the algorithms of GiST keys for the insertion of data. The reader will notice a similarity between the deletion algorithms in this section and the existing R-tree MySQL indexes (Section 4.5.1).

```
Input: info, keyinfo, key, key_length, page, page_size, ReinsertList,
       level
Output: Modifies one level in the GiST: −1 for Error, 0 if key was
        deleted, 1 if key was not found, 2 if the leaf is empty
foreach key $k \in$ node do                      /* loop the keys of the node */
    if node is internal then
        if key within k then                          /* gist_key_cmp  */
            child ← child page of k;
            res ← gist_delete_req (info, keyinfo, key, key_length,
            child, page_size, ReinsertList, level + 1);
            if res == 0 then
                if page is adequatly filled then
                    gist_set_key_mbr (k);              /* store key MBR */
                else
                    add k's child to ReinsertList;
                    gist_delete_key (k) ;
                if error during the above then
                    return −1
                return res
            else if res == 1 then                     /* key not found */
                continue the loop and check other keys;
            else if res == 2 then                 /* last key in leaf page */
                gist_delete_key;
                if any error during the above then
                    return −1;
                return 0;
            else
                return −1;
    else                                                  /* Leaf node */
        if key MBR is equal to k and refers to the same data then
        /* gist_key_cmp  */
            gist_delete_key;
            if page is now empty then
                return 2;
            else
                return 0;
            if any error during the above then
                return −1;
return 1;
```

Algorithm 5.3.11: `gist_delete_req`: MyISAM GiST deletion. Called recursively on each level of the tree.

First, in Section 5.3.3.1, the algorithm is presented from a high level view and then in Section 5.3.3.2 it is described with more details. Finally, in Section 5.3.3.3, the differences with the original GiST algorithm are discussed.

5.3.3.1 Abstract description

The code that is associated with the GiST insertion is found in the source files `storage/myisam/gist_*`. A high level view of the insertion flow is presented in Algorithms 5.3.12 and 5.3.13 and the most important methods are:

- `gist_insert_level`
- `gist_insert_req`
- `gist_add_key`

The method `gist_insert_level` (Algorithm 5.3.12) is called from the root of the tree and in turn calls `gist_insert_req`. When `gist_insert_req` returns, the new key has been added, at the leaf level, and all the nodes below the root have been adjusted. Then the root node itself is adjusted and the insertion has finished.

The method `gist_insert_req` (Algorithm 5.3.13) is called recursively and descends the tree towards the leaf nodes. If an internal node is encountered then `gist_insert_req` is called (line 3) to descend down one level and takes as arguments the child node and the increased level. When it returns, the current level is adjusted and it is split, if required. When a leaf node is encountered the key is added (line 8) and if necessary the node is split.

```
  Input: key
1 begin
2     gist_insert_req (key, 0);
3     Adjust root if needed;
4 end
```

Algorithm 5.3.12: `gist_insert_level` abstract: MyISAM GiST insertion abstract.

```
Input: key, level
1  begin
2      if can go one level down then
3          gist_insert_req (key, level + 1);
4          Adjust key if child node was modified;
5          Split node if necessary;
6          return
7      else
8          gist_add_key;
9          return
10 end
```

Algorithm 5.3.13: `gist_insert_req` abstract: MyISAM GiST insertion abstract.

5.3.3.2 Detailed description

This section describes the details of GiST insertion. More specifically the following methods are presented:

- `gist_insert` (Algorithm 5.3.14)
- `gist_insert_level` (Algorithm 5.3.15)
- `gist_insert_req` (Algorithm 5.3.16)
- `gist_add_key` (Algorithm 5.3.17)

Even if we do provide enough details to understand how insertions are performed, some details fall outside the scope of the description. The description focuses on the fact that somehow, the key information can be read, updated and saved, and that nodes can be read and saved permanently, but doesn't mention how this is performed. These are important but lower level MyISAM operations and the interested reader can check directly in the source code files.

gist_insert The method is described in Algorithm 5.3.14 and is the single point of entry for the insertion of keys in MySQL's GiSTs. It modifies the index by inserting one key and returns 0 for success and 1 if something went wrong. It is a wrapper around `gist_insert_level` (line 1, described in Algorithm 5.3.15). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.

2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. key: the new leaf key that will be inserted in the tree

4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

```
Input: $info$, $keynr$, $key$, $key\_length$
Output: Modifies GiST: 1 for Error, 0 for OK
1 $res \leftarrow$ gist_insert_level ($info$, $keynr$, $key$, $key\_length$, $-1$);
2 return $res$;
```

Algorithm 5.3.14: gist_insert: MyISAM GiST insertion.

gist_insert_level The method is described in Algorithm 5.3.15. It modifies the index by calling `gist_insert_req` to insert the key. Returns 0 if the root was not split, 1 if it was split and −1 if something went wrong. It is called either during insertion by `gist_insert` (Algorithm 5.3.14) or during deletion at the re-insertion stage (Section 5.3.2.2, Algorithm 5.3.11). The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.

2. keynr: the number of index that is being used. In each table, each index has a number that identifies it.

3. key: the new leaf key that will be inserted in the tree

4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.

5. ins_level: the level at which the key is going to be insert. To insert a leaf node (like from an SQL Insert command) −1 is used. To insert a key during delete reinsertion (Section 5.3.2.2, Algorithm 5.3.11) the level of the key is used.

First, from the info data structure the root of the tree and information regarding the table's keys are read. Afterwards, an empty new node is created in memory, because it might be required further down the algorithm. Then the existence of the root node is tested (line 4). If the root node doesn't exist it's created and the key is added to the empty root. If the root does exist, `gist_insert_req` is called (line 11). This method recursively calls itself, in order to insert the key to the leaf node and adjust all the associated internal nodes. It returns with either an error or success. If the root was split during the process, a new root is created and keys are added there.

```
Input: info, keynr, key, key_length, ins_level
Output: Modifies GiST: $-1$ for Error, 0 if root was not split, 1 if root
        was split

1  keyinfo ← take key information from info;
2  new_page ← new empty node;
3  old_root ← take root node from info;

4  if Root doesn't exist then
5      Create new root;
6      if error during new root creation then
7          return $-1$;
8      else
9          res ← gist_add_key;          /* add key to the empty node */
10         return res;

11 res ← gist_insert_req (info, keyinfo, key, key_length, old_root,
   new_page , ins_level, 0);

12 if $res == 0$ then                          /* Root was not split */
13     return 0
14 else if $res == 1$ then                         /* Root was split */
15     Create new root and add keys there;
16     if error during new root creation then
17         return $-1$
18     return 1
19 else
20     return $-1$
```

Algorithm 5.3.15: `gist_insert_level`: MyISAM GiST insertion. Called from the root of the tree.

gist_insert_req The method, described in in Algorithm 5.3.16, is called recursively and in each recursion it modifies one level of the tree. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keyinfo: data structure that includes information about the key associated with the insertion.
3. key: is the new leaf key that will be inserted in the tree
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. new_page: an new empty node in memory. It is a place holder to insert new keys if needed.
6. ins_level: the level at which the key is going to be insert. To insert a leaf node (like from an SQL Insert command) -1 is used. To insert a key during delete reinsertion (Section 5.3.2.2, Algorithm 5.3.11) the level of the key is used.
7. level: the current level of the tree. When `gist_insert_req` descends one level down then this argument is increased by one.

Initially, the algorithm decides if the recursion should go one level down towards the leaf nodes (line 1). In case `gist_insert_req` was called by `gist_insert_level` to insert a new key in the tree, then the recursion continues until the leaf nodes are reached. In case the `gist_insert_req` was called by `gist_delete`, during the deletion of a key to re-insert a node that became filled less than the fill factor, the recursion continues until the level of the re-inserted node is reached.

If the algorithm must go one level down (line 1), then one key is picked up from the available keys of the node (line 2). The child of this key is the node where the algorithm will descend into (line 4). Then `gist_insert_req` is called for this key. Once it returns, the key has been added somewhere below and all the nodes below the current level have been adjusted. If the child node was not split (line 5), then the current node is adjusted. If the child was split, (line 11), then a new key points to the new child node. Afterwards, the new key and the old key are adjusted, the new key is added to the node (line 14) and the method returns the result of `gist_add_key` or -1 if something went wrong.

If the algorithm decides not go down one level (line 21), then the key is added to the node (line 22) and the method returns the result of `gist_add_key`. Ir returns -1 if something went wrong.

```
Input: info, keyinfo, key, key_length, page, new_page, ins_level, level
Output: Modifies one level in the GiST: −1 for Error, 0 if child was not
        split, 1 if child was split
if go down one level then
    k ← gist_pick_key                          /* will insert into entry k */
    p ← node where k points to (internal node or data);
    res ← gist_insert_req (info, keyinfo, key, key_length, p,
    new_page , ins_level, level + 1);
    if res == 0 then                                 /* Child was not split */
        gist_combine_rect (k, key);              /* add key MBR to k MBR */
        save node;
        if error then
            return −1
        return 0;
    else if res == 1 then                                /* Child was split */
        new_key ← new child node;
        /* calculate & store new and existing key MBRs                     */
        gist_set_key_mbr (k); gist_set_key_mbr (new_key);
        /* add new key to current node                                      */
        res ← gist_add_key (new_key);
        save current node;
        if error during the above then
            return −1
        return res
    else
        return −1
else    /* Node is leaf or we don't have to go further down */
    res ← gist_add_key (key) ;
    save node;
    if error during write then
        return −1 ;
    else
        return res;
```

Algorithm 5.3.16: `gist_insert_req`: MyISAM GiST insertion. Called recurcively on each level of the tree.

gist_add_key This method is responsible for adding a key to a node and it is presented in Algorithm 5.3.17. The input arguments of this method are the following:

1. info: data structure that includes information about the database table associated with the insertion.
2. keyinfo: data structure that includes information about the key associated with the insertion.
3. key: is the new leaf key that will be inserted in the tree
4. key_length: the key length. Keys can have different lengths because they can be of columns of data types with different size.
5. new_page: a new empty node.

If the node has enough free space for one additional key, then the key is added (line 1). If the node is a leaf then the key points to the data stored. If the node is internal then the key points to a child node. The method returns 0 indicating that the node was not split.

If the node does not have enough space for one more key, then the node is split and the new node is written in new_page (line 7). The method returns −1 on error or 1 on success indicating that the node was split.

5.3.3.3 Comparison with original GiST insertion

The insertion algorithm closely follows the original GiST. The major difference is that we haven't implemented the full abstraction that GiST provides. Even if it is currently not fully implemented, when we do implement them, the flow of the algorithm will not change. The changes will have to be performed to the lower level methods

- `gist_key_cmp`
- `gist_add_key`
- `gist_set_key_mbr`

that currently use the existing R-tree. The same applies for the method that is responsible for splitting the nodes (called by `gist_add_key`). They will all simply call the appropriate `Compare` and `Union` methods for the specific variant the GiST index abstracts.

```
Input: info, keyinfo, key, key_length, new_page
Output: Modifies key node: −1 for Error, 0 for no split, 1 for split

1  if node has enough free space to hold one more key then
       /* modify key's pointer */
2      if node is not leaf then
3          add the child node link to the key;
4      else
5          add the data record link to the key;
6      return 0;
7  res ← gist_split_page;
8  if res == 1 then
9      return −1;
10 else
11     return 1;
```

Algorithm 5.3.17: `gist_add_key`: MyISAM GiST insertion. Add key to node

5.4 Evaluation

In the previous sections we described the technical and algorithmic details of the GiST implementation. In this section we will perform and evaluation of the work, as far as the initial goals are concerned, as well as further work that should be done at the implementation.

We managed to implement the algorithms as close as possible to the original GiST algorithms, provide a solid mechanism for abstracting search trees and to hook the existing R*-tree methods to it. As we already noted in Sections 5.3.3.3, 5.3.2.3 and 5.3.1.3 the algorithms don't abstract the tree as much as the original GiST algorithms can.

To sum up, the methods:

- `gist_key_cmp`
- `gist_add_key`
- `gist_delete_key`
- `gist_set_key_mbr`

are currently using are the existing R-tree functionality and are missing the usage of the GiST methods `Union` and `Compare`. However, the current implementation allows for the future addition of `Union` and `Compare` without change the flow of the insert, delete and search algorithms.

5.5 Testing the GiST implementation

As we have already discussed in Section 1.3.1 MySQL is an RDBMS widely used in production in heavy workload and large infrastructures. Such a product wouldn't be complete without a good testing framework. Indeed, MySQL provides an extensive testing framework [65].

Even if testing "Testing shows the presence, not the absence of bugs" [70, p. 16] it is a valuable tool. It can make sure that the already test vectors that verify the correct behavior of a program hasn't been disrupted. After we implemented the changes in the codebase, we run three types of tests:

- A general health check: we run the generic test after building the patched MySQL with `make test`. All tests were successful. This means that our implementation didn't break something in the core server.
- GIS-specific tests: the testing suite includes GIS functionality and R-tree specific tests:
  - `gis-precise.test`
  - `gis-rt-precise.test`
  - `gis-rtree.test`
  - `gis.test`

  They all were run and were successful. This means that our implementation didn't break the existing GIS and R-tree functionality.
- A GiST-specific test: We duplicated the `gis-rtree.test` and we changed it so that all the indexes created and operated upon are GiST instead of R-tree. All tests were successful, which means that our GiST index replicates the existing R-tree functionality.

5.6 Summary

This chapter presented our GiST in MySQL's MyISAM storage engine. The implementation is split in two parts The first makes MySQL aware of the presence of the new index type. The second one is the implementation of the the index functionality. The changes were described in two ways. We first examined the modifications in the codebase per source code file, and then the algorithms of the GiST indexes were analyzed. Finally we summed up the implementation presentation and discussed how we use MySQL's existing testing suite to make sure our changes work well.

Chapter 6

Conclusion

The goal of the research was to conduct a thorough study of the existing spatial indexing solutions and search tree abstraction data models, and to implement a working example in the RDBMS MySQL. Despite the fact that there are still details to be explored and implemented, the general goals set initially for this project have been completed successfully.

We begun by explaining how the original spatial index R-tree and the abstract search tree GiST work. We analyzed the basic properties of each indexing solution and we then described them in detail. We presented all the algorithms in a detailed and code-like way, so that they are as close as possible to implementing them.

In the next chapter we changed our focus to spatial indexing solutions, and more specifically variants of the R-tree. We examined six variants the R^+-tree, the R^*-tree, the Hilbert R-tree, two splitting algorithms, and finally the VoR-Tree. The all shared the basic properties of R-trees. For some of them all the search, delete and insert functionality was presented and for others we examined their special features.

We then switched to the implementation part of the research. We presented high level views of the MySQL server, the interaction with the storage engines and some details on the MyISAM storage engine. We the thoroughly investigated the way the R*-tree works in MyISAM.

After having understood the way indexes and R*-tree is implemented in MyISAM we extended the SQL the server can parse and then implemented our own GiST indexing solution. Under GiST trees we plugged the already existing R*-tree spatial index, in a way that future R-tree-like indexes can be implemented.

6.1 Further work

The main points that would require further investigation in order to complete the current state of the project are implementing the full abstraction of GiST trees as well as implementing more spatial indexing solutions under GiSTs.

Understanding the way MySQL uses indexes was a procedure with a steep learning curve, but we managed to deliver a working new index tree. In order to fully take advantage of GiSTs and the effort we made to understand MySQL and MyISAM internals, the future work should focus on abstracting the way the nodes are handled. As we already discussed in Section 5.4 the methods:

- `gist_key_cmp`
- `gist_add_key`
- `gist_delete_key`
- `gist_set_key_mbr`

currently use the existing R-tree functionality. They should be altered so that they are using `Union` and `Compare`. This addition requires to analyze in detail the code that handles the nodes and performs actions like:

- finding the position of the next key in the node
- finding the position end of the node
- finding the length of the key

The above mentioned changes will not require modifications in the flow our GiST implementation algorithms.

The next step would be to implement new indexes under our GiST implementation. All the R-tree variants discussed in Section 3 are possible candidates. VoR-Tree would be interesting to implement since it extends the leaf data structure, and we could also perform benchmark and test of the various splitting methods that were discussed in Sections 3.4 and 3.5 as well as in the R*-tree paper (Section 3.2).

Appendix A

Compiling and running MariaDB

In this Section we present the procedure we followed to download and compile the source code of MariaDB. Extensive instructions for different types of operating systems and architectures are given in [45, 43]. However, for the sake of completeness and the ability to reproduce the whole procedure we do present all the required steps to build the MariaDB server and clients from scratch.

In Figure A.1 we present the operating system commands needed to install the requires software packages. The Debian's `apt` package handling utility facilitates the procedure.

- `bzr`: is the version control system used by MariaDB
- `build-dep mysql-server`: installs all the dependencies required to build MySQL (as well as MariaDB)
- `exuberant-ctags`: this optional software annotates C and C++ code and makes source navigation with editors like `vi` and `emacs` very smooth.

In Figure A.2 we present the commands needed to download the source code from using the `bzr` version control system. The code repository is hosted on

```
# install necessary packages
$ apt-get install bzr build-dep mysql-server

# optional package for easy source code tagging
$ apt-get install exuberant-ctags
```

Figure A.1: Commands for the installation of packages needed in Ubuntu/Debian Linux systems.

```
# download the latest 'trunk'
$ bzr branch lp:maria <DIR>

# download the latest 5.5 branch source code
$ bzr branch lp:maria/5.5 <DIR>
```

Figure A.2: Commands for downloading the latest source code from launchpad.

`launchpad.net` [46]. The current versions of MariaDB are 5.5 and 5.3 which are on their own branches. The user can use `bzr` to download the latest code of each version.

If the source code is needed, without revision history or using `bzr`, a tarball of the code can be dowloaded from [44].

In Figure A.3 we present the command required to producde the annotation that editors can use. The annotation is save in a file called `TAGS`.

In Figure A.4 we present the commands required to build the source code. MariaDB does provide handy build scripts (in directory `BUILD/`). However, we wanted to have full control of the procedure and the ability to reproduce evey aspect of the compilation. So, we recreated from the compile scripts the commands required to build a version of MariaDB for linux for a 64-bit machine with debug support.

```
$ cd /path/to/the/source/code
$ ctags -e -R * # -e emacs format
```

Figure A.3: Commands for the creating the source tagging/browsing for vi and emacs.

The `make install` command is optional. In Figure A.5 we show how the compiled MariaDB server and clients can be run without the time-consuming step of the installation.

If the reader does want to perform the installation step then `configure` option `--prefix=<PATH>/compiled` can be used so that the `make install` installs everything under a specific directory, thus avoiding ovewriting of the currently installed version of MariaDB or MySQL.

The `make install` and the post installation commands are required to run once, in roder to create the directories where the data are saved, and the database `mysql` which holds the credentials for the database users. The initial root user password is empty and not required for loggin in the server.

In Figure A.5 we present the commands required to start the server, stop the server, check the status of the server, and the run a client that connects to this server. The commands require that the `make install` and the post installation commands (of Figure A.4) were executed once. The reader might notice that the sample commands for stoping the server, checking the status of the server and running the client, don't use the compiled clients and utilities but the system-wide programs. This is possible since MariaDB is both binary compatible with MySQL and uses the same network protocol.

In Figure A.6 we present a minimal MySQL `my.cnf` configuration file. All the other configuration options get their default values. The custom values are:

- `port`: a different port (3340) from the MySQL's default (3306) is used to make sure that there is no clash between an already installed MySQL or MariaDB and that the client will connect to the proper server.
- `data`: this is a path to the data directory of MySQL. This is were the files containing the database and table data are saved.

```
$ cd /path/to/version/5.5/source/code

# prepare makefiles and build infrastructure, run once (for 5.5 branch)
$ cmake .

# creates the ./configure script (optional)
$ bash BUILD/autorun.sh

# setup environment for GCC compilation (optional)
$ CC="gcc" \
 CFLAGS="-Wall -Wextra -Wunused -Wwrite-strings -DUNIV_MUST_NOT_INLINE \
   -DEXTRA_DEBUG -DFORCE_INIT_OF_VARS  -DSAFEMALLOC -DPEDANTIC_SAFEMALLOC \
   -O0 -g3 -gdwarf-2  " \
 CXX="g++" \
 CXXFLAGS="-Wall -Wextra -Wunused -Wwrite-strings -Wno-unused-parameter \
    -Wnon-virtual-dtor -felide-constructors -fno-exceptions -fno-rtti \
    -DUNIV_MUST_NOT_INLINE -DEXTRA_DEBUG -DFORCE_INIT_OF_VARS  -DSAFEMALLOC \
    -DPEDANTIC_SAFEMALLOC -O0 -g3 -gdwarf-2  " \
 CXXLDFLAGS=""

# configure (optional)
# option '--with-gist-index' requires that the code is patched
$ ./configure \
 --prefix=<PATH> \
 --enable-assembler  \
 --enable-thread-safe-client  \
 --with-big-tables \
 --with-plugin-aria \
 --with-aria-tmp-tables \
 --without-plugin-innodb_plugin \
 --with-mysqld-ldflags=-static \
 --with-client-ldflags=-static \
 --with-readline  \
 --with-debug=full \
 --with-ssl \
 --with-plugins=max \
 --with-libevent \
 --enable-local-infile

# build
$ make

# installation (optional)
$ make install

# post installation commands (optional, run once)
$ cd <PATH>/compiled
$ ./bin/mysql_install_db \
 --basedir=<PATH> \
 --datadir=<PATH>/data \
 --skip-name-resolve \
 --force
```

Figure A.4: Commands for compiling the MariaDB source code.

```
$ cd /path/to/the/source/code

# start the server
$ ./sql/mysqld --defaults-file=/home/vag/projects/mariadb/compiled/my.cnf

# start the server with a debug trace file
$ ./sql/mysqld --defaults-file=/home/vag/projects/mariadb/compiled/my.cnf
 --debug=d,info,error,query,general,where:0,/home/vag/mysql.trace:f,mi_create &

# check the status of the server
$ mysqladmin -uroot --port=3340 --host=127.0.0.1 ping

# stop the server
$ mysqladmin -uroot --port=3340 --host=127.0.0.1 shutdown

# start a client
$ mysql -uroot --port=3340 --host=127.0.0.1
```

Figure A.5: Commands for running the MariaDB server and clients.

```
[mysqld]
port=3340

data=<PATH>/data
language=<PATH>/share/
```

Figure A.6: Sample configuration file for running MariaDB server.

Appendix B

Patches for the MariaDB codebase

In this chapter we present the changes we performed in the MariaDB codebase for the implementation of GiSTs. In Section B.1 we present the changes required to make MariaDB GiST-aware and in Section B.2 we present the changes required for the core GiST implementation.

The changes are presented in diff format. The numbers on the left are line numbers of the patch file. The syntax highlighting is as follows:

- Gray background is used for the beginning of individual file diffs.

```
1 === path of the file that the diff applies to
```

- Dark gray letters are used for diff information regarding the chunk's line position and file properties.

```
1 --- client/mysql.cc      2012-08-09 15:22:00 +0000
2 +++ client/mysql.cc      2012-08-18 05:37:44 +0000
3 @@ -670,6 +670,8 @@
```

- Black letters are used for the lines of code that we added.

```
1 + line of code added
```

- Light gray letters are used for the code that is present in the diff, but wasn't changed.

```
 line of code already existing
```

B.1 Make MariaDB GiST-aware

```
=== modified file 'client/mysql.cc'
--- client/mysql.cc     2012-08-09 15:22:00 +0000
+++ client/mysql.cc     2012-08-18 05:37:44 +0000
@@ -670,6 +670,8 @@
   { "ROWS", 0, 0, 0, ""},
   { "ROW_FORMAT", 0, 0, 0, ""},
   { "RTREE", 0, 0, 0, ""},
+  { "GIST_RSTAR", 0, 0, 0, ""},
+  { "GIST_RGUT83", 0, 0, 0, ""},
   { "SAVEPOINT", 0, 0, 0, ""},
   { "SCHEMA", 0, 0, 0, ""},
   { "SCHEMAS", 0, 0, 0, ""},

=== modified file 'config.h.cmake'
--- config.h.cmake      2012-07-31 17:29:07 +0000
+++ config.h.cmake      2012-08-18 05:37:44 +0000
@@ -588,6 +588,7 @@
 */
 #define HAVE_SPATIAL 1
 #define HAVE_RTREE_KEYS 1
+#define HAVE_GIST_KEYS 1
 #define HAVE_QUERY_CACHE 1
 #define BIG_TABLES 1


=== modified file 'include/maria.h'
--- include/maria.h     2012-05-04 05:16:38 +0000
+++ include/maria.h     2012-08-18 05:37:44 +0000
@@ -177,7 +177,7 @@
   uint16 keysegs;                     /* Number of key-segment */
   uint16 flag;                        /* NOSAME, PACK_USED */

-  uint8 key_alg;                      /* BTREE, RTREE */
+  uint8 key_alg;                      /* BTREE, RTREE, GIST */
   uint8 key_nr;                              /* key number (auto) */
   uint16 block_length;                /* Length of keyblock (auto) */
   uint16 underflow_block_length;      /* When to execute underflow */

=== modified file 'include/my_base.h'
--- include/my_base.h   2012-05-21 18:54:41 +0000
+++ include/my_base.h   2012-08-18 05:37:44 +0000
@@ -91,7 +91,9 @@
   HA_KEY_ALG_BTREE=    1,             /* B-tree, default one          */
   HA_KEY_ALG_RTREE=    2,             /* R-tree, for spatial searches */
   HA_KEY_ALG_HASH=     3,             /* HASH keys (HEAP tables) */
-  HA_KEY_ALG_FULLTEXT= 4              /* FULLTEXT (MyISAM tables) */
+  HA_KEY_ALG_FULLTEXT= 4,             /* FULLTEXT (MyISAM tables) */
+  HA_KEY_ALG_GIST_RSTAR=    5,        /* GiST R-start algorithm */
+  HA_KEY_ALG_GIST_RGUT83=   6,        /* GiST R-tree Gutman's original
    algorithm */
 };
```

```

         /* Storage media types */
@@ -253,12 +255,13 @@
 #define HA_NULL_ARE_EQUAL       2048    /* NULL in key are cmp as equal */
 #define HA_GENERATED_KEY        8192    /* Automaticly generated key */
 #define HA_RTREE_INDEX          16384   /* For RTREE search */
+#define HA_GIST_INDEX           4096    /* For GIST search */

         /* The combination of the above can be used for key type comparison.
               */
 #define HA_KEYFLAG_MASK (HA_NOSAME | HA_PACK_KEY | HA_AUTO_KEY | \
                          HA_BINARY_PACK_KEY | HA_FULLTEXT | HA_UNIQUE_CHECK
                              | \
                          HA_SPATIAL | HA_NULL_ARE_EQUAL | HA_GENERATED_KEY |
                               \
-                         HA_RTREE_INDEX)
+                         HA_RTREE_INDEX | HA_GIST_INDEX )

 /*
   Key contains partial segments.

=== modified file 'include/myisam.h'
--- include/myisam.h    2012-03-27 23:04:46 +0000
+++ include/myisam.h    2012-08-18 05:37:44 +0000
@@ -163,7 +163,7 @@
   uint16 keysegs;                      /* Number of key-segment */
   uint16 flag;                         /* NOSAME, PACK_USED */

-  uint8  key_alg;                      /* BTREE, RTREE */
+  uint8  key_alg;                      /* BTREE, RTREE, GIST */
   uint16 block_length;                 /* Length of keyblock (auto) */
   uint16 underflow_block_length;       /* When to execute underflow */
   uint16 keylength;                    /* Tot length of keyparts (auto) */

=== modified file 'sql/handler.h'
--- sql/handler.h       2012-07-16 07:48:03 +0000
+++ sql/handler.h       2012-08-18 05:37:44 +0000
@@ -187,6 +187,7 @@
   engine.
 */
 #define HA_MUST_USE_TABLE_CONDITION_PUSHDOWN (LL(1) << 42)
+#define HA_CAN_GISTKEYS         (LL(1) << 43)

 /*
   Set of all binlog flags. Currently only contain the capabilities

=== modified file 'sql/lex.h'
--- sql/lex.h   2012-03-11 22:45:18 +0000
+++ sql/lex.h   2012-08-18 05:37:44 +0000
@@ -24,6 +24,7 @@
 SYM_GROUP sym_group_common= {"", ""};
 SYM_GROUP sym_group_geom= {"Spatial extentions", "HAVE_SPATIAL"};
 SYM_GROUP sym_group_rtree= {"RTree keys", "HAVE_RTREE_KEYS"};
+/*SYM_GROUP sym_group_dummy= {"Dummy keys", "HAVE_DUMMY_KEYS"};*/

 /* We don't want to include sql_yacc.h into gen_lex_hash */
 #ifdef NO_YACC_SYMBOLS
@@ -245,6 +246,8 @@
   { "GEOMETRY",                 SYM(GEOMETRY_SYM)},
   { "GEOMETRYCOLLECTION",SYM(GEOMETRYCOLLECTION)},
   { "GET_FORMAT",       SYM(GET_FORMAT)},
+  { "GIST_RSTAR",       SYM(GIST_RSTAR_SYM)},
+  { "GIST_RGUT83",      SYM(GIST_RGUT83_SYM)},
   { "GLOBAL",          SYM(GLOBAL_SYM)},
   { "GRANT",           SYM(GRANT)},
   { "GRANTS",          SYM(GRANTS)},

=== modified file 'sql/mysqld.cc'
--- sql/mysqld.cc       2012-08-09 15:22:00 +0000
+++ sql/mysqld.cc       2012-08-18 05:37:44 +0000
@@ -633,7 +633,7 @@
 MY_LOCALE *my_default_lc_time_names;

 SHOW_COMP_OPTION have_ssl, have_symlink, have_dlopen, have_query_cache;
-SHOW_COMP_OPTION have_geometry, have_rtree_keys;
+SHOW_COMP_OPTION have_geometry, have_rtree_keys, have_gist_keys;
 SHOW_COMP_OPTION have_crypt, have_compress;
 SHOW_COMP_OPTION have_profiling;

@@ -7346,6 +7346,11 @@
 #else
   have_rtree_keys=SHOW_OPTION_NO;
 #endif
+#ifdef HAVE_GIST_KEYS
+  have_gist_keys=SHOW_OPTION_YES;
+#else
+  have_gist_keys=SHOW_OPTION_NO;
+#endif
 #ifdef HAVE_CRYPT
   have_crypt=SHOW_OPTION_YES;
 #else

=== modified file 'sql/set_var.h'
--- sql/set_var.h       2012-03-27 23:04:46 +0000
+++ sql/set_var.h       2012-08-18 05:37:44 +0000
@@ -293,7 +293,7 @@

 extern SHOW_COMP_OPTION have_ssl, have_symlink, have_dlopen;
 extern SHOW_COMP_OPTION have_query_cache;
-extern SHOW_COMP_OPTION have_geometry, have_rtree_keys;
+extern SHOW_COMP_OPTION have_geometry, have_rtree_keys, have_gist_keys;
 extern SHOW_COMP_OPTION have_crypt;
 extern SHOW_COMP_OPTION have_compress;


=== modified file 'sql/sql_show.cc'
--- sql/sql_show.cc     2012-08-09 15:22:00 +0000
+++ sql/sql_show.cc     2012-08-18 05:37:44 +0000
@@ -1661,6 +1661,12 @@
         !(key_info->flags & HA_SPATIAL))
       packet->append(STRING_WITH_LEN(" USING RTREE"));

+    if (key_info->algorithm == HA_KEY_ALG_GIST_RSTAR )
+      packet->append(STRING_WITH_LEN(" USING GIST_RSTAR"));
+
+    if (key_info->algorithm == HA_KEY_ALG_GIST_RGUT83 )
+      packet->append(STRING_WITH_LEN(" USING GIST_RGUT83"));
+
     if ((key_info->flags & HA_USES_BLOCK_SIZE) &&
         table->s->key_block_size != key_info->block_size)
     {

=== modified file 'sql/sql_table.cc'
--- sql/sql_table.cc    2012-08-15 11:37:55 +0000
+++ sql/sql_table.cc    2012-08-18 05:37:44 +0000
```

```
@@ -3393,6 +3393,7 @@
     */

     /* TODO: Add proper checks if handler supports key_type and algorithm */
+                DBUG_PRINT("info", ("key_info->flags: %lu", key_info->flags))
     ;
     if (key_info->flags & HA_SPATIAL)
     {
       if (!(file->ha_table_flags() & HA_CAN_RTREEKEYS))

=== modified file 'sql/sql_yacc.yy'
--- sql/sql_yacc.yy     2012-08-09 15:22:00 +0000
+++ sql/sql_yacc.yy     2012-08-18 05:37:44 +0000
@@ -754,6 +754,7 @@
   enum enum_var_type var_type;
   Key::Keytype key_type;
   enum ha_key_alg key_alg;
+  enum ha_key_alg gist_key_alg;
   handlerton *db_type;
   enum row_type row_type;
   enum ha_rkey_function ha_rkey_mode;
@@ -1009,6 +1010,9 @@
 %token  GEOMETRYCOLLECTION
 %token  GEOMETRY_SYM
 %token  GET_FORMAT                    /* MYSQL-FUNC */
+%token  GIST_SYM                      /* GiST tree and algorithms*/
+%token  GIST_RSTAR_SYM
+%token  GIST_RGUT83_SYM
 %token  GLOBAL_SYM                    /* SQL-2003-R */
 %token  GRANT                         /* SQL-2003-R */
 %token  GRANTS
@@ -1533,6 +1537,10 @@
 %type <key_alg>
         btree_or_rtree

+%type <gist_key_alg>
+        gist_variant
+
+
 %type <string_list>
         using_list

@@ -2123,7 +2131,7 @@
             if (add_create_index_prepare(Lex, $7))
               MYSQL_YYABORT;
           }
-          '(' key_list ')' spatial_key_options
+          '(' key_list ')' gist_key_alg spatial_key_options
           {
             if (add_create_index(Lex, $2, $4))
               MYSQL_YYABORT;
@@ -5404,7 +5412,7 @@
         | spatial opt_key_or_index opt_ident init_key_options
             '(' key_list ')'
           { Lex->option_list= NULL; }
-            spatial_key_options
+            gist_key_alg spatial_key_options
           {
            if (add_create_index (Lex, $1, $3))
              MYSQL_YYABORT;
@@ -6271,6 +6279,11 @@
         | init_key_options key_using_alg
         ;

+gist_key_alg:
+          /* empty */ {}
+        | USING gist_variant     { Lex->key_create_info.algorithm= $2; }
+        ;
+
 normal_key_options:
          /* empty */ {}
        | normal_key_opts
@@ -6364,6 +6377,11 @@
        | HASH_SYM  { $$= HA_KEY_ALG_HASH; }
        ;

+gist_variant:
+          GIST_RSTAR_SYM  { $$= HA_KEY_ALG_GIST_RSTAR; }
+        | GIST_RGUT83_SYM { $$= HA_KEY_ALG_GIST_RGUT83; }
+        ;
+
 key_list:
          key_list ',' key_part order_dir { Lex->col_list.push_back($3); }
        | key_part order_dir { Lex->col_list.push_back($1); }
@@ -13069,6 +13087,9 @@
        | GEOMETRY_SYM              {}
        | GEOMETRYCOLLECTION        {}
        | GET_FORMAT                {}
+        | GIST_SYM                  {}
+        | GIST_RSTAR_SYM            {}
+        | GIST_RGUT83_SYM           {}
        | GRANTS                    {}
        | GLOBAL_SYM                {}
        | HASH_SYM                  {}

=== modified file 'sql/sys_vars.cc'
--- sql/sys_vars.cc     2012-08-14 10:40:40 +0000
+++ sql/sys_vars.cc     2012-08-18 05:37:44 +0000
@@ -3114,6 +3114,10 @@
        "have_rtree_keys", "have_rtree_keys",
        READ_ONLY GLOBAL_VAR(have_rtree_keys), NO_CMD_LINE);

+static Sys_var_have Sys_have_gist_keys(
+       "have_gist_keys", "have_gist_keys",
+       READ_ONLY GLOBAL_VAR(have_gist_keys), NO_CMD_LINE);
+
 static Sys_var_have Sys_have_ssl(
       "have_ssl", "have_ssl",
       READ_ONLY GLOBAL_VAR(have_ssl), NO_CMD_LINE);

=== modified file 'storage/myisam/CMakeLists.txt'
--- storage/myisam/CMakeLists.txt       2012-05-22 09:04:32 +0000
+++ storage/myisam/CMakeLists.txt       2012-08-18 05:37:44 +0000
@@ -25,7 +25,8 @@
                                mi_rsame.c mi_rsamepos.c mi_scan.c mi_search.
                                     c mi_static.c mi_statrec.c
                                mi_unique.c mi_update.c mi_write.c rt_index.c
                                      rt_key.c rt_mbr.c
                                rt_split.c sort.c sp_key.c mi_extrafunc.h
                                     myisamdef.h
-                               rt_index.h mi_rkey.c)
+                               rt_index.h mi_rkey.c
+                               gist_index.h gist_index.c)

 MYSQL_ADD_PLUGIN(myisam ${MYISAM_SOURCES}
   STORAGE_ENGINE

```

```
=== added file 'storage/myisam/gist_index.c'
--- storage/myisam/gist_index.c 1970-01-01 00:00:00 +0000
+++ storage/myisam/gist_index.c 2012-08-18 05:37:44 +0000
@@ -0,0 +1,219 @@
+/* Copyright (C) 2012 Monty Program AB & Vangelis Katsikaros
+
+   This program is free software; you can redistribute it and/or modify
+   it under the terms of the GNU General Public License as published by
+   the Free Software Foundation; version 2 of the License.
+
+   This program is distributed in the hope that it will be useful,
+   but WITHOUT ANY WARRANTY; without even the implied warranty of
+   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+   GNU General Public License for more details.
+
+   You should have received a copy of the GNU General Public License
+   along with this program; if not, write to the Free Software
+   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
      */
+
+#include "myisamdef.h"
+
+#ifdef HAVE_GIST_KEYS
+
+#include "gist_index.h"
+
+typedef struct st_page_level
+{
+  uint level;
+  my_off_t offs;
+} stPageLevel;
+
+typedef struct st_page_list
+{
+  ulong n_pages;
+  ulong m_pages;
+  stPageLevel *pages;
+} stPageList;
+
+
+
+
+/*
+  Find first key in gist-tree according to search_flag condition
+
+  SYNOPSIS
+   gist_find_first()
+   info                        Handler to MyISAM file
+   uint keynr          Key number to use
+   key                 Key to search for
+   key_length          Length of 'key'
+   search_flag         Bitmap of flags how to do the search
+
+  RETURN
+    -1  Error
+    0   Found
+    1   Not found
+*/
+
+int gist_find_first(MI_INFO *info, uint keynr, uchar *key, uint key_length,
+                    uint search_flag)
+{
+
+  my_off_t root;
+  //uint nod_cmp_flag;
+  //MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
+  DBUG_ENTER("gist_find_first"); // no DBUG were initially used
+  if ((root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
+  {
+    my_errno= HA_ERR_END_OF_FILE;
+    return -1;
+  }
+  DBUG_PRINT("gist", ("info: %lu  keynr: %u  key: %s key_length: %u
     search_flag: %u", (ulong) info, keynr, key, key_length, search_flag ) );
+  DBUG_RETURN(0); /* sceleton return */
+
+}
+
+
+/*
+   Find next key in gist-tree according to search_flag condition
+
+  SYNOPSIS
+   gist_find_next()
+   info                        Handler to MyISAM file
+   uint keynr           Key number to use
+   search_flag          Bitmap of flags how to do the search
+
+   RETURN
+      -1  Error
+      0   Found
+      1   Not found
+*/
+
+int gist_find_next(MI_INFO *info, uint keynr, uint search_flag)
+{
+  my_off_t root;
+  uint nod_cmp_flag;
+  MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
+
+  nod_cmp_flag = 0;
+  root = 0;
+  DBUG_PRINT("gist", ("info: %lu  keynr: %u  search_flag: %u", (ulong) info,
      keynr, search_flag ) );
+  DBUG_PRINT("gist", ("keyinfo: %lu  keynr: %u  search_flag: %lu", (ulong)
     keyinfo, nod_cmp_flag, (ulong) root ) );
+
+  if (info->update & HA_STATE_DELETED)
+    return gist_find_first(info, keynr, info->lastkey, info->lastkey_length,
+                           search_flag);
+
+    my_errno= HA_ERR_END_OF_FILE;
+    return -1;
+}
+
+
+
+/*
+  Get first key in gist-tree
+
+  RETURN
+    -1 Error
+    0  Found
+    1  Not found
+*/
+
+int gist_get_first(MI_INFO *info, uint keynr, uint key_length)
+{
+  my_off_t root;
+  MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
```

```
+
+  DBUG_PRINT("gist", ("nfo: %lu  keynr: %u  key_length: %u, keyinfo: %p", (
     ulong) info, keynr, key_length, keyinfo ) );
+
+
+  if ((root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
+  {
+    my_errno= HA_ERR_END_OF_FILE;
+    return -1;
+  }
+
+  return -1;
+}
+
+
+/*
+  Get next key in gist-tree
+
+  RETURN
+    -1 Error
+    0  Found
+    1  Not found
+*/
+
+int gist_get_next(MI_INFO *info, uint keynr, uint key_length)
+{
+  my_off_t root= info->s->state.key_root[keynr];
+  MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
+
+  DBUG_PRINT("gist", ("info: %lu  keynr: %u  key_length: %u, keyinfo: %p,
     root: %lu", (ulong) info, keynr, key_length, keyinfo, (ulong) root ) );
+
+  if (root == HA_OFFSET_ERROR)
+  {
+    my_errno= HA_ERR_END_OF_FILE;
+    return -1;
+  }
+
+  return -1;
+}
+
+
+
+
+/*
+  Insert key into the tree - interface function
+
+  RETURN
+    -1 Error
+    0  OK
+*/
+
+int gist_insert(MI_INFO *info, uint keynr, uchar *key, uint key_length)
+{
+  DBUG_ENTER("gist_insert");
+  /* DBUG_RETURN((!key_length || */
+  /*            (gist_insert_level(info, keynr, key, key_length, -1) ==
     -1)) ? */
+  /*              -1 : 0); */
+  DBUG_PRINT("gist", ("info: %lu  keynr: %u  key: %s key_length: %u", (ulong
     ) info, keynr, key, key_length ) );
+  DBUG_RETURN(-1); /* sceleton return */
+}
+
+
+
+
+/*
+  Delete key - interface function
+
+  RETURN
+    -1 Error
+    0  Deleted
+*/
+
+int gist_delete(MI_INFO *info, uint keynr, uchar *key, uint key_length)
+{
+  uint page_size;
+  stPageList ReinsertList;
+  my_off_t old_root;
+  MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
+  DBUG_ENTER("gist_delete");
+
+  if ((old_root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
+  {
+    my_errno= HA_ERR_END_OF_FILE;
+    DBUG_RETURN(-1); /* purecov: inspected */
+  }
+  DBUG_PRINT("rtree", ("starting deletion at root page: %lu",
+                       (ulong) old_root));
+
+  page_size = 0;
+  DBUG_PRINT("gist", ("info: %lu  keynr: %u  key: %s key_length: %u", (ulong
    ) info, keynr, key, key_length ) );
+  DBUG_PRINT("gist", ("page_size: %u  ReinsertList: %p keyinfo: %p ",
    page_size, &ReinsertList, keyinfo ) );
+  DBUG_RETURN(-1); /* sceleton return */
+}
+
+
+
+#endif /*HAVE_RTREE_KEYS*/
+

=== added file 'storage/myisam/gist_index.h'
--- storage/myisam/gist_index.h 1970-01-01 00:00:00 +0000
+++ storage/myisam/gist_index.h 2012-08-18 05:37:44 +0000
@@ -0,0 +1,39 @@
+/* Copyright (C) 2012 Monty Program AB & Vangelis Katsikaros
+
+   This program is free software; you can redistribute it and/or modify
+   it under the terms of the GNU General Public License as published by
+   the Free Software Foundation; version 2 of the License.
+
+   This program is distributed in the hope that it will be useful,
+   but WITHOUT ANY WARRANTY; without even the implied warranty of
+   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+   GNU General Public License for more details.
+
+   You should have received a copy of the GNU General Public License
+   along with this program; if not, write to the Free Software
+   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
     */
+
+#ifndef _gist_index_h
+#define _gist_index_h
+
+#ifdef HAVE_GIST_KEYS
+
+#define gist_PAGE_FIRST_KEY(page, nod_flag) (page + 2 + nod_flag)
+#define gist_PAGE_NEXT_KEY(key, key_length, nod_flag) (key + key_length + \
+               (nod_flag ? nod_flag : info->s->base.rec_reflength))
+#define gist_PAGE_END(page) (page + mi_getint(page))
+
+#define gist_PAGE_MIN_SIZE(block_length) ((uint)(block_length) / 3)
+
+int gist_insert(MI_INFO *info, uint keynr, uchar *key, uint key_length);
+int gist_delete(MI_INFO *info, uint keynr, uchar *key, uint key_length);
+
+int gist_find_first(MI_INFO *info, uint keynr, uchar *key, uint key_length,
+                     uint search_flag);
+int gist_find_next(MI_INFO *info, uint keynr, uint search_flag);
+
+int gist_get_first(MI_INFO *info, uint keynr, uint key_length);
+int gist_get_next(MI_INFO *info, uint keynr, uint key_length);
+
+#endif /*HAVE_GIST_KEYS*/
+#endif /* _gist_index_h */

=== added file 'storage/myisam/gist_key.c'
--- storage/myisam/gist_key.c   1970-01-01 00:00:00 +0000
+++ storage/myisam/gist_key.c   2012-08-18 05:37:44 +0000
@@ -0,0 +1,23 @@
+/* Copyright (C) 2012 Monty Program AB & Vangelis Katsikaros
+
+   This program is free software; you can redistribute it and/or modify
+   it under the terms of the GNU General Public License as published by
+   the Free Software Foundation; version 2 of the License.
+
+   This program is distributed in the hope that it will be useful,
+   but WITHOUT ANY WARRANTY; without even the implied warranty of
+   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+   GNU General Public License for more details.
+
+   You should have received a copy of the GNU General Public License
+   along with this program; if not, write to the Free Software
+   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
     */
+
+#include "myisamdef.h"
+
+#ifdef HAVE_GIST_KEYS
+#include "gist_index.h"
+#include "gist_key.h"
+
+
+#endif /*HAVE_GIST_KEYS*/

=== added file 'storage/myisam/gist_key.h'
--- storage/myisam/gist_key.h   1970-01-01 00:00:00 +0000
+++ storage/myisam/gist_key.h   2012-08-18 05:37:44 +0000
@@ -0,0 +1,23 @@
+/* Copyright (C) 2012 Monty Program AB & Vangelis Katsikaros
+
+   This program is free software; you can redistribute it and/or modify
+   it under the terms of the GNU General Public License as published by
+   the Free Software Foundation; version 2 of the License.
+
+   This program is distributed in the hope that it will be useful,
+   but WITHOUT ANY WARRANTY; without even the implied warranty of
+   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+   GNU General Public License for more details.
+
+   You should have received a copy of the GNU General Public License
+   along with this program; if not, write to the Free Software
+   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA */
+
+
+#ifndef _gist_key_h
+#define _gist_key_h
+
+#ifdef HAVE_GIST_KEYS
+
+#endif /*HAVE_GIST_KEYS*/
+#endif /* _gist_key_h */

=== modified file 'storage/myisam/ha_myisam.cc'
--- storage/myisam/ha_myisam.cc 2012-07-18 18:40:15 +0000
+++ storage/myisam/ha_myisam.cc 2012-08-18 05:37:44 +0000
@@ -242,6 +242,7 @@
     keydef[i].key_alg= pos->algorithm == HA_KEY_ALG_UNDEF ?
       (pos->flags & HA_SPATIAL ? HA_KEY_ALG_RTREE : HA_KEY_ALG_BTREE) :
       pos->algorithm;
+    DBUG_PRINT("debug", ("algorithm: %u, flag: %u", keydef[i].key_alg, keydef[i].flag ));
     keydef[i].block_length= pos->block_size;
     keydef[i].seg= keyseg;
     keydef[i].keysegs= pos->key_parts;
@@ -650,7 +651,7 @@
                   HA_CAN_VIRTUAL_COLUMNS |
                   HA_DUPLICATE_POS | HA_CAN_INDEX_BLOBS | HA_AUTO_PART_KEY |
                   HA_FILE_BASED | HA_CAN_GEOMETRY | HA_NO_TRANSACTIONS |
-                  HA_CAN_INSERT_DELAYED | HA_CAN_BIT_FIELD | HA_CAN_RTREEKEYS |
+                  HA_CAN_INSERT_DELAYED | HA_CAN_BIT_FIELD | HA_CAN_RTREEKEYS | HA_CAN_GISTKEYS |
                   HA_HAS_RECORDS | HA_STATS_RECORDS_IS_EXACT | HA_CAN_REPAIR),
    can_enable_indexes(1)
 {}
@@ -685,6 +686,10 @@
          "SPATIAL" :
          (table->key_info[key_number].algorithm == HA_KEY_ALG_RTREE) ?
          "RTREE" :
+         (table->key_info[key_number].algorithm == HA_KEY_ALG_GIST_RSTAR) ?
+         "GIST_RSTAR" :
+         (table->key_info[key_number].algorithm == HA_KEY_ALG_GIST_RGUT83) ?
+         "GIST_RGUT83" :
          "BTREE");
 }


=== modified file 'storage/myisam/mi_check.c'
--- storage/myisam/mi_check.c   2012-04-07 13:58:46 +0000
+++ storage/myisam/mi_check.c   2012-08-18 05:37:44 +0000
@@ -52,6 +52,7 @@
 #include <sys/mman.h>
 #endif
 #include "rt_index.h"
+#include "gist_index.h"

        /* Functions defined in this file */

@@ -1222,14 +1223,29 @@
              /* We don't need to lock the key tree here as we don't allow
                 concurrent threads when running myisamchk
              */
-              int search_result=
+              int search_result;
 #ifdef HAVE_RTREE_KEYS
-                (keyinfo->flag & HA_SPATIAL) ?
-                rtree_find_first(info, key, info->lastkey, key_length,
-                                 MBR_EQUAL | MBR_DATA) :
-#endif
+             if (keyinfo->flag & HA_SPATIAL)
+             {
+                search_result = rtree_find_first(info, key, info->lastkey,
+                                key_length, MBR_EQUAL | MBR_DATA);
+             }
+              else
+#endif
+#ifdef HAVE_GIST_KEYS
+             if (search_result && keyinfo->flag & HA_GIST_INDEX)
+             {
+                search_result = gist_find_first(info, key, info->lastkey,
+                                key_length, 0);
+             }
+              else
+#endif
+             if (search_result)
+             {
                 _mi_search(info,keyinfo,info->lastkey,key_length,
                            SEARCH_SAME, info->s->state.key_root[key]);
+             }
+
               if (search_result)
               {
                 mi_check_print_error(param,"Record at: %10s  "
@@ -1919,7 +1935,9 @@
   /* cannot sort index files with R-tree indexes */
   for (key= 0,keyinfo= &share->keyinfo[0]; key < share->base.keys ;
        key++,keyinfo++)
-    if (keyinfo->key_alg == HA_KEY_ALG_RTREE)
+    if (keyinfo->key_alg == HA_KEY_ALG_RTREE ||
+       keyinfo->key_alg == HA_KEY_ALG_GIST_RSTAR ||
+       keyinfo->key_alg == HA_KEY_ALG_GIST_RGUT83)
      DBUG_RETURN(0);

   if (!(param->testflag & T_SILENT))
@@ -2020,6 +2038,8 @@

   /* cannot walk over R-tree indices */
   DBUG_ASSERT(keyinfo->key_alg != HA_KEY_ALG_RTREE);
+  DBUG_ASSERT(keyinfo->key_alg != HA_KEY_ALG_GIST_RSTAR);
+  DBUG_ASSERT(keyinfo->key_alg != HA_KEY_ALG_GIST_RSTAR);
   new_page_pos=param->new_file_pos;
   param->new_file_pos+=keyinfo->block_length;


=== modified file 'storage/myisam/mi_create.c'
--- storage/myisam/mi_create.c  2012-03-06 19:46:07 +0000
+++ storage/myisam/mi_create.c  2012-08-18 05:37:44 +0000
@@ -254,9 +254,11 @@
     share.state.key_root[i]= HA_OFFSET_ERROR;
     min_key_length_skip=length=real_length_diff=0;
     key_length=pointer;
+    DBUG_PRINT("debug", ("keydef flag: %u", keydef->flag));
     if (keydef->flag & HA_SPATIAL)
     {
 #ifdef HAVE_SPATIAL
+
       /* BAR TODO to support 3D and more dimensions in the future */
       uint sp_segs=SPDIMS*2;
       keydef->flag=HA_SPATIAL;

=== modified file 'storage/myisam/mi_open.c'
--- storage/myisam/mi_open.c    2012-03-06 19:46:07 +0000
+++ storage/myisam/mi_open.c    2012-08-18 05:37:44 +0000
@@ -19,6 +19,7 @@
 #include "fulltext.h"
 #include "sp_defs.h"
 #include "rt_index.h"
+#include "gist_index.h"
 #include <m_ctype.h>
 #include <mysql_version.h>

@@ -71,7 +72,7 @@

 MI_INFO *mi_open(const char *name, int mode, uint open_flags)
 {
-  int lock_error,kfile,open_mode,save_errno,have_rtree=0, realpath_err;
+  int lock_error,kfile,open_mode,save_errno,have_rtree=0, have_gist=0,
     realpath_err;
   uint i,j,len,errpos,head_length,base_pos,offset,info_length,keys,
     key_parts,unique_key_parts,base_key_parts,fulltext_keys,uniques;
   char name_buff[FN_REFLEN], org_name[FN_REFLEN], index_name[FN_REFLEN],
@@ -322,6 +323,12 @@
                         end_pos);
          if (share->keyinfo[i].key_alg == HA_KEY_ALG_RTREE)
            have_rtree=1;
+         if (share->keyinfo[i].key_alg == HA_KEY_ALG_GIST_RSTAR ||
+             share->keyinfo[i].key_alg == HA_KEY_ALG_GIST_RGUT83)
+        {
+           have_gist=1;
+        }
+
        set_if_smaller(share->blocksize,share->keyinfo[i].block_length);
        share->keyinfo[i].seg=pos;
        for (j=0 ; j < share->keyinfo[i].keysegs; j++,pos++)
@@ -528,7 +535,7 @@
                            HA_OPTION_COMPRESS_RECORD |
                            HA_OPTION_TEMP_COMPRESS_RECORD)) ||
         (open_flags & HA_OPEN_TMP_TABLE) ||
-        have_rtree) ? 0 : 1;
+         have_rtree || have_gist) ? 0 : 1;
       if (share->concurrent_insert)
       {
        share->lock.get_status=mi_get_status;
@@ -560,6 +567,7 @@
       goto err;
     errpos=5;
     have_rtree= old_info->rtree_recursion_state != NULL;
+    have_gist= old_info->gist_recursion_state != NULL;
   }

   /* alloc and set up private structure parts */
@@ -572,6 +580,7 @@
                        &info.first_mbr_key, share->base.max_key_length,
                        &info.filename,strlen(name)+1,
                        &info.rtree_recursion_state,have_rtree ? 1024 : 0,
+                       &info.gist_recursion_state,have_gist ? 1024 : 0,
                        NullS))
     goto err;
   errpos=6;
```

```
@@ -579,6 +588,10 @@
   if (!have_rtree)
     info.rtree_recursion_state= NULL;

+  if (!have_gist)
+  {
+    info.gist_recursion_state= NULL;
+  }
   strmov(info.filename,name);
   memcpy(info.blobs,share->blobs,sizeof(MI_BLOB)*share->base.blobs);
   info.lastkey2=info.lastkey+share->base.max_key_length;
@@ -812,6 +825,17 @@
     DBUG_ASSERT(0); /* mi_open should check it never happens */
 #endif
   }
+  else if (keyinfo->key_alg == HA_KEY_ALG_GIST_RSTAR ||
+           keyinfo->key_alg == HA_KEY_ALG_GIST_RGUT83)
+  {
+#ifdef HAVE_GIST_KEYS
+    /* gist api will cal lthe proper key specific functionality */
+    keyinfo->ck_insert = gist_insert;
+    keyinfo->ck_delete = gist_delete;
+#else
+    DBUG_ASSERT(0); /* mi_open should check it never happens */
+#endif
+  }
   else
   {
     keyinfo->ck_insert = _mi_ck_write;
@@ -819,6 +843,7 @@
   }
   if (keyinfo->flag & HA_BINARY_PACK_KEY)
   {                                           /* Simple prefix compression
        */
+    DBUG_PRINT("info", ("HA_BINARY_PACK_KEY: bin_search -> _mi_seq_search"))
      ;
     keyinfo->bin_search=_mi_seq_search;
     keyinfo->get_key=_mi_get_binary_pack_key;
     keyinfo->pack_key=_mi_calc_bin_pack_key_length;
@@ -837,6 +862,7 @@
         cannot represent blank like ASCII does. In these cases we have
         to use _mi_seq_search() for the search.
       */
+      DBUG_PRINT("info", ("HA_VAR_LENGTH_KEY, HA_PACK_KEY: bin_search ->
      _mi_seq_search OR _mi_prefix_search"));
       if (!keyinfo->seg->charset || use_strnxfrm(keyinfo->seg->charset) ||
           (keyinfo->seg->flag & HA_NULL_PART) ||
           (keyinfo->seg->charset->mbminlen > 1))
@@ -848,6 +874,7 @@
     }
     else
     {
+      DBUG_PRINT("info", ("HA_VAR_LENGTH_KEY, no HA_PACK_KEY: bin_search ->
      _mi_seq_search"));
       keyinfo->bin_search=_mi_seq_search;
       keyinfo->pack_key=_mi_calc_var_key_length; /* Variable length key */
       keyinfo->store_key=_mi_store_static_key;
@@ -855,6 +882,7 @@
   }
   else
   {
+    DBUG_PRINT("info", ("other key flag: bin_search -> _mi_bin_search"));
     keyinfo->bin_search=_mi_bin_search;
     keyinfo->get_key=_mi_get_static_key;
     keyinfo->pack_key=_mi_calc_static_key_length;
```

```

=== modified file 'storage/myisam/mi_rkey.c'
--- storage/myisam/mi_rkey.c    2012-02-21 19:51:56 +0000
+++ storage/myisam/mi_rkey.c    2012-08-18 05:37:44 +0000
@@ -18,6 +18,7 @@

 #include "myisamdef.h"
 #include "rt_index.h"
+#include "gist_index.h"

        /* Read a record using key */
        /* Ordinary search_flag is 0 ; Give error if no record with key */
@@ -94,6 +95,30 @@
     }
     break;
 #endif
+#ifdef HAVE_GIST_KEYS
+  case HA_KEY_ALG_GIST_RSTAR:
+    if (gist_find_first(info,inx,key_buff,use_key_length,nextflag) < 0)
+    {
+      mi_print_error(info->s, HA_ERR_CRASHED);
+      my_errno=HA_ERR_CRASHED;
+      if (share->concurrent_insert)
+        rw_unlock(&share->key_root_lock[inx]);
+      fast_mi_writeinfo(info);
+      goto err;
+    }
+    break;
+  case HA_KEY_ALG_GIST_RGUT83:
+    if (gist_find_first(info,inx,key_buff,use_key_length,nextflag) < 0)
+    {
+      mi_print_error(info->s, HA_ERR_CRASHED);
+      my_errno=HA_ERR_CRASHED;
+      if (share->concurrent_insert)
+        rw_unlock(&share->key_root_lock[inx]);
+      fast_mi_writeinfo(info);
+      goto err;
+    }
+    break;
+#endif
   case HA_KEY_ALG_BTREE:
   default:
     if (!_mi_search(info, keyinfo, key_buff, use_key_length,

=== modified file 'storage/myisam/mi_rnext.c'
--- storage/myisam/mi_rnext.c   2012-01-13 14:50:02 +0000
+++ storage/myisam/mi_rnext.c   2012-08-18 05:37:44 +0000
@@ -17,6 +17,7 @@
 #include "myisamdef.h"

 #include "rt_index.h"
+#include "gist_index.h"

        /*
           Read next row with the same key as previous read
@@ -52,6 +53,14 @@
       error=rtree_get_first(info,inx,info->lastkey_length);
       break;
 #endif
+#ifdef HAVE_GIST_KEYS
+    case HA_KEY_ALG_GIST_RSTAR:
+      error=gist_get_first(info,inx,info->lastkey_length);
+      break;
+    case HA_KEY_ALG_GIST_RGUT83:
```

```
+       error=gist_get_first(info,inx,info->lastkey_length);
+       break;
+#endif
      case HA_KEY_ALG_BTREE:
      default:
        error=_mi_search_first(info,info->s->keyinfo+inx,
@@ -86,6 +95,21 @@
        error= rtree_get_next(info,inx,info->lastkey_length);
        break;
 #endif
+#ifdef HAVE_GIST_KEYS
+     case HA_KEY_ALG_GIST_RSTAR:
+       /*
+        Note (from rtree?)
+       */
+       error= gist_get_next(info,inx,info->lastkey_length);
+       break;
+     case HA_KEY_ALG_GIST_RGUT83:
+       /*
+        Note (from rtree?)
+       */
+       error= gist_get_next(info,inx,info->lastkey_length);
+       break;
+
+#endif
      case HA_KEY_ALG_BTREE:
      default:
        if (!changed)

=== modified file 'storage/myisam/mi_rnext_same.c'
--- storage/myisam/mi_rnext_same.c      2011-11-03 18:17:05 +0000
+++ storage/myisam/mi_rnext_same.c      2012-08-18 05:37:44 +0000
@@ -16,6 +16,7 @@

 #include "myisamdef.h"
 #include "rt_index.h"
+#include "gist_index.h"

         /*
            Read next row with the same key as previous read, but abort if
@@ -56,6 +57,28 @@
        }
        break;
 #endif
+#ifdef HAVE_GIST_KEYS
+     case HA_KEY_ALG_GIST_RSTAR:
+       if ((error=gist_find_next(info,inx,
+                                 myisam_read_vec[info->last_key_func])))
+       {
+        error=1;
+        my_errno=HA_ERR_END_OF_FILE;
+        info->lastpos= HA_OFFSET_ERROR;
+        break;
+       }
+       break;
+     case HA_KEY_ALG_GIST_RGUT83:
+       if ((error=gist_find_next(info,inx,
+                                 myisam_read_vec[info->last_key_func])))
+       {
+        error=1;
+        my_errno=HA_ERR_END_OF_FILE;
+        info->lastpos= HA_OFFSET_ERROR;
+        break;
+       }
+      break;
+#endif
     case HA_KEY_ALG_BTREE:
     default:
       if (!(info->update & HA_STATE_RNEXT_SAME))

=== modified file 'storage/myisam/myisamdef.h'
--- storage/myisam/myisamdef.h  2012-03-27 23:04:46 +0000
+++ storage/myisam/myisamdef.h  2012-08-18 05:37:44 +0000
@@ -301,6 +301,7 @@
   void *index_cond_func_arg;          /* parameter for the func */
   THR_LOCK_DATA lock;
   uchar *rtree_recursion_state;        /* For RTREE */
+  uchar *gist_recursion_state;         /* For GIST  */
   int rtree_recursion_depth;
 };
```

B.2 GiST implementation

```
=== added file 'mysql-test/t/gis-gist.test'
--- mysql-test/t/gis-gist.test  1970-01-01 00:00:00 +0000
+++ mysql-test/t/gis-gist.test  2012-08-18 11:29:56 +0000
@@ -0,0 +1,958 @@
+-- source include/have_geometry.inc
+
+#
+# test of rtree (using with spatial data)
+#
+--disable_warnings
+DROP TABLE IF EXISTS t1, t2;
+--enable_warnings
+
+CREATE TABLE t1 (
+  fid INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
+  g GEOMETRY NOT NULL,
+  SPATIAL KEY(g) USING GIST_RSTAR
+) ENGINE=MyISAM;
+
+SHOW CREATE TABLE t1;
+
+let $1=150;
+let $2=150;
+while ($1)
+{
+  eval INSERT INTO t1 (g) VALUES (GeomFromText('LineString($1 $1, $2 $2)'));
+  dec $1;
+  inc $2;
+}
+
+SELECT count(*) FROM t1;
+EXPLAIN SELECT fid, AsText(g) FROM t1 WHERE Within(g, GeomFromText('Polygon
     ((140 140,160 140,160 160,140 160,140 140))'));
+SELECT fid, AsText(g) FROM t1 WHERE Within(g, GeomFromText('Polygon((140
     140,160 140,160 160,140 160,140 140))'));
+
+DROP TABLE t1;
+
+CREATE TABLE t2 (
+  fid INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
+  g GEOMETRY NOT NULL
+) ENGINE=MyISAM;
```

```
+
+let $1=10;
+while ($1)
+{
+  let $2=10;
+  while ($2)
+  {
+    eval INSERT INTO t2 (g) VALUES (LineString(Point($1 * 10 - 9, $2 * 10 -
     9), Point($1 * 10, $2 * 10)));
+    dec $2;
+  }
+  dec $1;
+}
+
+ALTER TABLE t2 ADD SPATIAL KEY(g) USING GIST_RSTAR;
+SHOW CREATE TABLE t2;
+SELECT count(*) FROM t2;
+EXPLAIN SELECT fid, AsText(g) FROM t2 WHERE Within(g,
+  GeomFromText('Polygon((40 40,60 40,60 60,40 60,40 40))'));
+SELECT fid, AsText(g) FROM t2 WHERE Within(g,
+  GeomFromText('Polygon((40 40,60 40,60 60,40 60,40 40))'));
+
+let $1=10;
+while ($1)
+{
+  let $2=10;
+  while ($2)
+  {
+    eval DELETE FROM t2 WHERE Within(g, Envelope(GeometryFromWKB(Point($1 *
     10 - 9, $2 * 10 - 9), Point($1 * 10, $2 * 10))));
+    SELECT count(*) FROM t2;
+    dec $2;
+  }
+  dec $1;
+}
+
+DROP TABLE t2;
+
+drop table if exists t1;
+CREATE TABLE t1 (a geometry NOT NULL, SPATIAL (a) USING GIST_RSTAR );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+INSERT INTO t1 VALUES (GeomFromText("LINESTRING(100 100, 200 200, 300 300)")
     );
+check table t1;
+analyze table t1;
+drop table t1;
+
+#
+# The following crashed gis
+#
+
+CREATE TABLE t1 (
+  fid INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
+  g GEOMETRY NOT NULL,
+  SPATIAL KEY(g) USING GIST_RSTAR
+) ENGINE=MyISAM;
+
+INSERT INTO t1 (g) VALUES (GeomFromText('LineString(1 2, 2 3)')),(
     GeomFromText('LineString(1 2, 2 4)'));
+#select * from t1 where g<GeomFromText('LineString(1 2, 2 3)');
+drop table t1;
+
+CREATE TABLE t1 (
+  line LINESTRING NOT NULL,
+  kind ENUM('po', 'pp', 'rr', 'dr', 'rd', 'ts', 'cl') NOT NULL DEFAULT 'po',
+  name VARCHAR(32),
+
+  SPATIAL KEY (line) USING GIST_RSTAR
+
+
+) engine=myisam;
+
+ALTER TABLE t1 DISABLE KEYS;
+INSERT INTO t1 (name, kind, line) VALUES
+  ("Aadaouane", "pp", GeomFromText("POINT(32.816667 35.983333)")),
+  ("Aadassiye", "pp", GeomFromText("POINT(35.816667 36.216667)")),
+  ("Aadbel", "pp", GeomFromText("POINT(34.533333 36.100000)")),
+  ("Aadchit", "pp", GeomFromText("POINT(33.347222 35.423611)")),
+  ("Aadchite", "pp", GeomFromText("POINT(33.347222 35.423611)")),
+  ("Aadchit el Qoussair", "pp", GeomFromText("POINT(33.283333 35.483333)")),
+  ("Aaddaye", "pp", GeomFromText("POINT(36.716667 40.833333)")),
+  ("'Aadeissa", "pp", GeomFromText("POINT(32.823889 35.698889)")),
+  ("Aaderup", "pp", GeomFromText("POINT(55.216667 11.766667)")),
+  ("Qalaat Aades", "pp", GeomFromText("POINT(33.503333 35.377500)")),
+  ("A ad'ino", "pp", GeomFromText("POINT(54.812222 38.209167)")),
+  ("Aadi Noia", "pp", GeomFromText("POINT(13.800000 39.833333)")),
+  ("Aad La Macta", "pp", GeomFromText("POINT(35.779444 -0.129167)")),
+  ("Aadland", "pp", GeomFromText("POINT(60.366667 5.483333)")),
+  ("Aadliye", "pp", GeomFromText("POINT(33.366667 36.333333)")),
+  ("Aadloun", "pp", GeomFromText("POINT(33.403889 35.273889)")),
+  ("Aadma", "pp", GeomFromText("POINT(58.798333 22.663889)")),
+  ("Aadma Asundus", "pp", GeomFromText("POINT(58.798333 22.663889)")),
+  ("Aadmoun", "pp", GeomFromText("POINT(34.150000 35.650000)")),
+  ("Aadneram", "pp", GeomFromText("POINT(59.016667 6.933333)")),
+  ("Aadneskaar", "pp", GeomFromText("POINT(58.083333 6.983333)")),
+  ("Aadorf", "pp", GeomFromText("POINT(47.483333 8.900000)")),
+  ("Aadorp", "pp", GeomFromText("POINT(52.366667 6.633333)")),
+  ("Aadouane", "pp", GeomFromText("POINT(32.816667 35.983333)")),
+  ("Aadoui", "pp", GeomFromText("POINT(34.450000 35.983333)")),
+  ("Aadouiye", "pp", GeomFromText("POINT(34.583333 36.183333)")),
+  ("Aadouss", "pp", GeomFromText("POINT(33.512500 35.601389)")),
+  ("Aadra", "pp", GeomFromText("POINT(33.616667 36.500000)")),
+  ("Aadzi", "pp", GeomFromText("POINT(38.100000 64.850000)"));
+
+ALTER TABLE t1 ENABLE KEYS;
+INSERT INTO t1 (name, kind, line) VALUES ("austria", "pp", GeomFromText('
     LINESTRING(14.9906 48.9887,14.9946 48.9904,14.9947 48.9916)'));
+drop table t1;
+
+CREATE TABLE t1 (st varchar(100));
+INSERT INTO t1 VALUES ("Fake string");
+CREATE TABLE t2 (geom GEOMETRY NOT NULL, SPATIAL KEY gk(geom) USING
     GIST_RSTAR);
+--error 1416
+INSERT INTO t2 SELECT GeomFromText(st) FROM t1;
+drop table t1, t2;
+
+CREATE TABLE t1 (`geometry` geometry NOT NULL default '',SPATIAL KEY `gndx`
     (`geometry`) USING GIST_RSTAR) ENGINE=MyISAM DEFAULT CHARSET=latin1;
+
+INSERT INTO t1 (geometry) VALUES
+(PolygonFromText('POLYGON((-18.6086111000 -66.9327777000, -18.6055555000
+-66.8158332999, -18.7186111000 -66.8102777000, -18.7211111000
     -66.9269443999,
+-18.6086111000 -66.9327777000))'));
+
+INSERT INTO t1 (geometry) VALUES
+(PolygonFromText('POLYGON((-65.7402776999 -96.6686111000, -65.7372222000
+-96.5516666000, -65.8502777000 -96.5461111000, -65.8527777000
     -96.6627777000,
+-65.7402776999 -96.6686111000))'));
+check table t1 extended;
+
+drop table t1;
+
+#
+# Bug#17877 - Corrupted  index
+#
+CREATE TABLE t1 (
+  c1 geometry NOT NULL default '',
+  SPATIAL KEY i1 (c1) USING GIST_RSTAR
+) ENGINE=MyISAM DEFAULT CHARSET=latin1;
+INSERT INTO t1 (c1) VALUES (
+  PolygonFromText('POLYGON((-18.6086111000 -66.9327777000,
+                             -18.6055555000 -66.8158332999,
+                             -18.7186111000 -66.8102777000,
+                             -18.7211111000 -66.9269443999,
+                             -18.6086111000 -66.9327777000))'));
+# This showed a missing key.
+CHECK TABLE t1 EXTENDED;
+DROP TABLE t1;
+#
+CREATE TABLE t1 (
+  c1 geometry NOT NULL default '',
+  SPATIAL KEY i1 (c1) USING GIST_RSTAR
+) ENGINE=MyISAM DEFAULT CHARSET=latin1;
+INSERT INTO t1 (c1) VALUES (
+  PolygonFromText('POLYGON((-18.6086111000 -66.9327777000,
+                             -18.6055555000 -66.8158332999,
+                             -18.7186111000 -66.8102777000,
+                             -18.7211111000 -66.9269443999,
+                             -18.6086111000 -66.9327777000))'));
+INSERT INTO t1 (c1) VALUES (
+  PolygonFromText('POLYGON((-65.7402776999 -96.6686111000,
+                             -65.7372222000 -96.5516666000,
+                             -65.8502777000 -96.5461111000,
+                             -65.8527777000 -96.6627777000,
+                             -65.7402776999 -96.6686111000))'));
+# This is the same as the first insert to get a non-unique key.
+INSERT INTO t1 (c1) VALUES (
+  PolygonFromText('POLYGON((-18.6086111000 -66.9327777000,
+                             -18.6055555000 -66.8158332999,
+                             -18.7186111000 -66.8102777000,
+                             -18.7211111000 -66.9269443999,
+                             -18.6086111000 -66.9327777000))'));
+# This showed (and still shows) OK.
+CHECK TABLE t1 EXTENDED;
+DROP TABLE t1;
+
+#
+# Bug #21888: Query on GEOMETRY field using PointFromWKB() results in lost
     connection
+#
+CREATE TABLE t1 (foo GEOMETRY NOT NULL, SPATIAL INDEX(foo) USING GIST_RSTAR)
     ;
+INSERT INTO t1 (foo) VALUES (POINT(1,1));
+INSERT INTO t1 (foo) VALUES (POINT(1,0));
+INSERT INTO t1 (foo) VALUES (POINT(0,1));
+INSERT INTO t1 (foo) VALUES (POINT(0,0));
+SELECT 1 FROM t1 WHERE foo != POINT(0,0);
+DROP TABLE t1;
+
+#
+# Bug#25673 - spatial index corruption, error 126 incorrect key file for
     table
+#
+CREATE TABLE t1 (id bigint(12) unsigned NOT NULL auto_increment,
+  c2 varchar(15) collate utf8_bin default NULL,
+  c1 varchar(15) collate utf8_bin default NULL,
+  c3 varchar(10) collate utf8_bin default NULL,
+  spatial_point point NOT NULL,
+  PRIMARY KEY(id),
+  SPATIAL KEY (spatial_point) USING GIST_RSTAR
+  )ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
+#
```

```
+INSERT INTO t1 (c2, c1, c3, spatial_point) VALUES
+  ('y', 's', 'j', GeomFromText('POINT(167 74)')),
+  ('r', 'n', 'd', GeomFromText('POINT(215 118)')),
+  ('g', 'n', 'e', GeomFromText('POINT(203 98)')),
+  ('h', 'd', 'd', GeomFromText('POINT(54 193)')),
+  ('r', 'x', 'y', GeomFromText('POINT(47 69)')),
+  ('t', 'q', 'r', GeomFromText('POINT(109 42)')),
+  ('a', 'z', 'd', GeomFromText('POINT(0 154)')),
+  ('x', 'v', 'o', GeomFromText('POINT(174 131)')),
+  ('b', 'r', 'a', GeomFromText('POINT(114 253)')),
+  ('x', 'z', 'i', GeomFromText('POINT(163 21)')),
+  ('w', 'p', 'i', GeomFromText('POINT(42 102)')),
+  ('g', 'j', 'j', GeomFromText('POINT(170 133)')),
+  ('m', 'g', 'n', GeomFromText('POINT(28 22)')),
+  ('b', 'z', 'h', GeomFromText('POINT(174 28)')),
+  ('q', 'k', 'f', GeomFromText('POINT(233 73)')),
+  ('w', 'w', 'a', GeomFromText('POINT(124 200)')),
+  ('t', 'j', 'w', GeomFromText('POINT(252 101)')),
+  ('d', 'r', 'd', GeomFromText('POINT(98 18)')),
+  ('w', 'o', 'y', GeomFromText('POINT(165 31)')),
+  ('y', 'h', 't', GeomFromText('POINT(14 220)')),
+  ('d', 'p', 'u', GeomFromText('POINT(223 196)')),
+  ('g', 'y', 'g', GeomFromText('POINT(207 96)')),
+  ('x', 'm', 'n', GeomFromText('POINT(214 3)')),
+  ('g', 'v', 'e', GeomFromText('POINT(140 205)')),
+  ('g', 'm', 'm', GeomFromText('POINT(10 236)')),
+  ('i', 'r', 'j', GeomFromText('POINT(137 228)')),
+  ('w', 's', 'p', GeomFromText('POINT(115 6)')),
+  ('o', 'n', 'k', GeomFromText('POINT(158 129)')),
+  ('j', 'h', 'l', GeomFromText('POINT(129 72)')),
+  ('f', 'x', 'l', GeomFromText('POINT(139 207)')),
+  ('u', 'd', 'n', GeomFromText('POINT(125 109)')),
+  ('b', 'a', 'z', GeomFromText('POINT(30 32)')),
+  ('m', 'h', 'o', GeomFromText('POINT(251 251)')),
+  ('f', 'r', 'd', GeomFromText('POINT(243 211)')),
+  ('b', 'd', 'r', GeomFromText('POINT(232 80)')),
+  ('g', 'k', 'v', GeomFromText('POINT(15 100)')),
+  ('i', 'f', 'c', GeomFromText('POINT(109 66)')),
+  ('r', 't', 'j', GeomFromText('POINT(178 6)')),
+  ('y', 'n', 'f', GeomFromText('POINT(233 211)')),
+  ('f', 'y', 'm', GeomFromText('POINT(99 16)')),
+  ('z', 'q', 'l', GeomFromText('POINT(39 49)')),
+  ('j', 'c', 'r', GeomFromText('POINT(75 187)')),
+  ('c', 'y', 'y', GeomFromText('POINT(246 253)')),
+  ('w', 'u', 'd', GeomFromText('POINT(56 190)')),
+  ('n', 'q', 'm', GeomFromText('POINT(73 149)')),
+  ('d', 'y', 'a', GeomFromText('POINT(134 6)')),
+  ('z', 's', 'w', GeomFromText('POINT(216 225)')),
+  ('d', 'u', 'k', GeomFromText('POINT(132 70)')),
+  ('f', 'v', 't', GeomFromText('POINT(187 141)')),
+  ('r', 'r', 'a', GeomFromText('POINT(152 39)')),
+  ('y', 'p', 'o', GeomFromText('POINT(45 27)')),
+  ('p', 'n', 'm', GeomFromText('POINT(228 148)')),
+  ('e', 'g', 'e', GeomFromText('POINT(88 81)')),
+  ('m', 'a', 'h', GeomFromText('POINT(35 29)')),
+  ('m', 'h', 'f', GeomFromText('POINT(30 71)')),
+  ('h', 'k', 'i', GeomFromText('POINT(244 78)')),
+  ('z', 'v', 'd', GeomFromText('POINT(241 38)')),
+  ('q', 'l', 'j', GeomFromText('POINT(13 71)')),
+  ('s', 'p', 'g', GeomFromText('POINT(108 38)')),
+  ('q', 's', 'j', GeomFromText('POINT(92 101)')),
+  ('l', 'h', 'g', GeomFromText('POINT(120 78)')),
+  ('w', 't', 'b', GeomFromText('POINT(193 109)')),
+  ('b', 's', 's', GeomFromText('POINT(223 211)')),
+  ('w', 'w', 'y', GeomFromText('POINT(122 42)')),
+  ('q', 'c', 'c', GeomFromText('POINT(104 102)')),
+  ('w', 'g', 'n', GeomFromText('POINT(213 120)')),
+  ('p', 'q', 'a', GeomFromText('POINT(247 148)')),
+  ('c', 'z', 'e', GeomFromText('POINT(18 106)')),
+  ('z', 'u', 'n', GeomFromText('POINT(70 133)')),
+  ('j', 'n', 'x', GeomFromText('POINT(232 13)')),
+  ('e', 'h', 'f', GeomFromText('POINT(22 135)')),
+  ('w', 'l', 'f', GeomFromText('POINT(9 180)')),
+  ('a', 'v', 'q', GeomFromText('POINT(163 228)')),
+  ('i', 'z', 'o', GeomFromText('POINT(180 100)')),
+  ('e', 'c', 'l', GeomFromText('POINT(182 231)')),
+  ('c', 'k', 'o', GeomFromText('POINT(19 60)')),
+  ('q', 'f', 'p', GeomFromText('POINT(79 95)')),
+  ('m', 'd', 'r', GeomFromText('POINT(3 127)')),
+  ('m', 'e', 't', GeomFromText('POINT(136 154)')),
+  ('w', 'w', 'w', GeomFromText('POINT(102 15)')),
+  ('l', 'n', 'q', GeomFromText('POINT(71 196)')),
+  ('p', 'k', 'c', GeomFromText('POINT(47 139)')),
+  ('j', 'o', 'r', GeomFromText('POINT(177 128)')),
+  ('j', 'q', 'a', GeomFromText('POINT(170 6)')),
+  ('b', 'a', 'o', GeomFromText('POINT(63 211)')),
+  ('g', 's', 'o', GeomFromText('POINT(144 251)')),
+  ('w', 'u', 'w', GeomFromText('POINT(221 214)')),
+  ('g', 'a', 'm', GeomFromText('POINT(14 102)')),
+  ('u', 'q', 'z', GeomFromText('POINT(86 200)')),
+  ('k', 'a', 'm', GeomFromText('POINT(144 222)')),
+  ('j', 'u', 'r', GeomFromText('POINT(216 142)')),
+  ('q', 'k', 'v', GeomFromText('POINT(121 236)')),
+  ('p', 'o', 'r', GeomFromText('POINT(108 102)')),
+  ('b', 'd', 'x', GeomFromText('POINT(127 198)')),
+  ('k', 's', 'a', GeomFromText('POINT(2 150)')),
+  ('f', 'm', 'f', GeomFromText('POINT(160 191)')),
+  ('q', 'y', 'x', GeomFromText('POINT(98 111)')),
+  ('o', 'f', 'm', GeomFromText('POINT(232 218)')),
+  ('c', 'w', 'j', GeomFromText('POINT(156 165)')),
+  ('s', 'q', 'v', GeomFromText('POINT(98 161)'));
+SET @@RAND_SEED1=692635050, @@RAND_SEED2=297339954;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=159925977, @@RAND_SEED2=942570618;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=328169745, @@RAND_SEED2=410451954;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=178507359, @@RAND_SEED2=332493072;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=1034033013, @@RAND_SEED2=558966507;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+UPDATE t1 set spatial_point=GeomFromText('POINT(230 9)') where  c1 like 'y
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(95 35)') where  c1 like 'j
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(93 99)') where  c1 like 'a
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(19 81)') where  c1 like 'r
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(20 177)') where  c1 like 'h
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(221 193)') where  c1 like 'u
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(195 205)') where  c1 like 'd
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(15 213)') where  c1 like 'u
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(214 63)') where  c1 like 'n
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(243 171)') where  c1 like 'c
```

```
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(198 82)') where  c1 like 'y
     %';
+INSERT INTO t1 (c2, c1, c3, spatial_point) VALUES
+  ('f', 'y', 'p', GeomFromText('POINT(109 235)')),
+  ('b', 'e', 'v', GeomFromText('POINT(20 48)')),
+  ('i', 'u', 'f', GeomFromText('POINT(15 55)')),
+  ('o', 'r', 'z', GeomFromText('POINT(105 64)')),
+  ('a', 'p', 'a', GeomFromText('POINT(142 236)')),
+  ('g', 'i', 'k', GeomFromText('POINT(10 49)')),
+  ('x', 'z', 'x', GeomFromText('POINT(192 200)')),
+  ('c', 'v', 'r', GeomFromText('POINT(94 168)')),
+  ('y', 'z', 'e', GeomFromText('POINT(141 51)')),
+  ('h', 'm', 'd', GeomFromText('POINT(35 251)')),
+  ('v', 'm', 'q', GeomFromText('POINT(44 90)')),
+  ('j', 'l', 'z', GeomFromText('POINT(67 237)')),
+  ('i', 'v', 'a', GeomFromText('POINT(75 14)')),
+  ('b', 'q', 't', GeomFromText('POINT(153 33)')),
+  ('e', 'm', 'a', GeomFromText('POINT(247 49)')),
+  ('l', 'y', 'g', GeomFromText('POINT(56 203)')),
+  ('v', 'o', 'r', GeomFromText('POINT(90 54)')),
+  ('r', 'n', 'd', GeomFromText('POINT(135 83)')),
+  ('j', 't', 'u', GeomFromText('POINT(174 239)')),
+  ('u', 'n', 'g', GeomFromText('POINT(104 191)')),
+  ('p', 'q', 'y', GeomFromText('POINT(63 171)')),
+  ('o', 'q', 'p', GeomFromText('POINT(192 103)')),
+  ('f', 'x', 'e', GeomFromText('POINT(244 30)')),
+  ('n', 'x', 'c', GeomFromText('POINT(92 103)')),
+  ('r', 'q', 'z', GeomFromText('POINT(166 20)')),
+  ('s', 'a', 'j', GeomFromText('POINT(137 205)')),
+  ('z', 't', 't', GeomFromText('POINT(99 134)')),
+  ('o', 'm', 'j', GeomFromText('POINT(217 3)')),
+  ('n', 'h', 'j', GeomFromText('POINT(211 17)')),
+  ('v', 'v', 'a', GeomFromText('POINT(41 137)')),
+  ('q', 'o', 'j', GeomFromText('POINT(5 92)')),
+  ('z', 'y', 'e', GeomFromText('POINT(175 212)')),
+  ('j', 'z', 'h', GeomFromText('POINT(224 194)')),
+  ('a', 'g', 'm', GeomFromText('POINT(31 119)')),
+  ('p', 'c', 'f', GeomFromText('POINT(17 221)')),
+  ('t', 'h', 'k', GeomFromText('POINT(26 203)')),
+  ('u', 'w', 'p', GeomFromText('POINT(47 185)')),
+  ('z', 'a', 'c', GeomFromText('POINT(61 133)')),
+  ('u', 'k', 'a', GeomFromText('POINT(210 115)')),
+  ('k', 'f', 'h', GeomFromText('POINT(125 113)')),
+  ('t', 'v', 'y', GeomFromText('POINT(12 239)')),
+  ('u', 'v', 'd', GeomFromText('POINT(90 24)')),
+  ('m', 'y', 'w', GeomFromText('POINT(25 243)')),
+  ('d', 'n', 'g', GeomFromText('POINT(122 92)')),
+  ('z', 'm', 'f', GeomFromText('POINT(235 110)')),
+  ('q', 'd', 'f', GeomFromText('POINT(233 217)')),
+  ('a', 'v', 'u', GeomFromText('POINT(69 59)')),
+  ('x', 'k', 'p', GeomFromText('POINT(240 14)')),
+  ('i', 'v', 'r', GeomFromText('POINT(154 42)')),
+  ('w', 'h', 'l', GeomFromText('POINT(178 156)')),
+  ('d', 'h', 'n', GeomFromText('POINT(65 157)')),
+  ('c', 'k', 'z', GeomFromText('POINT(62 33)')),
+  ('e', 'l', 'w', GeomFromText('POINT(162 1)')),
+  ('r', 'f', 'i', GeomFromText('POINT(127 71)')),
+  ('q', 'm', 'c', GeomFromText('POINT(63 118)')),
+  ('c', 'h', 'u', GeomFromText('POINT(205 203)')),
+  ('d', 't', 'p', GeomFromText('POINT(234 87)')),
+  ('s', 'g', 'h', GeomFromText('POINT(149 34)')),
+  ('o', 'b', 'q', GeomFromText('POINT(159 179)')),
+  ('k', 'u', 'f', GeomFromText('POINT(202 254)')),
+  ('u', 'f', 'g', GeomFromText('POINT(70 15)')),
+  ('x', 's', 'b', GeomFromText('POINT(25 181)')),
+  ('s', 'c', 'g', GeomFromText('POINT(252 17)')),
+  ('a', 'c', 'f', GeomFromText('POINT(89 67)')),
+  ('r', 'e', 'q', GeomFromText('POINT(55 54)')),
+  ('f', 'i', 'k', GeomFromText('POINT(178 230)')),
+  ('p', 'e', 'l', GeomFromText('POINT(198 28)')),
+  ('w', 'o', 'd', GeomFromText('POINT(204 189)')),
+  ('c', 'a', 'g', GeomFromText('POINT(230 178)')),
+  ('r', 'o', 'e', GeomFromText('POINT(61 116)')),
+  ('w', 'a', 'a', GeomFromText('POINT(178 237)')),
+  ('v', 'd', 'e', GeomFromText('POINT(70 85)')),
+  ('k', 'c', 'e', GeomFromText('POINT(147 118)')),
+  ('d', 'q', 't', GeomFromText('POINT(218 77)')),
+  ('k', 'g', 'f', GeomFromText('POINT(192 113)')),
+  ('w', 'n', 'e', GeomFromText('POINT(92 124)')),
+  ('r', 'm', 'q', GeomFromText('POINT(130 65)')),
+  ('o', 'r', 'r', GeomFromText('POINT(174 233)')),
+  ('k', 'n', 't', GeomFromText('POINT(175 147)')),
+  ('q', 'm', 'r', GeomFromText('POINT(18 208)')),
+  ('l', 'd', 'i', GeomFromText('POINT(13 104)')),
+  ('w', 'o', 'y', GeomFromText('POINT(207 39)')),
+  ('p', 'u', 'o', GeomFromText('POINT(114 31)')),
+  ('y', 'a', 'p', GeomFromText('POINT(106 59)')),
+  ('a', 'x', 'z', GeomFromText('POINT(17 57)')),
+  ('v', 'h', 'x', GeomFromText('POINT(170 13)')),
+  ('t', 's', 'u', GeomFromText('POINT(84 18)')),
+  ('z', 'z', 'f', GeomFromText('POINT(250 197)')),
+  ('l', 'z', 't', GeomFromText('POINT(59 80)')),
+  ('j', 'g', 's', GeomFromText('POINT(54 26)')),
+  ('g', 'v', 'm', GeomFromText('POINT(89 98)')),
+  ('q', 'v', 'b', GeomFromText('POINT(39 240)')),
+  ('x', 'k', 'v', GeomFromText('POINT(246 207)')),
+  ('k', 'u', 'i', GeomFromText('POINT(105 111)')),
+  ('w', 'z', 's', GeomFromText('POINT(235 8)')),
+  ('d', 'd', 'd', GeomFromText('POINT(105 4)')),
+  ('c', 'z', 'q', GeomFromText('POINT(13 140)')),
+  ('m', 'k', 'i', GeomFromText('POINT(208 120)')),
+  ('g', 'a', 'g', GeomFromText('POINT(9 182)')),
+  ('z', 'j', 'r', GeomFromText('POINT(149 153)')),
+  ('h', 'f', 'g', GeomFromText('POINT(81 236)')),
+  ('m', 'e', 'q', GeomFromText('POINT(209 215)')),
+  ('c', 'h', 'y', GeomFromText('POINT(235 70)')),
+  ('i', 'e', 'g', GeomFromText('POINT(138 26)')),
+  ('m', 't', 'u', GeomFromText('POINT(119 237)')),
+  ('o', 'w', 's', GeomFromText('POINT(193 166)')),
+  ('f', 'm', 'q', GeomFromText('POINT(85 96)')),
+  ('x', 'l', 'x', GeomFromText('POINT(58 115)')),
+  ('x', 'q', 'u', GeomFromText('POINT(108 210)')),
+  ('b', 'h', 'i', GeomFromText('POINT(250 139)')),
+  ('y', 'd', 'x', GeomFromText('POINT(199 135)')),
+  ('w', 'h', 'p', GeomFromText('POINT(247 233)')),
+  ('p', 'z', 't', GeomFromText('POINT(148 249)')),
+  ('q', 'a', 'u', GeomFromText('POINT(174 78)')),
+  ('v', 't', 'm', GeomFromText('POINT(70 228)')),
+  ('t', 'n', 'f', GeomFromText('POINT(123 2)')),
+  ('x', 't', 'b', GeomFromText('POINT(35 50)')),
+  ('r', 'j', 'f', GeomFromText('POINT(200 51)')),
+  ('s', 'q', 'o', GeomFromText('POINT(23 184)')),
+  ('u', 'v', 'z', GeomFromText('POINT(7 113)')),
+  ('v', 'u', 'l', GeomFromText('POINT(145 190)')),
+  ('o', 'k', 'i', GeomFromText('POINT(161 122)')),
+  ('l', 'y', 'e', GeomFromText('POINT(17 232)')),
+  ('t', 'b', 'e', GeomFromText('POINT(120 50)')),
+  ('e', 's', 'u', GeomFromText('POINT(254 1)')),
+  ('d', 'd', 'u', GeomFromText('POINT(167 140)')),
+  ('o', 'b', 'x', GeomFromText('POINT(186 237)')),
+  ('m', 's', 's', GeomFromText('POINT(172 149)')),
+  ('t', 'y', 'a', GeomFromText('POINT(149 85)')),
+  ('x', 't', 'r', GeomFromText('POINT(10 165)')),
+  ('g', 'c', 'e', GeomFromText('POINT(95 165)')),
+  ('e', 'e', 'z', GeomFromText('POINT(98 65)')),
+  ('f', 'v', 'i', GeomFromText('POINT(149 144)')),
+  ('o', 'p', 'm', GeomFromText('POINT(233 67)')),
+  ('t', 'u', 'b', GeomFromText('POINT(109 215)')),
+  ('o', 'o', 'b', GeomFromText('POINT(130 48)')),
+  ('e', 'm', 'h', GeomFromText('POINT(88 189)')),
+  ('e', 'v', 'y', GeomFromText('POINT(55 29)')),
+  ('e', 't', 'm', GeomFromText('POINT(129 55)')),
+  ('p', 'p', 'i', GeomFromText('POINT(126 222)')),
+  ('c', 'i', 'c', GeomFromText('POINT(19 158)')),
+  ('c', 'b', 's', GeomFromText('POINT(13 19)')),
+  ('u', 'y', 'a', GeomFromText('POINT(114 5)')),
+  ('a', 'o', 'f', GeomFromText('POINT(227 232)')),
+  ('t', 'c', 'z', GeomFromText('POINT(63 62)')),
+  ('d', 'o', 'k', GeomFromText('POINT(48 228)')),
+  ('x', 'c', 'e', GeomFromText('POINT(204 2)')),
+  ('e', 'e', 'g', GeomFromText('POINT(125 43)')),
+  ('o', 'r', 'f', GeomFromText('POINT(171 140)'));
+UPDATE t1 set spatial_point=GeomFromText('POINT(163 157)') where  c1 like 'w
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(53 151)') where  c1 like 'd
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(96 183)') where  c1 like 'r
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(57 91)') where  c1 like 'q
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(202 110)') where  c1 like 'c
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(120 137)') where  c1 like 'w
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(207 147)') where  c1 like 'c
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(31 125)') where  c1 like 'e
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(27 36)') where  c1 like 'r
     %';
+INSERT INTO t1 (c2, c1, c3, spatial_point) VALUES
+  ('b', 'c', 'e', GeomFromText('POINT(41 137)')),
+  ('p', 'y', 'k', GeomFromText('POINT(50 22)')),
+  ('s', 'c', 'h', GeomFromText('POINT(208 173)')),
+  ('x', 'u', 'l', GeomFromText('POINT(199 175)')),
+  ('s', 'r', 'h', GeomFromText('POINT(85 192)')),
+  ('j', 'k', 'u', GeomFromText('POINT(18 25)')),
+  ('p', 'w', 'h', GeomFromText('POINT(152 197)')),
+  ('e', 'd', 'c', GeomFromText('POINT(229 3)')),
+  ('o', 'x', 'k', GeomFromText('POINT(187 155)')),
+  ('o', 'b', 'k', GeomFromText('POINT(208 150)')),
+  ('d', 'a', 'j', GeomFromText('POINT(70 87)')),
+  ('f', 'e', 'k', GeomFromText('POINT(156 96)')),
+  ('u', 'y', 'p', GeomFromText('POINT(239 193)')),
+  ('n', 'v', 'p', GeomFromText('POINT(223 98)')),
+  ('z', 'j', 'r', GeomFromText('POINT(87 89)')),
+  ('h', 'x', 'x', GeomFromText('POINT(92 0)')),
+  ('r', 'v', 'r', GeomFromText('POINT(159 139)')),
+  ('v', 'g', 'g', GeomFromText('POINT(16 229)')),
+  ('z', 'k', 'u', GeomFromText('POINT(99 52)')),
+  ('p', 'p', 'o', GeomFromText('POINT(105 125)')),
+  ('w', 'h', 'y', GeomFromText('POINT(105 154)')),
+  ('v', 'y', 'z', GeomFromText('POINT(134 238)')),
+  ('x', 'o', 'o', GeomFromText('POINT(178 88)')),
+  ('z', 'w', 'd', GeomFromText('POINT(123 60)')),
+  ('q', 'f', 'u', GeomFromText('POINT(64 90)')),
+  ('s', 'n', 't', GeomFromText('POINT(50 138)')),
+  ('v', 'p', 't', GeomFromText('POINT(114 91)')),
+  ('a', 'o', 'n', GeomFromText('POINT(78 43)')),
+  ('k', 'u', 'd', GeomFromText('POINT(185 161)')),
+  ('w', 'd', 'n', GeomFromText('POINT(25 92)')),
+  ('k', 'w', 'a', GeomFromText('POINT(59 238)')),
+  ('t', 'c', 'f', GeomFromText('POINT(65 87)')),
+  ('g', 's', 'p', GeomFromText('POINT(238 126)')),
+  ('d', 'n', 'y', GeomFromText('POINT(107 173)')),
+  ('l', 'a', 'w', GeomFromText('POINT(125 152)')),
+  ('m', 'd', 'j', GeomFromText('POINT(146 53)')),
+  ('q', 'm', 'c', GeomFromText('POINT(217 187)')),
+  ('i', 'r', 'r', GeomFromText('POINT(6 113)')),
+  ('e', 'j', 'b', GeomFromText('POINT(37 83)')),
+  ('w', 'w', 'h', GeomFromText('POINT(83 199)')),
+  ('k', 'b', 's', GeomFromText('POINT(170 64)')),
+  ('s', 'b', 'c', GeomFromText('POINT(163 130)')),
+  ('c', 'h', 'a', GeomFromText('POINT(141 3)')),
+  ('k', 'j', 'u', GeomFromText('POINT(143 76)')),
+  ('r', 'h', 'o', GeomFromText('POINT(243 92)')),
+  ('i', 'd', 'b', GeomFromText('POINT(205 13)')),
+  ('r', 'y', 'q', GeomFromText('POINT(138 8)')),
+  ('m', 'o', 'i', GeomFromText('POINT(36 45)')),
+  ('v', 'g', 'm', GeomFromText('POINT(0 40)')),
+  ('f', 'e', 'i', GeomFromText('POINT(76 6)')),
+  ('c', 'q', 'q', GeomFromText('POINT(115 248)')),
+  ('x', 'c', 'i', GeomFromText('POINT(29 74)')),
+  ('l', 's', 't', GeomFromText('POINT(83 18)')),
+  ('t', 't', 'a', GeomFromText('POINT(26 168)')),
+  ('u', 'n', 'x', GeomFromText('POINT(200 110)')),
+  ('j', 'b', 'd', GeomFromText('POINT(216 136)')),
+  ('s', 'p', 'w', GeomFromText('POINT(38 156)')),
+  ('f', 'b', 'v', GeomFromText('POINT(29 186)')),
+  ('v', 'e', 'r', GeomFromText('POINT(149 40)')),
+  ('v', 't', 'm', GeomFromText('POINT(184 24)')),
+  ('y', 'g', 'a', GeomFromText('POINT(219 105)')),
+  ('s', 'f', 'i', GeomFromText('POINT(114 130)')),
+  ('e', 'q', 'h', GeomFromText('POINT(203 135)')),
+  ('h', 'g', 'b', GeomFromText('POINT(9 208)')),
+  ('o', 'l', 'r', GeomFromText('POINT(245 79)')),
+  ('s', 's', 'v', GeomFromText('POINT(238 198)')),
+  ('w', 'w', 'z', GeomFromText('POINT(209 232)')),
+  ('v', 'd', 'n', GeomFromText('POINT(30 193)')),
+  ('q', 'w', 'k', GeomFromText('POINT(133 18)')),
+  ('o', 'h', 'o', GeomFromText('POINT(42 140)')),
+  ('f', 'f', 'h', GeomFromText('POINT(145 1)')),
+  ('u', 's', 'r', GeomFromText('POINT(70 62)')),
+  ('x', 'n', 'q', GeomFromText('POINT(33 86)')),
+  ('u', 'p', 'v', GeomFromText('POINT(232 220)')),
+  ('z', 'e', 'a', GeomFromText('POINT(130 69)')),
+  ('r', 'u', 'z', GeomFromText('POINT(243 241)')),
+  ('b', 'n', 't', GeomFromText('POINT(120 12)')),
+  ('u', 'f', 's', GeomFromText('POINT(190 212)')),
+  ('a', 'd', 'q', GeomFromText('POINT(235 191)')),
+  ('f', 'q', 'm', GeomFromText('POINT(176 2)')),
+  ('n', 'c', 's', GeomFromText('POINT(218 163)')),
+  ('e', 'm', 'h', GeomFromText('POINT(163 108)')),
+  ('c', 'f', 'l', GeomFromText('POINT(220 115)')),
+  ('c', 'v', 'q', GeomFromText('POINT(66 45)')),
+  ('w', 'v', 'x', GeomFromText('POINT(251 220)')),
+  ('f', 'w', 'z', GeomFromText('POINT(146 149)')),
+  ('h', 'n', 'h', GeomFromText('POINT(148 128)')),
+  ('y', 'k', 'v', GeomFromText('POINT(28 110)')),
+  ('c', 'x', 'q', GeomFromText('POINT(13 13)')),
+  ('e', 'd', 's', GeomFromText('POINT(91 190)')),
+  ('c', 'w', 'c', GeomFromText('POINT(10 231)')),
+  ('u', 'j', 'n', GeomFromText('POINT(250 21)')),
+  ('w', 'n', 'x', GeomFromText('POINT(141 69)')),
+  ('f', 'p', 'y', GeomFromText('POINT(228 246)')),
+  ('d', 'q', 'f', GeomFromText('POINT(194 22)')),
+  ('d', 'z', 'l', GeomFromText('POINT(233 181)')),
+  ('c', 'a', 'q', GeomFromText('POINT(183 96)')),
+  ('m', 'i', 'd', GeomFromText('POINT(117 226)')),
+  ('z', 'y', 'y', GeomFromText('POINT(62 81)')),
+  ('g', 'v', 'm', GeomFromText('POINT(66 158)'));
+SET @@RAND_SEED1=481064922, @@RAND_SEED2=438133497;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=280535103, @@RAND_SEED2=444518646;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=1072017234, @@RAND_SEED2=484203885;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=358851897, @@RAND_SEED2=358495224;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+SET @@RAND_SEED1=509031459, @@RAND_SEED2=675962925;
+DELETE FROM t1 ORDER BY RAND() LIMIT 10;
+UPDATE t1 set spatial_point=GeomFromText('POINT(61 203)') where  c1 like 'y
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(202 194)') where  c1 like 'f
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(228 18)') where  c1 like 'h
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(88 18)') where  c1 like 'l
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(176 94)') where  c1 like 'e
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(44 47)') where  c1 like 'g
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(95 191)') where  c1 like 'b
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(179 218)') where  c1 like 'y
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(239 40)') where  c1 like 'g
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(248 41)') where  c1 like 'q
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(167 82)') where  c1 like 't
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(13 104)') where  c1 like 'u
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(139 84)') where  c1 like 'a
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(145 108)') where  c1 like 'p
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(147 57)') where  c1 like 't
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(217 144)') where  c1 like 'n
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(160 224)') where  c1 like 'w
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(38 28)') where  c1 like 'j
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(104 114)') where  c1 like 'q
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(88 19)') where  c1 like 'c
     %';
+INSERT INTO t1 (c2, c1, c3, spatial_point) VALUES
+  ('f', 'x', 'p', GeomFromText('POINT(92 181)')),
+  ('s', 'i', 'c', GeomFromText('POINT(49 60)')),
+  ('c', 'c', 'i', GeomFromText('POINT(7 57)')),
+  ('n', 'g', 'k', GeomFromText('POINT(252 105)')),
+  ('g', 'b', 'm', GeomFromText('POINT(180 11)')),
+  ('u', 'l', 'r', GeomFromText('POINT(32 90)')),
+  ('c', 'x', 'e', GeomFromText('POINT(143 24)')),
+  ('x', 'u', 'a', GeomFromText('POINT(123 92)')),
+  ('s', 'b', 'h', GeomFromText('POINT(190 108)')),
+  ('c', 'x', 'b', GeomFromText('POINT(104 100)')),
+  ('i', 'd', 't', GeomFromText('POINT(214 104)')),
+  ('r', 'w', 'g', GeomFromText('POINT(29 67)')),
+  ('b', 'f', 'g', GeomFromText('POINT(149 46)')),
+  ('r', 'r', 'd', GeomFromText('POINT(242 196)')),
+  ('j', 'l', 'a', GeomFromText('POINT(90 196)')),
+  ('e', 't', 'b', GeomFromText('POINT(190 64)')),
+  ('l', 'x', 'w', GeomFromText('POINT(250 73)')),
+  ('q', 'y', 'r', GeomFromText('POINT(120 182)')),
+  ('s', 'j', 'a', GeomFromText('POINT(180 175)')),
+  ('n', 'i', 'y', GeomFromText('POINT(124 136)')),
+  ('s', 'x', 's', GeomFromText('POINT(176 209)')),
+  ('u', 'f', 's', GeomFromText('POINT(215 173)')),
+  ('m', 'j', 'x', GeomFromText('POINT(44 140)')),
+  ('v', 'g', 'x', GeomFromText('POINT(177 233)')),
+  ('u', 't', 'b', GeomFromText('POINT(136 197)')),
+  ('f', 'g', 'b', GeomFromText('POINT(10 8)')),
+  ('v', 'c', 'j', GeomFromText('POINT(13 81)')),
+  ('d', 's', 'q', GeomFromText('POINT(200 100)')),
+  ('a', 'p', 'j', GeomFromText('POINT(33 40)')),
+  ('i', 'c', 'g', GeomFromText('POINT(168 204)')),
+  ('k', 'h', 'i', GeomFromText('POINT(93 243)')),
+  ('s', 'b', 's', GeomFromText('POINT(157 13)')),
+  ('v', 'l', 'l', GeomFromText('POINT(103 6)')),
+  ('r', 'b', 'k', GeomFromText('POINT(244 137)')),
+  ('l', 'd', 'r', GeomFromText('POINT(162 254)')),
+  ('q', 'b', 'z', GeomFromText('POINT(136 246)')),
+  ('x', 'x', 'p', GeomFromText('POINT(120 37)')),
+  ('m', 'e', 'z', GeomFromText('POINT(203 167)')),
+  ('q', 'n', 'p', GeomFromText('POINT(94 119)')),
+  ('b', 'g', 'u', GeomFromText('POINT(93 248)')),
+  ('r', 'v', 'v', GeomFromText('POINT(53 88)')),
+  ('y', 'a', 'i', GeomFromText('POINT(98 219)')),
+  ('a', 's', 'g', GeomFromText('POINT(173 138)')),
+  ('c', 'a', 't', GeomFromText('POINT(235 135)')),
+  ('q', 'm', 'd', GeomFromText('POINT(224 208)')),
+  ('e', 'p', 'k', GeomFromText('POINT(161 238)')),
+  ('n', 'g', 'q', GeomFromText('POINT(35 204)')),
+  ('t', 't', 'x', GeomFromText('POINT(230 178)')),
+  ('w', 'f', 'a', GeomFromText('POINT(150 221)')),
+  ('z', 'm', 'z', GeomFromText('POINT(119 42)')),
+  ('l', 'j', 's', GeomFromText('POINT(97 96)')),
+  ('f', 'z', 'x', GeomFromText('POINT(208 65)')),
+  ('i', 'v', 'c', GeomFromText('POINT(145 79)')),
+  ('l', 'f', 'k', GeomFromText('POINT(83 234)')),
+  ('u', 'a', 's', GeomFromText('POINT(250 49)')),
+  ('o', 'k', 'p', GeomFromText('POINT(46 50)')),
+  ('d', 'e', 'z', GeomFromText('POINT(30 198)')),
+  ('r', 'r', 'l', GeomFromText('POINT(78 189)')),
+  ('y', 'l', 'f', GeomFromText('POINT(188 132)')),
+  ('d', 'q', 'm', GeomFromText('POINT(247 107)')),
+  ('p', 'j', 'n', GeomFromText('POINT(148 227)')),
+  ('b', 'o', 'i', GeomFromText('POINT(172 25)')),
+  ('e', 'v', 'd', GeomFromText('POINT(94 248)')),
+  ('q', 'd', 'f', GeomFromText('POINT(15 29)')),
+  ('w', 'b', 'b', GeomFromText('POINT(74 111)')),
+  ('g', 'q', 'f', GeomFromText('POINT(107 215)')),
+  ('o', 'h', 'r', GeomFromText('POINT(25 168)')),
+  ('u', 't', 'w', GeomFromText('POINT(251 188)')),
+  ('h', 's', 'w', GeomFromText('POINT(254 247)')),
+  ('f', 'f', 'b', GeomFromText('POINT(166 103)'));
+SET @@RAND_SEED1=866613816, @@RAND_SEED2=92289615;
+INSERT INTO t1 (c2, c1, c3, spatial_point) VALUES
+  ('l', 'c', 'l', GeomFromText('POINT(202 98)')),
+  ('k', 'c', 'b', GeomFromText('POINT(46 206)')),
+  ('r', 'y', 'm', GeomFromText('POINT(74 140)')),
+  ('y', 'z', 'd', GeomFromText('POINT(200 160)')),
+  ('s', 'y', 's', GeomFromText('POINT(156 205)')),
+  ('u', 'v', 'p', GeomFromText('POINT(86 82)')),
+  ('j', 's', 's', GeomFromText('POINT(91 233)')),
+  ('x', 'j', 'f', GeomFromText('POINT(3 14)')),
+  ('l', 'z', 'v', GeomFromText('POINT(123 156)')),
+  ('h', 'i', 'o', GeomFromText('POINT(145 229)')),
+  ('o', 'r', 'd', GeomFromText('POINT(15 22)')),
+  ('f', 'x', 't', GeomFromText('POINT(21 60)')),
+  ('t', 'g', 'h', GeomFromText('POINT(50 153)')),
+  ('g', 'u', 'b', GeomFromText('POINT(82 85)')),
+  ('v', 'a', 'p', GeomFromText('POINT(231 178)')),
+  ('n', 'v', 'o', GeomFromText('POINT(183 25)')),
+  ('j', 'n', 'm', GeomFromText('POINT(50 144)')),
+  ('e', 'f', 'i', GeomFromText('POINT(46 16)')),
+  ('d', 'w', 'a', GeomFromText('POINT(66 6)')),
+  ('f', 'x', 'a', GeomFromText('POINT(107 197)')),
+  ('m', 'o', 'a', GeomFromText('POINT(142 80)')),
+  ('q', 'l', 'g', GeomFromText('POINT(251 23)')),
+  ('c', 's', 's', GeomFromText('POINT(158 43)')),
+  ('y', 'd', 'o', GeomFromText('POINT(196 228)')),
+  ('d', 'p', 'l', GeomFromText('POINT(107 5)')),
+  ('h', 'a', 'b', GeomFromText('POINT(183 166)')),
+  ('m', 'w', 'p', GeomFromText('POINT(19 59)')),
+  ('b', 'y', 'o', GeomFromText('POINT(178 30)')),
+  ('x', 'w', 'i', GeomFromText('POINT(168 94)')),
+  ('t', 'k', 'z', GeomFromText('POINT(171 5)')),
+  ('r', 'm', 'a', GeomFromText('POINT(222 19)')),
+  ('u', 'v', 'e', GeomFromText('POINT(224 80)')),
+  ('q', 'r', 'k', GeomFromText('POINT(212 218)')),
+  ('d', 'p', 'j', GeomFromText('POINT(169 7)')),
+  ('d', 'r', 'v', GeomFromText('POINT(193 23)')),
+  ('n', 'y', 'y', GeomFromText('POINT(130 178)')),
+  ('m', 'z', 'r', GeomFromText('POINT(81 200)')),
+  ('j', 'e', 'w', GeomFromText('POINT(145 239)')),
+  ('v', 'h', 'x', GeomFromText('POINT(24 105)')),
+  ('z', 'm', 'a', GeomFromText('POINT(175 129)')),
+  ('b', 'c', 'v', GeomFromText('POINT(213 10)')),
+  ('t', 't', 'u', GeomFromText('POINT(2 129)')),
+  ('r', 's', 'v', GeomFromText('POINT(209 192)')),
+  ('x', 'p', 'g', GeomFromText('POINT(43 63)')),
+  ('t', 'e', 'u', GeomFromText('POINT(139 210)')),
+  ('l', 'e', 't', GeomFromText('POINT(245 148)')),
+  ('a', 'i', 'k', GeomFromText('POINT(167 195)')),
+  ('m', 'o', 'h', GeomFromText('POINT(206 120)')),
+  ('g', 'z', 's', GeomFromText('POINT(169 240)')),
+  ('z', 'u', 's', GeomFromText('POINT(202 120)')),
+  ('i', 'b', 'a', GeomFromText('POINT(216 18)')),
+  ('w', 'y', 'g', GeomFromText('POINT(119 236)')),
+  ('h', 'y', 'p', GeomFromText('POINT(161 24)'));
+UPDATE t1 set spatial_point=GeomFromText('POINT(33 100)') where  c1 like 't
     %';
+UPDATE t1 set spatial_point=GeomFromText('POINT(41 46)') where  c1 like 'f
     %';
+CHECK TABLE t1 EXTENDED;
+DROP TABLE t1;
+
+#
+# Bug #30286 spatial index cause corruption and server crash!
+#
+
+create table t1 (a geometry not null, spatial index(a));
+insert into t1 values (POINT(1.1517219314031e+164, 131072));
+insert into t1 values (POINT(9.1248812352444e+192, 2.9740338169556e+284));
+insert into t1 values (POINT(4.7783097267365e-299, -0));
+insert into t1 values (POINT(1.49166814624e-154, 2.0880974297595e-53));
+insert into t1 values (POINT(4.0917382598702e+149, 1.2024538023802e+111));
+insert into t1 values (POINT(2.0349165139404e+236, 2.9993936277913e-241));
+insert into t1 values (POINT(2.5243548967072e-29, 1.2024538023802e+111));
+insert into t1 values (POINT(0, 6.9835074892995e-251));
+insert into t1 values (POINT(2.0880974297595e-53, 3.1050361846014e+231));
+insert into t1 values (POINT(2.8728483499323e-188, 2.4600631144627e+260));
+insert into t1 values (POINT(3.0517578125e-05, 2.0349165139404e+236));
+insert into t1 values (POINT(1.1517219314031e+164, 1.1818212630766e-125));
+insert into t1 values (POINT(2.481040258324e-265, 5.7766220027675e-275));
+insert into t1 values (POINT(2.0880974297595e-53, 2.5243548967072e-29));
+insert into t1 values (POINT(5.7766220027675e-275, 9.9464647281957e+86));
+insert into t1 values (POINT(2.2181357552967e+130, 3.7857669957337e-270));
+insert into t1 values (POINT(4.5767114681874e-246, 3.6893488147419e+19));
+insert into t1 values (POINT(4.5767114681874e-246, 3.7537584144024e+255));
+insert into t1 values (POINT(3.7857669957337e-270, 1.8033161362863e-130));
+insert into t1 values (POINT(0, 5.8774717541114e-39));
+insert into t1 values (POINT(1.1517219314031e+164, 2.2761049594727e-159));
+insert into t1 values (POINT(6.243497100632e+144, 3.7857669957337e-270));
+insert into t1 values (POINT(3.7857669957337e-270, 2.6355494858076e-82));
+insert into t1 values (POINT(2.0349165139404e+236, 3.8518598887745e-34));
+insert into t1 values (POINT(4.6566128730774e-10, 2.0880974297595e-53));
+insert into t1 values (POINT(2.0880974297595e-53, 1.8827498946116e-183));
+insert into t1 values (POINT(1.8033161362863e-130, 9.1248812352444e+192));
+insert into t1 values (POINT(4.7783097267365e-299, 2.2761049594727e-159));
+insert into t1 values (POINT(1.94906280228e+289, 1.2338789709327e-178));
+drop table t1;
+
+# End of 4.1 tests
+
+#
+# bug #21790 (UNKNOWN ERROR on NULLs in RTree)
+#
+CREATE TABLE t1(foo GEOMETRY NOT NULL, SPATIAL INDEX(foo) USING GIST_RSTAR )
     ;
+--error 1048
+INSERT INTO t1(foo) VALUES (NULL);
+--error 1416
+INSERT INTO t1() VALUES ();
+--error 1416
+INSERT INTO t1(foo) VALUES ('');
+DROP TABLE t1;
+
+#
+# Bug #23578: Corruption prevents Optimize table from working properly with
     a
+#             spatial index
+#
+
+CREATE TABLE t1 (a INT AUTO_INCREMENT, b POINT NOT NULL, KEY (a), SPATIAL
     KEY (b) USING GIST_RSTAR);
+
+INSERT INTO t1 (b) VALUES (GeomFromText('POINT(1 2)'));
+INSERT INTO t1 (b) SELECT b FROM t1;
+INSERT INTO t1 (b) SELECT b FROM t1;
+INSERT INTO t1 (b) SELECT b FROM t1;
+INSERT INTO t1 (b) SELECT b FROM t1;
```

```
+INSERT INTO t1 (b) SELECT b FROM t1;
+
+OPTIMIZE TABLE t1;
+DROP TABLE t1;
+
+
+#
+# Bug #29070: Error in spatial index
+#
+
+CREATE TABLE t1 (a INT, b GEOMETRY NOT NULL, SPATIAL KEY b(b) USING
     GIST_RSTAR);
+INSERT INTO t1 VALUES (1, GEOMFROMTEXT('LINESTRING(1102218.456 1,2000000 2)
     '));
+INSERT INTO t1 VALUES (2, GEOMFROMTEXT('LINESTRING(1102218.456 1,2000000 2)
     '));
+
+# must return the same number as the next select
+SELECT COUNT(*) FROM t1 WHERE
+  MBRINTERSECTS(b, GEOMFROMTEXT('LINESTRING(1 1,1102219 2)') );
+SELECT COUNT(*) FROM t1 IGNORE INDEX (b) WHERE
+  MBRINTERSECTS(b, GEOMFROMTEXT('LINESTRING(1 1,1102219 2)') );
+
+DROP TABLE t1;
+
+
+--echo #
+--echo # Bug #48258: Assertion failed when using a spatial index
+--echo #
+CREATE TABLE t1(a LINESTRING NOT NULL, SPATIAL KEY(a) USING GIST_RSTAR);
+INSERT INTO t1 VALUES
+  (GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1 -1, -1 1, 1 1)')),
+  (GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1 -1, -1 1, 1 1)'));
+EXPLAIN SELECT 1 FROM t1 WHERE a = GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1
     -1, -1 1, 1 1)');
+SELECT 1 FROM t1 WHERE a = GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1 -1, -1
     1, 1 1)');
+EXPLAIN SELECT 1 FROM t1 WHERE a < GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1
     -1, -1 1, 1 1)');
+SELECT 1 FROM t1 WHERE a < GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1 -1, -1
     1, 1 1)');
+EXPLAIN SELECT 1 FROM t1 WHERE a <= GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1
      -1, -1 1, 1 1)');
+SELECT 1 FROM t1 WHERE a <= GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1 -1, -1
     1, 1 1)');
+EXPLAIN SELECT 1 FROM t1 WHERE a > GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1
     -1, -1 1, 1 1)');
+SELECT 1 FROM t1 WHERE a > GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1 -1, -1
     1, 1 1)');
+EXPLAIN SELECT 1 FROM t1 WHERE a >= GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1
      -1, -1 1, 1 1)');
+SELECT 1 FROM t1 WHERE a >= GEOMFROMTEXT('LINESTRING(-1 -1, 1 -1, -1 -1, -1
     1, 1 1)');
+DROP TABLE t1;
+
+
+--echo #
+--echo # Bug #51357: crash when using handler commands on spatial indexes
+--echo #
+
+CREATE TABLE t1(a GEOMETRY NOT NULL,SPATIAL INDEX a(a) USING GIST_RSTAR);
+HANDLER t1 OPEN;
+HANDLER t1 READ a FIRST;
+HANDLER t1 READ a NEXT;
+HANDLER t1 READ a PREV;
```

```
+HANDLER t1 READ a LAST;
+HANDLER t1 CLOSE;
+
+# second crash fixed when the tree has changed since the last search.
+HANDLER t1 OPEN;
+HANDLER t1 READ a FIRST;
+INSERT INTO t1 VALUES (GeomFromText('Polygon((40 40,60 40,60 60,40 60,40 40)
    )'));
+--echo # should not crash
+--disable_result_log
+HANDLER t1 READ a NEXT;
+--enable_result_log
+HANDLER t1 CLOSE;
+
+DROP TABLE t1;
+
+
+--echo End of 5.0 tests.
+
+
+--echo #
+--echo # Bug #57323/11764487: myisam corruption with insert ignore
+--echo # and invalid spatial data
+--echo #
+
+CREATE TABLE t1(a LINESTRING NOT NULL, b GEOMETRY NOT NULL,
+  SPATIAL KEY(a) USING GIST_RSTAR, SPATIAL KEY(b) USING GIST_RSTAR) ENGINE=
    MyISAM;
+INSERT INTO t1 VALUES(GEOMFROMTEXT("point (0 0)"), GEOMFROMTEXT("point (1 1)
    "));
+--error ER_CANT_CREATE_GEOMETRY_OBJECT
+INSERT IGNORE INTO t1 SET a=GEOMFROMTEXT("point (-6 0)"), b=GEOMFROMTEXT("
    error");
+--error ER_CANT_CREATE_GEOMETRY_OBJECT
+INSERT IGNORE INTO t1 SET a=GEOMFROMTEXT("point (-6 0)"), b=NULL;
+SELECT ASTEXT(a), ASTEXT(b) FROM t1;
+DROP TABLE t1;
+
+CREATE TABLE t1(a INT NOT NULL, b GEOMETRY NOT NULL,
+  KEY(a), SPATIAL KEY(b) USING GIST_RSTAR) ENGINE=MyISAM;
+INSERT INTO t1 VALUES(0, GEOMFROMTEXT("point (1 1)"));
+--error ER_CANT_CREATE_GEOMETRY_OBJECT
+INSERT IGNORE INTO t1 SET a=0, b=GEOMFROMTEXT("error");
+--error ER_CANT_CREATE_GEOMETRY_OBJECT
+INSERT IGNORE INTO t1 SET a=1, b=NULL;
+SELECT a, ASTEXT(b) FROM t1;
+DROP TABLE t1;
+
+--echo End of 5.1 tests

=== modified file 'storage/myisam/CMakeLists.txt'
--- storage/myisam/CMakeLists.txt      2012-08-18 05:37:44 +0000
+++ storage/myisam/CMakeLists.txt      2012-08-18 11:29:56 +0000
@@ -26,7 +26,7 @@
                                mi_unique.c mi_update.c mi_write.c rt_index.c
                                     rt_key.c rt_mbr.c
                                rt_split.c sort.c sp_key.c mi_extrafunc.h
                                    myisamdef.h
                                rt_index.h mi_rkey.c
-                              gist_index.h gist_index.c)
+                              gist_index.h gist_index.c gist_key.h gist_key
    .c gist_functions.h gist_functions.c sp_reinsert.h)

 MYSQL_ADD_PLUGIN(myisam ${MYISAM_SOURCES}
   STORAGE_ENGINE

=== added file 'storage/myisam/gist_functions.c'
--- storage/myisam/gist_functions.c     1970-01-01 00:00:00 +0000
+++ storage/myisam/gist_functions.c     2012-08-18 11:29:56 +0000
@@ -0,0 +1,66 @@
+/*
+   Copyright (c) 2012 Monty Program AB & Vangelis Katsikaros
+
+   This program is free software; you can redistribute it and/or modify
+   it under the terms of the GNU General Public License as published by
+   the Free Software Foundation; version 2 of the License.
+
+   This program is distributed in the hope that it will be useful,
+   but WITHOUT ANY WARRANTY; without even the implied warranty of
+   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+   GNU General Public License for more details.
+
+   You should have received a copy of the GNU General Public License
+   along with this program; if not, write to the Free Software
+   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301  USA
+*/
+
+#include "myisamdef.h"
+
+#ifdef HAVE_GIST_KEYS
+
+#include "rt_index.h" // for rtree_split_page
+#include "rt_mbr.h"   // for rtree_combine_rect
+
+// TODO now it's just a wrapper: convert to GiST proper wrapper
+int gist_split_page(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *page, uchar *
     key,
+                      uint key_length, my_off_t *new_page_offs)
+{
+  DBUG_ENTER("gist_split_page");
+
+  DBUG_PRINT("gist", ("key_alg: %d", keyinfo->key_alg));
+  if (keyinfo->key_alg == HA_KEY_ALG_GIST_RSTAR ){
+    DBUG_PRINT("gist", ("will call rtree_split_page" ) );
+    DBUG_RETURN((rtree_split_page(info, keyinfo, page, key, key_length,
+                                  new_page_offs) ? -1 : 1));
+  }
+  else{
+    DBUG_PRINT("gist", ("Unkown key_alg: will fail with assert"));
+    // this should never happen
+    DBUG_ASSERT(0);
+  }
+}
+
+
+
+
+// rtree_combine_rect wrapper
+// PROBABLY not needed, replaced combine_rect with set_mbr
+/* int gist_adjust_key(HA_KEYSEG *keyseg, uchar* a, uchar* b, uchar* c,  */
+/*                      uint key_length) */
+/* { */
+/*   DBUG_ENTER("gist_adjust_key"); */
+
+/*   if (keyinfo->key_alg == HA_KEY_ALG_GIST_RSTAR ){ */
+/*     DBUG_RETURN((rtree_combine_rect(keyinfo->seg, k, key, k, key_length))
     ;     */
+/*   } */
+/*   else{ */
+/*     // this should never happen */
+/*     // TODO ASSERT */
+/*     DBUG_RETURN(-1); */
+/*   } */
+
+/* } */
+
+
+#endif /*HAVE_GIST_KEYS*/

=== added file 'storage/myisam/gist_functions.h'
--- storage/myisam/gist_functions.h     1970-01-01 00:00:00 +0000
+++ storage/myisam/gist_functions.h     2012-08-18 11:29:56 +0000
@@ -0,0 +1,26 @@
+/* Copyright (C) 2012 Monty Program AB & Vangelis Katsikaros
+
+   This program is free software; you can redistribute it and/or modify
+   it under the terms of the GNU General Public License as published by
+   the Free Software Foundation; version 2 of the License.
+
+   This program is distributed in the hope that it will be useful,
+   but WITHOUT ANY WARRANTY; without even the implied warranty of
+   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+   GNU General Public License for more details.
+
+   You should have received a copy of the GNU General Public License
+   along with this program; if not, write to the Free Software
+   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
   */
+
+
+#ifndef _gist_functions_h
+#define _gist_functions_h
+
+#ifdef HAVE_GIST_KEYS
+
+int gist_split_page(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *page, uchar *
     key,
+                   uint key_length, my_off_t *new_page_offs);
+
+#endif /*HAVE_GIST_KEYS*/
+#endif /* _gist_functions_h */

=== modified file 'storage/myisam/gist_index.c'
--- storage/myisam/gist_index.c 2012-08-18 05:37:44 +0000
+++ storage/myisam/gist_index.c 2012-08-18 11:29:56 +0000
@@ -17,21 +17,158 @@

 #ifdef HAVE_GIST_KEYS

+#include "gist_key.h"                                  // TODO can gist_key.
     h and gist_functions.h be combined?
 #include "gist_index.h"
-
-typedef struct st_page_level
-{
-  uint level;
-  my_off_t offs;
-} stPageLevel;
-
-typedef struct st_page_list
-{
-  ulong n_pages;
-  ulong m_pages;
-  stPageLevel *pages;
-} stPageList;
-
+#include "gist_functions.h"
+
+// These 2 are needed for rtree_pick_key                // TODO remove
     function from here to gist_functions.h. What about static?
+#include "rt_index.h"
+#include "rt_mbr.h"
+
+
+
+/*
+  Fill reinsert page buffer
+
+  RETURN
+    -1 Error
+    0  OK
+*/
+
+static int gist_fill_reinsert_list(stPageList *ReinsertList, my_off_t page,
+                                   int level)
+{
+  DBUG_ENTER("gist_fill_reinsert_list");
+  DBUG_PRINT("gist", ("page: %lu  level: %d", (ulong) page, level));
+  if (ReinsertList->n_pages == ReinsertList->m_pages)
+  {
+    ReinsertList->m_pages += REINSERT_BUFFER_INC;
+    if (!(ReinsertList->pages = (stPageLevel*)my_realloc((uchar*)
     ReinsertList->pages,
+      ReinsertList->m_pages * sizeof(stPageLevel), MYF(MY_ALLOW_ZERO_PTR))))
+      goto err1;
+  }
+  /* save page to ReinsertList */
+  ReinsertList->pages[ReinsertList->n_pages].offs = page;
+  ReinsertList->pages[ReinsertList->n_pages].level = level;
+  ReinsertList->n_pages++;
+  DBUG_RETURN(0);
+
+err1:
+  DBUG_RETURN(-1); /* purecov: inspected */
+}
+
+
+/*
+   Find next key in gist-tree according to search_flag recursively
+
+   NOTES
+     Used in gist_find_first() and gist_find_next()
+
+   RETURN
+     -1         Error
+     0   Found
+     1   Not found
+*/
+
+static int gist_find_req(MI_INFO *info, MI_KEYDEF *keyinfo, uint search_flag
     ,
+                         uint nod_cmp_flag, my_off_t page, int level)
+{
+  uchar *k;
+  uchar *last;
+  uint nod_flag;
+  int res;
+  uchar *page_buf;
+  int k_len;
+  uint *saved_key = (uint*) (info->gist_recursion_state) + level;
+
+  DBUG_ENTER("gist_find_req");
+  if (!(page_buf = (uchar*)my_alloca((uint)keyinfo->block_length)))
+  {
+    my_errno = HA_ERR_OUT_OF_MEM;
+    DBUG_RETURN(-1);
+  }
+  if (!_mi_fetch_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf, 0))
+    goto err1;
+  nod_flag = mi_test_if_nod(page_buf);
+
+  k_len = keyinfo->keylength - info->s->base.rec_reflength;
+
+  if(info->gist_recursion_depth >= level)
+  {
+    k = page_buf + *saved_key;
+  }
+  else
+  {
+    k = rt_PAGE_FIRST_KEY(page_buf, nod_flag);
+  }
+  last = rt_PAGE_END(page_buf);
+
+  for (; k < last; k = rt_PAGE_NEXT_KEY(k, k_len, nod_flag))
+  {
+    if (nod_flag)
+    {
+      /* this is an internal node in the tree */
+      if (!(res = gist_key_cmp(keyinfo->seg, info->first_mbr_key, k,
+                          info->last_rkey_length, nod_cmp_flag)))
+      {
+        switch ((res = gist_find_req(info, keyinfo, search_flag,
       nod_cmp_flag,
+                                   _mi_kpos(nod_flag, k), level + 1)))
+        {
+          case 0: /* found - exit from recursion */
+            *saved_key = (uint) (k - page_buf);
+            goto ok;
+          case 1: /* not found - continue searching */
+            info->gist_recursion_depth = level;
+            break;
+          default: /* error */
+          case -1:
+            goto err1;
+        }
+      }
+    }
+    else
+    {
+      /* this is a leaf */
+      if (!gist_key_cmp(keyinfo->seg, info->first_mbr_key, k,
+                        info->last_rkey_length, search_flag))
+      {
+        uchar *after_key = rt_PAGE_NEXT_KEY(k, k_len, nod_flag);
+        info->lastpos = _mi_dpos(info, 0, after_key);
+        info->lastkey_length = k_len + info->s->base.rec_reflength;
+        memcpy(info->lastkey, k, info->lastkey_length);
+        info->gist_recursion_depth = level;
+        *saved_key = (uint) (last - page_buf);
+
+        if (after_key < last)
+        {
```

```
+          info->int_keypos = info->buff;
+          info->int_maxpos = info->buff + (last - after_key);
+          memcpy(info->buff, after_key, last - after_key);
+          info->buff_used = 0;
+        }
+        else
+        {
+         info->buff_used = 1;
+        }
+
+        res = 0;
+        goto ok;
+      }
+    }
+  }
+  info->lastpos = HA_OFFSET_ERROR;
+  my_errno = HA_ERR_KEY_NOT_FOUND;
+  res = 1;
+
+ok:
+  my_afree((uchar*)page_buf);
+  DBUG_RETURN(res);
+
+err1:
+  my_afree((uchar*)page_buf);
+  info->lastpos = HA_OFFSET_ERROR;
+  DBUG_RETURN(-1);
+}



@@ -55,19 +192,42 @@
 int gist_find_first(MI_INFO *info, uint keynr, uchar *key, uint key_length,
                     uint search_flag)
 {
-
   my_off_t root;
-  //uint nod_cmp_flag;
-  //MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
-  DBUG_ENTER("gist_find_first"); // no DBUG were initially used
+  uint nod_cmp_flag;
+  MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
+
+  DBUG_ENTER("gist_find_first");
+  /*
+    At the moment index can only properly handle the
+    MBR_INTERSECT, so we use it for all sorts of queries.
+    TODO: better searsh for CONTAINS/WITHIN.
+  */
+  search_flag= nod_cmp_flag= MBR_INTERSECT;
+
   if ((root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
   {
     my_errno= HA_ERR_END_OF_FILE;
-    return -1;
+    DBUG_RETURN(-1);
   }
+
+  /*
+    Save searched key, include data pointer.
+    The data pointer is required if the search_flag contains MBR_DATA.
+    (minimum bounding rectangle)
+  */
+  memcpy(info->first_mbr_key, key, keyinfo->keylength);
+  info->last_rkey_length = key_length;
```

```
+
+  info->gist_recursion_depth = -1;
+  info->buff_used = 1;
+
+  /*
+    TODO better search for CONTAINS/WITHIN.
+    nod_cmp_flag= ((search_flag & (MBR_EQUAL | MBR_WITHIN)) ?
+                   MBR_WITHIN : MBR_INTERSECT);
+  */
   DBUG_PRINT("gist", ("info: %lu  keynr: %u  key: %s key_length: %u
       search_flag: %u", (ulong) info, keynr, key, key_length, search_flag )
        );
-  DBUG_RETURN(0); /* sceleton return */
-
+  DBUG_RETURN( gist_find_req(info, keyinfo, search_flag, nod_cmp_flag, root,
     0) );
 }


@@ -86,24 +246,192 @@
      1   Not found
 */

+/*
+   Find next key in gist-tree according to search_flag condition
+
+  SYNOPSIS
+   gist_find_next()
+   info                          Handler to MyISAM file
+   uint keynr           Key number to use
+   search_flag          Bitmap of flags how to do the search
+
+   RETURN
+     -1  Error
+     0   Found
+     1   Not found
+*/
+
 int gist_find_next(MI_INFO *info, uint keynr, uint search_flag)
 {
   my_off_t root;
   uint nod_cmp_flag;
   MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;

-  nod_cmp_flag = 0;
-  root = 0;
+  DBUG_ENTER("gist_find_next");
+  /*
+    At the moment index can only properly handle the
+    MBR_INTERSECT, so we use it for all sorts of queries.
+    TODO: better searsh for CONTAINS/WITHIN.
+  */
+  search_flag= nod_cmp_flag= MBR_INTERSECT;
+
   DBUG_PRINT("gist", ("info: %lu  keynr: %u  search_flag: %u", (ulong) info,
        keynr, search_flag ) );
   DBUG_PRINT("gist", ("keyinfo: %lu  keynr: %u  search_flag: %lu", (ulong)
       keyinfo, nod_cmp_flag, (ulong) root ) );

   if (info->update & HA_STATE_DELETED)
-    return gist_find_first(info, keynr, info->lastkey, info->lastkey_length,
-                           search_flag);
-
+    DBUG_RETURN( gist_find_first(info, keynr, info->lastkey, info->
     lastkey_length,
+                                 search_flag) );
+
+  if (!info->buff_used)
+  {
+    uchar *key= info->int_keypos;
+
+    while (key < info->int_maxpos)
+    {
+      if (!gist_key_cmp(keyinfo->seg, info->first_mbr_key, key,
+                        info->last_rkey_length, search_flag))
+      {
+        uchar *after_key = key + keyinfo->keylength;
+
+        info->lastpos= _mi_dpos(info, 0, after_key);
+        memcpy(info->lastkey, key, info->lastkey_length);
+
+        if (after_key < info->int_maxpos)
+         info->int_keypos= after_key;
+        else
+         info->buff_used= 1;
+        DBUG_RETURN(0);
+      }
+      key+= keyinfo->keylength;
+    }
+  }
+  if ((root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
+  {
     my_errno= HA_ERR_END_OF_FILE;
-    return -1;
-}
+    DBUG_RETURN(-1);
+  }
+
+  /*
+    TODO better search for CONTAINS/WITHIN.
+    nod_cmp_flag= (((search_flag & (MBR_EQUAL | MBR_WITHIN)) ?
+                    MBR_WITHIN : MBR_INTERSECT));
+  */
+  DBUG_RETURN(gist_find_req(info, keyinfo, search_flag, nod_cmp_flag, root,
     0));
+}
+
+
+
+
+/*
+  Get next key in gist-tree recursively
+
+  NOTES
+    Used in rtree_get_first() and rtree_get_next()
+
+  RETURN
+    -1  Error
+    0   Found
+    1   Not found
+*/
+
+static int gist_get_req(MI_INFO *info, MI_KEYDEF *keyinfo, uint key_length,
+                        my_off_t page, int level)
+{
+  uchar *k;
+  uchar *last;
+  uint nod_flag;
+  int res;
+  uchar *page_buf;
+  uint k_len;
+  uint *saved_key = (uint*) (info->gist_recursion_state) + level;
+
+  DBUG_ENTER("gist_find_req");
+
+  if (!(page_buf = (uchar*)my_alloca((uint)keyinfo->block_length)))
+    DBUG_RETURN(-1);
+  if (!_mi_fetch_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf, 0))
+    goto err1;
+  nod_flag = mi_test_if_nod(page_buf);
+
+  k_len = keyinfo->keylength - info->s->base.rec_reflength;
+
+  if(info->gist_recursion_depth >= level)
+  {
+    k = page_buf + *saved_key;
+    if (!nod_flag)
+    {
+      /* Only leaf pages contain data references. */
+      /* Need to check next key with data reference. */
+      k = rt_PAGE_NEXT_KEY(k, k_len, nod_flag);
+    }
+  }
+  else
+  {
+    k = rt_PAGE_FIRST_KEY(page_buf, nod_flag);
+  }
+  last = rt_PAGE_END(page_buf);
+
+  for (; k < last; k = rt_PAGE_NEXT_KEY(k, k_len, nod_flag))
+  {
+    if (nod_flag)
+    {
+      /* this is an internal node in the tree */
+      switch ((res = gist_get_req(info, keyinfo, key_length,
+                                  _mi_kpos(nod_flag, k), level + 1)))
+      {
+        case 0: /* found - exit from recursion */
+          *saved_key = (uint) (k - page_buf);
+          goto ok;
+        case 1: /* not found - continue searching */
+          info->gist_recursion_depth = level;
+          break;
+        default:
+        case -1: /* error */
+          goto err1;
+      }
+    }
+    else
+    {
+      /* this is a leaf */
+      uchar *after_key = rt_PAGE_NEXT_KEY(k, k_len, nod_flag);
+      info->lastpos = _mi_dpos(info, 0, after_key);
+      info->lastkey_length = k_len + info->s->base.rec_reflength;
+      memcpy(info->lastkey, k, info->lastkey_length);
+
+      info->gist_recursion_depth = level;
+      *saved_key = (uint) (k - page_buf);
+
+      if (after_key < last)
+      {
+        info->int_keypos = (uchar*)saved_key;
+        memcpy(info->buff, page_buf, keyinfo->block_length);
+        info->int_maxpos = rt_PAGE_END(info->buff);
+        info->buff_used = 0;
+       }
+       else
+       {
+        info->buff_used = 1;
+       }
+
+       res = 0;
+       goto ok;
+     }
+   }
+   info->lastpos = HA_OFFSET_ERROR;
+   my_errno = HA_ERR_KEY_NOT_FOUND;
+   res = 1;
+
+ok:
+   my_afree((uchar*)page_buf);
+   DBUG_RETURN(res);
+
+err1:
+   my_afree((uchar*)page_buf);
+   info->lastpos = HA_OFFSET_ERROR;
+   DBUG_RETURN(-1);
+}
+
+



@@ -121,19 +449,23 @@
    my_off_t root;
    MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;

+   DBUG_ENTER("gist_get_first");
    DBUG_PRINT("gist", ("nfo: %lu  keynr: %u  key_length: %u, keyinfo: %p", (
         ulong) info, keynr, key_length, keyinfo ) );

-
    if ((root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
    {
      my_errno= HA_ERR_END_OF_FILE;
-     return -1;
+     DBUG_RETURN(-1);
    }

-   return -1;
+   info->gist_recursion_depth = -1;
+   info->buff_used = 1;
+
+   DBUG_RETURN(gist_get_req(info, keyinfo, key_length, root, 0));
 }


+
 /*
    Get next key in gist-tree

@@ -142,21 +474,265 @@
      0  Found
      1  Not found
 */
-
 int gist_get_next(MI_INFO *info, uint keynr, uint key_length)
 {
    my_off_t root= info->s->state.key_root[keynr];
    MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
```

```

+  DBUG_ENTER("gist_get_next");
   DBUG_PRINT("gist", ("info: %lu  keynr: %u  key_length: %u, keyinfo: %p,
        root: %lu", (ulong) info, keynr, key_length, keyinfo, (ulong) root )
        );
-
+
   if (root == HA_OFFSET_ERROR)
   {
     my_errno= HA_ERR_END_OF_FILE;
-    return -1;
-  }
-
-  return -1;
+    DBUG_RETURN(-1);
+  }
+
+  if (!info->buff_used && !info->page_changed)
+  {
+    uint k_len = keyinfo->keylength - info->s->base.rec_reflength;
+    /* rt_PAGE_NEXT_KEY(info->int_keypos) */
+    uchar *key = info->buff + *(int*)info->int_keypos + k_len +
+                 info->s->base.rec_reflength;
+    /* rt_PAGE_NEXT_KEY(key) */
+    uchar *after_key = key + k_len + info->s->base.rec_reflength;
+
+    info->lastpos = _mi_dpos(info, 0, after_key);
+    info->lastkey_length = k_len + info->s->base.rec_reflength;
+    memcpy(info->lastkey, key, k_len + info->s->base.rec_reflength);
+
+    *(uint*)info->int_keypos = (uint) (key - info->buff);
+    if (after_key >= info->int_maxpos)
+    {
+      info->buff_used = 1;
+    }
+
+    DBUG_RETURN(0);
+  }
+
+  DBUG_RETURN(gist_get_req(info, keyinfo, key_length, root, 0));
+}
+
+
+
+static uchar *gist_pick_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                            uint key_length, uchar *page_buf, uint nod_flag)
+{
+  /*

      // TODO gist_penalty
+    TODO reform this to match the gist ChooseSubtree algorith:
+    loop all entires and calulate gist_Penalty( key_in_node, new_key )
+    K = entry e with the minimum penalty;
+
+    gist_Penalty(E1 , E2 ) {
+      if(rtree){
+        // rtree specific implementation
+        q = gist_Union(E1 , E2 )
+       return area(q)- area(E1 ).
+       }
+    }
+  */
+
+  if (keyinfo->key_alg == HA_KEY_ALG_GIST_RSTAR ){
+    return rtree_pick_key(info, keyinfo, key, key_length, page_buf,
+                         nod_flag);
+  }
+  // TODO assert this should never happen
+  return NULL;
+}
+
+
+
+
+/*
+  Go down and insert key into tree
+
+  RETURN
+    -1 Error
+    0  Child was not split
+    1  Child was split
+*/
+
+static int gist_insert_req(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                            uint key_length, my_off_t page, my_off_t *
     new_page,
+                            int ins_level, int level)
+{
+  uchar *k;
+  uint nod_flag;
+  uchar *page_buf;
+  int res;
+  DBUG_ENTER("gist_insert_req");
+
+  if (!(page_buf = (uchar*)my_alloca((uint)keyinfo->block_length +
+                                     HA_MAX_KEY_BUFF)))
+  {
+    my_errno = HA_ERR_OUT_OF_MEM;
+    DBUG_RETURN(-1); /* purecov: inspected */
+  }
+  if (!_mi_fetch_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf, 0))
+    goto err1;
+  nod_flag = mi_test_if_nod(page_buf);
+  DBUG_PRINT("gist", ("page: %lu  level: %d  ins_level: %d  nod_flag: %u",
+                       (ulong) page, level, ins_level, nod_flag));
+
+  if ((ins_level == -1 && nod_flag) ||       /* key: go down to leaf */
+      (ins_level > -1 && ins_level > level)) /* branch: go down to ins_level
      */
+  {
+    DBUG_PRINT("gist", ("go one level down"));
+    if ((k = gist_pick_key(info, keyinfo, key, key_length, page_buf,
                   // TODO pick key
+                             nod_flag)) == NULL)
+      goto err1;
+    switch ((res = gist_insert_req(info, keyinfo, key, key_length,
                     // ...
+                      _mi_kpos(nod_flag, k), new_page, ins_level, level + 1))
     )
+    {
+      case 0: /* child was not split */
+      {
+       DBUG_PRINT("gist", ("child was not split"));
+        gist_set_key_mbr(info, keyinfo, k, key_length,_mi_kpos(nod_flag, k))
     ;      // TODO adjust. REPLACED: rtree_combine_rect with
     rtree_set_key_mbr.
+        if (_mi_write_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf)
     )
+          goto err1;
+        goto ok;
+       }
+       case 1: /* child was split */
+       {
+        DBUG_PRINT("gist", ("child was split"));
+         uchar *new_key = page_buf + keyinfo->block_length + nod_flag;
+         /* set proper MBR for key */
+         if (gist_set_key_mbr(info, keyinfo, k, key_length,
                                   // TODO adjust
+                                   _mi_kpos(nod_flag, k)))
+           goto err1;
+         /* add new key for new page */
+         _mi_kpointer(info, new_key - nod_flag, *new_page);
+         if (gist_set_key_mbr(info, keyinfo, new_key, key_length, *new_page))
            // TODO adjust
+           goto err1;
+         res = gist_add_key(info, keyinfo, new_key, key_length,
                                // TODO gist_add_key
+                                  page_buf, new_page);
+         if (_mi_write_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf)
     )
+           goto err1;
+         goto ok;
+       }
+       default:
+       case -1: /* error */
+       {
+        DBUG_PRINT("gist", ("error"));
+         goto err1;
+       }
+     }
+   }
+   else
+   {
+     DBUG_PRINT("gist", ("don't go down: add key"));
+     res = gist_add_key(info, keyinfo, key, key_length, page_buf, new_page);
            // TODO gist_add_key
+     if (_mi_write_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf))
+       goto err1;
+     DBUG_PRINT("gist", ("added with res: %d", res));
+     goto ok;
+   }
+
+ok:
+   DBUG_PRINT("gist", ("ok: return %d", res));
+   my_afree((uchar*)page_buf);
+   DBUG_RETURN(res);
+
+err1:
+   DBUG_PRINT("gist", ("error"));
+   my_afree((uchar*)page_buf);
+   DBUG_RETURN(-1); /* purecov: inspected */
+}
+
+
+
+
+
+/*
+   Insert key into the tree
+
+   RETURN
+     -1 Error
+     0  Root was not split
+     1  Root was split
+*/
```

```
+
+static int gist_insert_level(MI_INFO *info, uint keynr, uchar *key,
+                             uint key_length, int ins_level)
+{
+  my_off_t old_root;
+  MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;
+  int res;
+  my_off_t new_page;
+  DBUG_ENTER("gist_insert_level");
+
+  if ((old_root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
+  {
+    DBUG_PRINT("gist", ("special install new root"));
+    if ((old_root = _mi_new(info, keyinfo, DFLT_INIT_HITS)) ==
    HA_OFFSET_ERROR)
+      DBUG_RETURN(-1);
+    info->buff_used = 1;
+    mi_putint(info->buff, 2, 0);
+    res = gist_add_key(info, keyinfo, key, key_length, info->buff, NULL);
           // TODO gist_add_key
+    if (_mi_write_keypage(info, keyinfo, old_root, DFLT_INIT_HITS, info->
    buff))
+      DBUG_RETURN(1);
+    info->s->state.key_root[keynr] = old_root;
+    DBUG_RETURN(res);
+  }
+
+  DBUG_PRINT("gist", ("calling gist_insert_req"));
+  switch ((res = gist_insert_req(info, keyinfo, key, key_length,
                   // TODO gist_insert_req
+                                old_root, &new_page, ins_level, 0)))
+  {
+    case 0: /* root was not split */
+    {
+      DBUG_PRINT("gist", ("root was not split"));
+      break;
+    }
+    case 1: /* root was split, grow a new root */
+    {
+      DBUG_PRINT("gist", ("root split, grow new root"));
+      uchar *new_root_buf= info->buff + info->s->base.max_key_block_length;
+      my_off_t new_root;
+      uchar *new_key;
+      uint nod_flag = info->s->base.key_reflength;
+
+      DBUG_PRINT("gist", ("root was split, grow a new root"));
+
+      mi_putint(new_root_buf, 2, nod_flag);
+      if ((new_root = _mi_new(info, keyinfo, DFLT_INIT_HITS)) ==
+         HA_OFFSET_ERROR)
+        goto err1;
+
+      new_key = new_root_buf + keyinfo->block_length + nod_flag;
+
+      _mi_kpointer(info, new_key - nod_flag, old_root);
+      if (gist_set_key_mbr(info, keyinfo, new_key, key_length, old_root))
          // TODO
+        goto err1;
+      if (gist_add_key(info, keyinfo, new_key, key_length, new_root_buf,
    NULL)  // TODO gist_add_key
+          == -1)
+        goto err1;
+      _mi_kpointer(info, new_key - nod_flag, new_page);
+      if (gist_set_key_mbr(info, keyinfo, new_key, key_length, new_page))
          // TODO
+        goto err1;
+      if (gist_add_key(info, keyinfo, new_key, key_length, new_root_buf,
     NULL)  // TODO gist_add_key
+          == -1)
+        goto err1;
+      if (_mi_write_keypage(info, keyinfo, new_root,
+                            DFLT_INIT_HITS, new_root_buf))
+        goto err1;
+      info->s->state.key_root[keynr] = new_root;
+      DBUG_PRINT("gist", ("new root page: %lu  level: %d  nod_flag: %u",
+                          (ulong) new_root, 0, mi_test_if_nod(new_root_buf)
     ));
+
+      break;
+err1:
+      DBUG_PRINT("gist", ("error during insert_level"));
+      DBUG_RETURN(-1); /* purecov: inspected */
+    }
+    default:
+    case -1: /* error */
+    {
+      DBUG_PRINT("gist", ("req returned error"));
+      break;
+    }
+  }
+  DBUG_RETURN(res);
 }


@@ -173,11 +749,158 @@
 int gist_insert(MI_INFO *info, uint keynr, uchar *key, uint key_length)
 {
   DBUG_ENTER("gist_insert");
-  /* DBUG_RETURN((!key_length || */
-  /*               (gist_insert_level(info, keynr, key, key_length, -1) ==
      -1)) ? */
-  /*              -1 : 0); */
   DBUG_PRINT("gist", ("info: %lu  keynr: %u  key: %s key_length: %u", (ulong
       ) info, keynr, key, key_length ) );
-  DBUG_RETURN(-1); /* sceleton return */
+  DBUG_RETURN((!key_length ||
+               (gist_insert_level(info, keynr, key, key_length, -1) == -1))
     ?
+              -1 : 0);
+}
+
+
+
+
+/*
+  Go down and delete key from the tree
+
+  RETURN
+    -1 Error
+    0  Deleted
+    1  Not found
+    2  Empty leaf
+*/
+
+static int gist_delete_req(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                           uint key_length, my_off_t page, uint *page_size,
+                           stPageList *ReinsertList, int level)
+{
+  uchar *k;
+  uchar *last;
+  ulong i;
+  uint nod_flag;
+  uchar *page_buf;
+  int res;
+  DBUG_ENTER("gist_delete_req");
+
+  if (!(page_buf = (uchar*)my_alloca((uint)keyinfo->block_length)))
+  {
+    my_errno = HA_ERR_OUT_OF_MEM;
+    DBUG_RETURN(-1); /* purecov: inspected */
+  }
+  if (!_mi_fetch_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf, 0))
+    goto err1;
+  nod_flag = mi_test_if_nod(page_buf);
+  DBUG_PRINT("gist", ("page: %lu  level: %d  nod_flag: %u",
+                      (ulong) page, level, nod_flag));
+
+  k = rt_PAGE_FIRST_KEY(page_buf, nod_flag);
+  last = rt_PAGE_END(page_buf);
+
+  for (i = 0; k < last; k = rt_PAGE_NEXT_KEY(k, key_length, nod_flag), ++i)
           // TODO iterate the keys
+  {
+    if (nod_flag)
+    {
+      /* not leaf */
+      if (!rtree_key_cmp(keyinfo->seg, key, k, key_length, MBR_WITHIN))
               // TODO compare
+      {
+        switch ((res = gist_delete_req(info, keyinfo, key, key_length,
               // TODO recursive
+                 _mi_kpos(nod_flag, k), page_size, ReinsertList, level + 1)
   ))
+        {
+          case 0: /* deleted */
+          {
+            /* test page filling */
+            if (*page_size + key_length >= rt_PAGE_MIN_SIZE(keyinfo->
   block_length))
+            {
+              /* OK */
+              /* Calculate a new key value (MBR) for the shrinked block. */
+              if (gist_set_key_mbr(info, keyinfo, k, key_length,
                      // TODO adjust
+                                   _mi_kpos(nod_flag, k)))
+                goto err1;
+              if (_mi_write_keypage(info, keyinfo, page,
+                                    DFLT_INIT_HITS, page_buf))
+                goto err1;
+            }
+            else
+            {
+              /*
+                Too small: delete key & add it descendant to reinsert list.
+                Store position and level of the block so that it can be
+                accessed later for inserting the remaining keys.
+              */
+              DBUG_PRINT("gist", ("too small. move block to reinsert list"))
   ;
+              if (gist_fill_reinsert_list(ReinsertList, _mi_kpos(nod_flag, k
   ),      // TODO reinsert fill
+                                          level + 1))
+                goto err1;
+              /*
+                Delete the key that references the block. This makes the
+                block disappear from the index. Hence we need to insert
+                its remaining keys later. Note: if the block is a branch
+                block, we do not only remove this block, but the whole
+                subtree. So we need to re-insert its keys on the same
+                level later to reintegrate the subtrees.
+              */
+              gist_delete_key(info, page_buf, k, key_length, nod_flag); // TODO delete key
+              if (_mi_write_keypage(info, keyinfo, page,
+                                    DFLT_INIT_HITS, page_buf))
+                goto err1;
+              *page_size = mi_getint(page_buf);
+            }
+
+            goto ok;
+          }
+          case 1: /* not found - continue searching */
+          {
+            break;
+          }
+          case 2: /* vacuous case: last key in the leaf */
+          {
+            gist_delete_key(info, page_buf, k, key_length, nod_flag);      // TODO delete key
+            if (_mi_write_keypage(info, keyinfo, page,
+                                  DFLT_INIT_HITS, page_buf))
+              goto err1;
+            *page_size = mi_getint(page_buf);
+            res = 0;
+            goto ok;
+          }
+          default: /* error */
+          case -1:
+          {
+            goto err1;
+          }
+        }
+      }
+    }
+    else
+    {
+      /* leaf */
+      if (!gist_key_cmp(keyinfo->seg, key, k, key_length, MBR_EQUAL | MBR_DATA))     // TODO compare
+      {
+        gist_delete_key(info, page_buf, k, key_length, nod_flag);           // TODO delete keys
+        *page_size = mi_getint(page_buf);
+        if (*page_size == 2)
+        {
+          /* last key in the leaf */
+          res = 2;
+          if (_mi_dispose(info, keyinfo, page, DFLT_INIT_HITS))
+            goto err1;
+        }
+        else
+        {
+          res = 0;
+          if (_mi_write_keypage(info, keyinfo, page, DFLT_INIT_HITS, page_buf))
+            goto err1;
+        }
+        goto ok;
+      }
+    }
+  }
+  res = 1;
+
+ok:
+  my_afree((uchar*)page_buf);
+  DBUG_RETURN(res);
+
+err1:
+  my_afree((uchar*)page_buf);
+  DBUG_RETURN(-1); /* purecov: inspected */
 }


@@ -204,16 +927,116 @@
     my_errno= HA_ERR_END_OF_FILE;
     DBUG_RETURN(-1); /* purecov: inspected */
   }
-  DBUG_PRINT("rtree", ("starting deletion at root page: %lu",
+  DBUG_PRINT("gist", ("starting deletion at root page: %lu",
                        (ulong) old_root));
-
-  page_size = 0;
   DBUG_PRINT("gist", ("info: %lu  keynr: %u  key: %s key_length: %u", (ulong
       ) info, keynr, key, key_length ) );
   DBUG_PRINT("gist", ("page_size: %u  ReinsertList: %p keyinfo: %p ",
      page_size, &ReinsertList, keyinfo ) );
-  DBUG_RETURN(-1); /* sceleton return */
+
+
+  ReinsertList.pages = NULL;
+  ReinsertList.n_pages = 0;
+  ReinsertList.m_pages = 0;
+
+  switch (gist_delete_req(info, keyinfo, key, key_length, old_root,
+                          &page_size, &ReinsertList, 0))                  //
     TODO gist recursive
+  {
+    case 2: /* empty */
+    {
+      info->s->state.key_root[keynr] = HA_OFFSET_ERROR;
+      DBUG_RETURN(0);
+    }
+    case 0: /* deleted */
+    {
+      uint nod_flag;
+      ulong i;
+      for (i = 0; i < ReinsertList.n_pages; ++i)
+      {
+        uchar *page_buf;
+        uchar *k;
+        uchar *last;
+
+        if (!(page_buf = (uchar*)my_alloca((uint)keyinfo->block_length)))
+        {
+          my_errno = HA_ERR_OUT_OF_MEM;
+          goto err1;
+        }
+        if (!_mi_fetch_keypage(info, keyinfo, ReinsertList.pages[i].offs,
+                               DFLT_INIT_HITS, page_buf, 0))
+          goto err1;
+        nod_flag = mi_test_if_nod(page_buf);
+        DBUG_PRINT("gist", ("reinserting keys from "
+                            "page: %lu  level: %d  nod_flag: %u",
+                            (ulong) ReinsertList.pages[i].offs,
+                            ReinsertList.pages[i].level, nod_flag));
+
+         k = rt_PAGE_FIRST_KEY(page_buf, nod_flag);
+         last = rt_PAGE_END(page_buf);
+         for (; k < last; k = rt_PAGE_NEXT_KEY(k, key_length, nod_flag))
+         {
+           int res;
+           if ((res= gist_insert_level(info, keynr, k, key_length,
+                                       ReinsertList.pages[i].level)) == -1)
          // TODO reinsert
+           {
+             my_afree((uchar*)page_buf);
+             goto err1;
+           }
+           if (res)
+           {
+             ulong j;
+             DBUG_PRINT("gist", ("root has been split, adjust levels"));
+             for (j= i; j < ReinsertList.n_pages; j++)
+             {
+               ReinsertList.pages[j].level++;
+               DBUG_PRINT("gist", ("keys from page: %lu  now level: %d",
+                                   (ulong) ReinsertList.pages[i].offs,
+                                   ReinsertList.pages[i].level));
+             }
+           }
+         }
+         my_afree((uchar*)page_buf);
+         if (_mi_dispose(info, keyinfo, ReinsertList.pages[i].offs,
+             DFLT_INIT_HITS))
+           goto err1;
+       }
+       if (ReinsertList.pages)
+        my_free(ReinsertList.pages);
+
+       /* check for redundant root (not leaf, 1 child) and eliminate */
+       if ((old_root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
+         goto err1;
+       if (!_mi_fetch_keypage(info, keyinfo, old_root, DFLT_INIT_HITS,
+                              info->buff, 0))
+         goto err1;
+       nod_flag = mi_test_if_nod(info->buff);
+       page_size = mi_getint(info->buff);
+       if (nod_flag && (page_size == 2 + key_length + nod_flag))
+       {
+         my_off_t new_root = _mi_kpos(nod_flag,
+                                      rt_PAGE_FIRST_KEY(info->buff, nod_flag)
     );
+         if (_mi_dispose(info, keyinfo, old_root, DFLT_INIT_HITS))
+           goto err1;
+         info->s->state.key_root[keynr] = new_root;
+       }
+       info->update= HA_STATE_DELETED;
+       DBUG_RETURN(0);
+
+err1:
+       DBUG_RETURN(-1); /* purecov: inspected */
+     }
+     case 1: /* not found */
+     {
+       my_errno = HA_ERR_KEY_NOT_FOUND;
+       DBUG_RETURN(-1); /* purecov: inspected */
+     }
+     default:
+     case -1: /* error */
+     {
```

```
+       DBUG_RETURN(-1); /* purecov: inspected */
+     }
+   }
 }


-
-#endif /*HAVE_RTREE_KEYS*/
+
+#endif /*HAVE_GIST_KEYS*/


=== modified file 'storage/myisam/gist_index.h'
--- storage/myisam/gist_index.h 2012-08-18 05:37:44 +0000
+++ storage/myisam/gist_index.h 2012-08-18 11:29:56 +0000
@@ -16,6 +16,8 @@
 #ifndef _gist_index_h
 #define _gist_index_h

+#include "sp_reinsert.h"
+
 #ifdef HAVE_GIST_KEYS

 #define gist_PAGE_FIRST_KEY(page, nod_flag) (page + 2 + nod_flag)
@@ -35,5 +37,8 @@
 int gist_get_first(MI_INFO *info, uint keynr, uint key_length);
 int gist_get_next(MI_INFO *info, uint keynr, uint key_length);

+int gist_split_page(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *page, uchar *
    key,
+                   uint key_length, my_off_t *new_page_offs);
+
 #endif /*HAVE_GIST_KEYS*/
 #endif /* _gist_index_h */

=== modified file 'storage/myisam/gist_key.c'
--- storage/myisam/gist_key.c   2012-08-18 05:37:44 +0000
+++ storage/myisam/gist_key.c   2012-08-18 11:29:56 +0000
@@ -19,5 +19,127 @@
 #include "gist_index.h"
 #include "gist_key.h"

+/* GIST_RSTAR */
+#include "rt_index.h"
+#include "rt_key.h"
+#include "rt_mbr.h"
+
+/*
+  Add key to the page
+
+  RESULT VALUES
+    -1          Error
+    0  Not split
+    1  Split
+*/
+
+int gist_add_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                 uint key_length, uchar *page_buf, my_off_t *new_page)
+{
+  uint page_size = mi_getint(page_buf);
+  uint nod_flag = mi_test_if_nod(page_buf);
+  DBUG_ENTER("gist_add_key");
+
+  if (page_size + key_length + info->s->base.rec_reflength <=
+      keyinfo->block_length)
+  {
+    DBUG_PRINT("gist", ("checking..."));
+    /* split won't be necessary */
+    if (nod_flag)
+    {
+      DBUG_PRINT("gist", ("split won't be necessary"));
+      /* save key */
+      DBUG_ASSERT(_mi_kpos(nod_flag, key) < info->state->key_file_length);
+      memcpy(rt_PAGE_END(page_buf), key - nod_flag, key_length + nod_flag);
      // rt_mbr
+      page_size += key_length + nod_flag;
+    }
+    else
+    {
+      DBUG_PRINT("gist", ("save key"));
+      /* save key */
+      DBUG_ASSERT(_mi_dpos(info, nod_flag, key + key_length +
+                           info->s->base.rec_reflength) <
+                  info->state->data_file_length + info->s->base.
    pack_reclength);
+      memcpy(rt_PAGE_END(page_buf), key, key_length +
                            // rt_mbr
+                                         info->s->base.rec_reflength);
+      page_size += key_length + info->s->base.rec_reflength;
+    }
+    mi_putint(page_buf, page_size, nod_flag);
+    DBUG_RETURN(0);
+  }
+
+  DBUG_PRINT("gist", ("will call gist_split_page"));
+  DBUG_RETURN(gist_split_page(info, keyinfo, page_buf, key, key_length,
         // gist_split_page
+                                new_page));
+}
+
+/*
+  Calculate and store key MBR
+*/
+
+int gist_set_key_mbr(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                     uint key_length, my_off_t child_page)
+{
+  DBUG_ENTER("gist_set_key_mbr");
+
+  if (!_mi_fetch_keypage(info, keyinfo, child_page,
+                         DFLT_INIT_HITS, info->buff, 0))
+    DBUG_RETURN(-1); /* purecov: inspected */
+
+  if (keyinfo->key_alg == HA_KEY_ALG_GIST_RSTAR ){
+    DBUG_RETURN(rtree_page_mbr(info, keyinfo->seg, info->buff, key,
    key_length));
+  }
+  else{
+    // this should never happen
+    // TODO ASSERT
+    DBUG_RETURN(-1);
+  }
+}
+
+
+/*
+  Delete key from the page
+*/
+int gist_delete_key(MI_INFO *info, uchar *page_buf, uchar *key,
+                    uint key_length, uint nod_flag)
+{
+  uint16 page_size = mi_getint(page_buf);
+  uchar *key_start;
+
+  key_start= key - nod_flag;
+  if (!nod_flag)
+    key_length += info->s->base.rec_reflength;
+
+  memmove(key_start, key + key_length, page_size - key_length -
+         (key - page_buf));
+  page_size -= key_length + nod_flag;
+
+  mi_putint(page_buf, page_size, nod_flag);
+  return 0;
+}
+
+
+// TODO now it's just a wrapper: convert to GiST proper wrapper
+/*
+ Compares two keys a and b depending on nextflag
+ nextflag can contain these flags:
+   MBR_INTERSECT(a,b)  a overlaps b
+   MBR_CONTAIN(a,b)    a contains b
+   MBR_DISJOINT(a,b)   a disjoint b
+   MBR_WITHIN(a,b)     a within   b
+   MBR_EQUAL(a,b)      All coordinates of MBRs are equal
+   MBR_DATA(a,b)       Data reference is the same
+ Returns 0 on success.
+*/
+
+int gist_key_cmp(HA_KEYSEG *keyseg, uchar *b, uchar *a, uint key_length,
+                  uint nextflag)
+{
+  DBUG_ENTER("gist_set_key_cmp");
+
+  DBUG_RETURN(rtree_key_cmp(keyseg, b, a, key_length, nextflag));
+}
+
+

 #endif /*HAVE_GIST_KEYS*/

=== modified file 'storage/myisam/gist_key.h'
--- storage/myisam/gist_key.h   2012-08-18 05:37:44 +0000
+++ storage/myisam/gist_key.h   2012-08-18 11:29:56 +0000
@@ -19,5 +19,18 @@

 #ifdef HAVE_GIST_KEYS

+int gist_add_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                 uint key_length, uchar *page_buf, my_off_t *new_page);
+
+int gist_set_key_mbr(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                    uint key_length, my_off_t child_page);
+
+int gist_delete_key(MI_INFO *info, uchar *page_buf, uchar *key,
+                   uint key_length, uint nod_flag);
+
+int gist_key_cmp(HA_KEYSEG *keyseg, uchar *b, uchar *a, uint key_length,
+                uint nextflag);
+
+
 #endif /*HAVE_GIST_KEYS*/
 #endif /* _gist_key_h */

=== modified file 'storage/myisam/ha_myisam.cc'
--- storage/myisam/ha_myisam.cc 2012-08-18 05:37:44 +0000
+++ storage/myisam/ha_myisam.cc 2012-08-18 11:29:56 +0000
@@ -701,7 +701,9 @@
     flags= 0;
   else
   if ((table_share->key_info[inx].flags & HA_SPATIAL ||
-      table_share->key_info[inx].algorithm == HA_KEY_ALG_RTREE))
+       table_share->key_info[inx].algorithm == HA_KEY_ALG_RTREE ||
+       table_share->key_info[inx].algorithm == HA_KEY_ALG_GIST_RSTAR ||
+       table_share->key_info[inx].algorithm == HA_KEY_ALG_GIST_RGUT83))
   {
     /* All GIS scans are non-ROR scans. We also disable
         IndexConditionPushdown */
     flags= HA_READ_NEXT | HA_READ_PREV | HA_READ_RANGE |

=== modified file 'storage/myisam/mi_check.c'
--- storage/myisam/mi_check.c   2012-08-18 05:37:44 +0000
+++ storage/myisam/mi_check.c   2012-08-18 11:29:56 +0000
@@ -1225,18 +1225,23 @@
              */
              int search_result;
 #ifdef HAVE_RTREE_KEYS
-            if (keyinfo->flag & HA_SPATIAL)
+            //if (keyinfo->flag & HA_SPATIAL)
+            if(keyinfo->key_alg == HA_KEY_ALG_RTREE)
             {
+              DBUG_PRINT("info", ("rtree"));
               search_result = rtree_find_first(info, key, info->lastkey,
                                                key_length, MBR_EQUAL | MBR_DATA);
             }
             else
 #endif
 #ifdef HAVE_GIST_KEYS
-            if (search_result && keyinfo->flag & HA_GIST_INDEX)
+              //if (keyinfo->flag & HA_GIST_INDEX)
+            if(keyinfo->key_alg == HA_KEY_ALG_GIST_RSTAR ||
+               keyinfo->key_alg == HA_KEY_ALG_GIST_RGUT83)
             {
+              DBUG_PRINT("info", ("gist tree"));
               search_result = gist_find_first(info, key, info->lastkey,
-                                              key_length, 0);
+                                              key_length, MBR_EQUAL | MBR_DATA);
             }
             else
 #endif

=== modified file 'storage/myisam/mi_dynrec.c'
--- storage/myisam/mi_dynrec.c  2012-04-10 06:28:13 +0000
+++ storage/myisam/mi_dynrec.c  2012-08-18 11:29:56 +0000
@@ -295,6 +295,7 @@
   error=write_dynamic_record(info,rec_buff+ALIGN_SIZE(
       MI_MAX_DYN_BLOCK_HEADER),
                              reclength2);
   my_afree(rec_buff);
+  DBUG_PRINT("info",("Finished _mi_write_blob_record. Res: %d", error));
   return(error);
 }

@@ -375,8 +376,10 @@
       goto err;
   } while (reclength);

+  DBUG_PRINT("info",("Return with ok 0"));
   DBUG_RETURN(0);
 err:
+  DBUG_PRINT("info",("Return with error 1"));
   DBUG_RETURN(1);
 }


=== modified file 'storage/myisam/mi_key.c'
--- storage/myisam/mi_key.c     2011-11-03 18:17:05 +0000
+++ storage/myisam/mi_key.c     2012-08-18 11:29:56 +0000
@@ -225,7 +225,9 @@
   DBUG_ENTER("_mi_pack_key");

   /* "one part" rtree key is 2*SPDIMS part key in MyISAM */
-  if (info->s->keyinfo[keynr].key_alg == HA_KEY_ALG_RTREE)
+  if (info->s->keyinfo[keynr].key_alg == HA_KEY_ALG_RTREE ||
+      info->s->keyinfo[keynr].key_alg == HA_KEY_ALG_GIST_RSTAR ||
+      info->s->keyinfo[keynr].key_alg == HA_KEY_ALG_GIST_RGUT83)
     keypart_map= (((key_part_map)1) << (2*SPDIMS)) - 1;

   /* only key prefixes are supported */

=== modified file 'storage/myisam/mi_open.c'
--- storage/myisam/mi_open.c    2012-08-18 05:37:44 +0000
+++ storage/myisam/mi_open.c    2012-08-18 11:29:56 +0000
@@ -782,6 +782,7 @@
     share->base.pack_reclength+= share->base.pack_bits;
     if (share->base.blobs)
     {
+      DBUG_PRINT("info",("Will call _mi_write_blob_record"));
       share->update_record=_mi_update_blob_record;
       share->write_record=_mi_write_blob_record;
     }

=== modified file 'storage/myisam/mi_range.c'
--- storage/myisam/mi_range.c   2012-01-13 14:50:02 +0000
+++ storage/myisam/mi_range.c   2012-08-18 11:29:56 +0000
@@ -92,6 +92,68 @@
     break;
   }
 #endif
+#ifdef HAVE_GIST_KEYS
+  case HA_KEY_ALG_GIST_RSTAR:
+  {
+    // all this come from case HA_KEY_ALG_RTREE:
+    uchar * key_buff;
+    uint start_key_len;
+
+    /*
+      The problem is that the optimizer doesn't support
+      RTree keys properly at the moment.
+      Hope this will be fixed some day.
+      But now NULL in the min_key means that we
+      didn't make the task for the RTree key
+      and expect BTree functionality from it.
+      As it's not able to handle such request
+      we return the error.
+    */
+    if (!min_key)
+    {
```

```
+       res= HA_POS_ERROR;
+       break;
+     }
+     key_buff= info->lastkey+info->s->base.max_key_length;
+     start_key_len= _mi_pack_key(info,inx, key_buff,
+                                  (uchar*) min_key->key, min_key->keypart_map,
+                                  (HA_KEYSEG**) 0);
+     res= rtree_estimate(info, inx, key_buff, start_key_len,
+                         myisam_read_vec[min_key->flag]);
+     res= res ? res : 1;                       /* Don't return 0 */
+     break;
+   }
+   case HA_KEY_ALG_GIST_RGUT83:
+   {
+     // all this come from case HA_KEY_ALG_RTREE:
+     uchar * key_buff;
+     uint start_key_len;
+
+     /*
+       The problem is that the optimizer doesn't support
+       RTree keys properly at the moment.
+       Hope this will be fixed some day.
+       But now NULL in the min_key means that we
+       didn't make the task for the RTree key
+       and expect BTree functionality from it.
+       As it's not able to handle such request
+       we return the error.
+     */
+     if (!min_key)
+     {
+       res= HA_POS_ERROR;
+       break;
+     }
+     key_buff= info->lastkey+info->s->base.max_key_length;
+     start_key_len= _mi_pack_key(info,inx, key_buff,
+                                  (uchar*) min_key->key, min_key->keypart_map,
+                                  (HA_KEYSEG**) 0);
+     res= rtree_estimate(info, inx, key_buff, start_key_len,
+                         myisam_read_vec[min_key->flag]);
+     res= res ? res : 1;                       /* Don't return 0 */
+     break;
+   }
+#endif
    case HA_KEY_ALG_BTREE:
    default:
      start_pos= (min_key ?  _mi_record_pos(info, min_key->key,

=== modified file 'storage/myisam/mi_rkey.c'
--- storage/myisam/mi_rkey.c    2012-08-18 05:37:44 +0000
+++ storage/myisam/mi_rkey.c    2012-08-18 11:29:56 +0000
@@ -84,6 +84,7 @@
    switch (info->s->keyinfo[inx].key_alg) {
 #ifdef HAVE_RTREE_KEYS
    case HA_KEY_ALG_RTREE:
+     DBUG_PRINT("info", ("Rtree"));
      if (rtree_find_first(info,inx,key_buff,use_key_length,nextflag) < 0)
      {
        mi_print_error(info->s, HA_ERR_CRASHED);
@@ -97,6 +98,7 @@
 #endif
 #ifdef HAVE_GIST_KEYS
    case HA_KEY_ALG_GIST_RSTAR:
+     DBUG_PRINT("info", ("Will call gist_find_first"));
      if (gist_find_first(info,inx,key_buff,use_key_length,nextflag) < 0)
      {
        mi_print_error(info->s, HA_ERR_CRASHED);
@@ -108,6 +110,7 @@
      }
      break;
    case HA_KEY_ALG_GIST_RGUT83:
+     DBUG_PRINT("info", ("Will call gist_find_first"));
      if (gist_find_first(info,inx,key_buff,use_key_length,nextflag) < 0)
      {
        mi_print_error(info->s, HA_ERR_CRASHED);
@@ -121,6 +124,7 @@
 #endif
    case HA_KEY_ALG_BTREE:
    default:
+     DBUG_PRINT("info", ("Btree"));
      if (!_mi_search(info, keyinfo, key_buff, use_key_length,
                      myisam_read_vec[search_flag], info->s->state.key_root[
                           inx]))
      {

=== modified file 'storage/myisam/mi_rnext.c'
--- storage/myisam/mi_rnext.c   2012-08-18 05:37:44 +0000
+++ storage/myisam/mi_rnext.c   2012-08-18 11:29:56 +0000
@@ -47,22 +47,27 @@
    changed=_mi_test_if_changed(info);
    if (!flag)
    {
+     DBUG_PRINT("info", ("Read first"));
      switch(info->s->keyinfo[inx].key_alg){
 #ifdef HAVE_RTREE_KEYS
      case HA_KEY_ALG_RTREE:
+       DBUG_PRINT("info", ("Rtree"));
        error=rtree_get_first(info,inx,info->lastkey_length);
        break;
 #endif
 #ifdef HAVE_GIST_KEYS
      case HA_KEY_ALG_GIST_RSTAR:
+       DBUG_PRINT("info", ("Will call gist_get_first"));
        error=gist_get_first(info,inx,info->lastkey_length);
        break;
      case HA_KEY_ALG_GIST_RGUT83:
+       DBUG_PRINT("info", ("Will call gist_get_first"));
        error=gist_get_first(info,inx,info->lastkey_length);
        break;
 #endif
      case HA_KEY_ALG_BTREE:
      default:
+       DBUG_PRINT("info", ("Btree"));
        error=_mi_search_first(info,info->s->keyinfo+inx,
                               info->s->state.key_root[inx]);
        break;
@@ -84,9 +89,11 @@
    }
    else
    {
+     DBUG_PRINT("info", ("Read next"));
      switch (info->s->keyinfo[inx].key_alg) {
 #ifdef HAVE_RTREE_KEYS
      case HA_KEY_ALG_RTREE:
+       DBUG_PRINT("info", ("Rtree"));
        /*
         Note that rtree doesn't support that the table
         may be changed since last call, so we do need
@@ -100,18 +107,21 @@
```

```
        /*
         Note (from rtree?)
        */
+       DBUG_PRINT("info", ("Will call gist_get_next"));
        error= gist_get_next(info,inx,info->lastkey_length);
        break;
      case HA_KEY_ALG_GIST_RGUT83:
        /*
         Note (from rtree?)
        */
+       DBUG_PRINT("info", ("Will call gist_get_next"));
        error= gist_get_next(info,inx,info->lastkey_length);
        break;

 #endif
      case HA_KEY_ALG_BTREE:
      default:
+       DBUG_PRINT("info", ("Btree"));
        if (!changed)
         error= _mi_search_next(info,info->s->keyinfo+inx,info->lastkey,
                                info->lastkey_length,flag,

=== modified file 'storage/myisam/mi_rnext_same.c'
--- storage/myisam/mi_rnext_same.c      2012-08-18 05:37:44 +0000
+++ storage/myisam/mi_rnext_same.c      2012-08-18 11:29:56 +0000
@@ -47,6 +47,7 @@
    {
 #ifdef HAVE_RTREE_KEYS
      case HA_KEY_ALG_RTREE:
+       DBUG_PRINT("info", ("Rtree"));
        if ((error=rtree_find_next(info,inx,
                                  myisam_read_vec[info->last_key_func])))
        {
@@ -59,6 +60,7 @@
 #endif
 #ifdef HAVE_GIST_KEYS
      case HA_KEY_ALG_GIST_RSTAR:
+       DBUG_PRINT("info", ("Will call gist_find_next"));
        if ((error=gist_find_next(info,inx,
                                  myisam_read_vec[info->last_key_func])))
        {
@@ -69,6 +71,7 @@
        }
        break;
      case HA_KEY_ALG_GIST_RGUT83:
+       DBUG_PRINT("info", ("gist_find_next"));
        if ((error=gist_find_next(info,inx,
                                  myisam_read_vec[info->last_key_func])))
        {
@@ -81,6 +84,7 @@
 #endif
      case HA_KEY_ALG_BTREE:
      default:
+       DBUG_PRINT("info", ("Btree"));
        if (!(info->update & HA_STATE_RNEXT_SAME))
        {
          /* First rnext_same; Store old key */

=== modified file 'storage/myisam/mi_search.c'
--- storage/myisam/mi_search.c  2012-01-13 14:50:02 +0000
+++ storage/myisam/mi_search.c  2012-08-18 11:29:56 +0000
@@ -99,6 +99,7 @@

   if (flag)
   {
+    DBUG_PRINT("info", ("flag from bin_search"));
     if ((error=_mi_search(info,keyinfo,key,key_len,nextflag,
                           _mi_kpos(nod_flag,keypos))) <= 0)
       DBUG_RETURN(error);
@@ -114,6 +115,7 @@
   }
   else
   {
+    DBUG_PRINT("info", ("no flag from bin_search"));
     if ((nextflag & SEARCH_FIND) && nod_flag &&
        ((keyinfo->flag & (HA_NOSAME | HA_NULL_PART)) != HA_NOSAME ||
         key_len != USE_WHOLE_KEY))

=== modified file 'storage/myisam/mi_write.c'
--- storage/myisam/mi_write.c   2012-01-13 14:50:02 +0000
+++ storage/myisam/mi_write.c   2012-08-18 11:29:56 +0000
@@ -118,6 +118,7 @@
       }
       else
       {
+        DBUG_PRINT("info",("Will call ck_insert"));
         if (share->keyinfo[i].ck_insert(info,i,buff,
                         _mi_make_key(info,i,buff,record,filepos)))
         {

=== modified file 'storage/myisam/myisamdef.h'
--- storage/myisam/myisamdef.h  2012-08-18 05:37:44 +0000
+++ storage/myisam/myisamdef.h  2012-08-18 11:29:56 +0000
@@ -303,6 +303,7 @@
   uchar *rtree_recursion_state;         /* For RTREE */
   uchar *gist_recursion_state;          /* For GIST  */
   int rtree_recursion_depth;
+  int gist_recursion_depth;
 };

 #define USE_WHOLE_KEY   HA_MAX_KEY_BUFF*2 /* Use whole key in _mi_search()
      */

=== modified file 'storage/myisam/rt_index.c'
--- storage/myisam/rt_index.c   2012-06-04 15:26:11 +0000
+++ storage/myisam/rt_index.c   2012-08-18 11:29:56 +0000
@@ -21,23 +21,9 @@
 #include "rt_key.h"
 #include "rt_mbr.h"

-#define REINSERT_BUFFER_INC 10
 #define PICK_BY_AREA
 /*#define PICK_BY_PERIMETER*/

-typedef struct st_page_level
-{
-  uint level;
-  my_off_t offs;
-} stPageLevel;
-
-typedef struct st_page_list
-{
-  ulong n_pages;
-  ulong m_pages;
-  stPageLevel *pages;
-} stPageList;
-
```

```

 /*
    Find next key in r-tree according to search_flag recursively
@@ -61,6 +47,8 @@
   uchar *page_buf;
   int k_len;
   uint *saved_key = (uint*) (info->rtree_recursion_state) + level;
+
+  DBUG_PRINT("info", ("rtree_find_req: Level %d", level));

   if (!(page_buf = (uchar*)my_alloca((uint)keyinfo->block_length)))
   {
@@ -399,6 +387,10 @@
   my_off_t root;
   MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;

+  DBUG_PRINT("info", ("rtree_find_first"));
+
+  DBUG_PRINT("info", ("rtree_get_first"));
+
   if ((root = info->s->state.key_root[keynr]) == HA_OFFSET_ERROR)
   {
     my_errno= HA_ERR_END_OF_FILE;
@@ -426,6 +418,8 @@
   my_off_t root= info->s->state.key_root[keynr];
   MI_KEYDEF *keyinfo = info->s->keyinfo + keynr;

+  DBUG_PRINT("info", ("rtree_get_next"));
+
   if (root == HA_OFFSET_ERROR)
   {
     my_errno= HA_ERR_END_OF_FILE;
@@ -463,7 +457,7 @@
 */

 #ifdef PICK_BY_PERIMETER
-static uchar *rtree_pick_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+uchar *rtree_pick_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
                              uint key_length, uchar *page_buf, uint nod_flag)
 {
   double increase;
@@ -496,7 +490,7 @@
 #endif /*PICK_BY_PERIMETER*/

 #ifdef PICK_BY_AREA
-static uchar *rtree_pick_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+uchar *rtree_pick_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
                              uint key_length, uchar *page_buf, uint nod_flag)
 {
   double increase;
@@ -728,7 +722,7 @@
     0  OK
 */

-static int rtree_fill_reinsert_list(stPageList *ReinsertList, my_off_t page,
+int rtree_fill_reinsert_list(stPageList *ReinsertList, my_off_t page,
                                     int level)
 {
   DBUG_ENTER("rtree_fill_reinsert_list");

=== modified file 'storage/myisam/rt_index.h'
--- storage/myisam/rt_index.h   2006-12-31 00:32:21 +0000
+++ storage/myisam/rt_index.h   2012-08-18 11:29:56 +0000
@@ -16,6 +16,8 @@
```

```
 #ifndef _rt_index_h
 #define _rt_index_h

+#include "sp_reinsert.h"
+
 #ifdef HAVE_RTREE_KEYS

 #define rt_PAGE_FIRST_KEY(page, nod_flag) (page + 2 + nod_flag)
@@ -41,5 +43,14 @@
 int rtree_split_page(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *page, uchar *
     key,
                     uint key_length, my_off_t *new_page_offs);

+
+
+uchar *rtree_pick_key(MI_INFO *info, MI_KEYDEF *keyinfo, uchar *key,
+                      uint key_length, uchar *page_buf, uint nod_flag);
+
+int rtree_fill_reinsert_list(stPageList *ReinsertList, my_off_t page, int
+    level);
+
+
+
 #endif /*HAVE_RTREE_KEYS*/
 #endif /* _rt_index_h */

=== modified file 'storage/myisam/rt_mbr.h'
--- storage/myisam/rt_mbr.h     2006-12-31 00:32:21 +0000
+++ storage/myisam/rt_mbr.h     2012-08-18 11:29:56 +0000
@@ -32,5 +32,9 @@
                                 uint key_length, double *ab_perim);
 int rtree_page_mbr(MI_INFO *info, HA_KEYSEG *keyseg, uchar *page_buf,
                    uchar* c, uint key_length);
+
+int rtree_key_cmp(HA_KEYSEG *keyseg, uchar *b, uchar *a, uint key_length,
+                  uint nextflag);
+
 #endif /*HAVE_RTREE_KEYS*/
 #endif /* _rt_mbr_h */

=== added file 'storage/myisam/sp_reinsert.h'
--- storage/myisam/sp_reinsert.h        1970-01-01 00:00:00 +0000
+++ storage/myisam/sp_reinsert.h        2012-08-18 11:29:56 +0000
@@ -0,0 +1,36 @@
+/* Copyright (C) 2012 Monty Program AB & Vangelis Katsikaros
+
+   This program is free software; you can redistribute it and/or modify
+   it under the terms of the GNU General Public License as published by
+   the Free Software Foundation; version 2 of the License.
+
+   This program is distributed in the hope that it will be useful,
+   but WITHOUT ANY WARRANTY; without even the implied warranty of
+   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+   GNU General Public License for more details.
+
+   You should have received a copy of the GNU General Public License
+   along with this program; if not, write to the Free Software
+   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
+    */
+
+#ifndef _SP_REINSERT_H
+#define _SP_REINSERT_H
+
+#define REINSERT_BUFFER_INC 10
```

```
+
+typedef struct st_page_level
+{
+  uint level;
+  my_off_t offs;
+} stPageLevel;
+
+typedef struct st_page_list
+{
+  ulong n_pages;
+  ulong m_pages;
+  stPageLevel *pages;
+} stPageList;
+
+
+
+#endif /* _SP_REINSERT_H */
```

Index

Glossary

API Application Programming Interface.

B$^+$-tree B-tree variant. By Knuth [38]. Name first used in [13, p. 129].

B-tree Binary Search Tree. By Bayer and McCreight [6].

CAD Computer-Aided Design.

CAM Computer-Aided Manufacturing.

DBMS DataBase Management System.

DICOM Digital Imaging and Communications in Medicine.

GIS Geographic Information System.

GiST Generalized Search Tree. By Hellerstein, Naughton and Pfeffer [29].

IDC International Data Corporation.

ISO International Organization for Standardization.

K-D-B-tree B-tree multidimensional variant. By Robinson [100].

MBR Minimum Bounding Rectangle.

MySQL An RDBMS. Known as "The world's most popular open source database".

OGC Open Geospatial Consortium.

OSM Open Street Maps.

R$^+$-tree R-tree variant. By Sellis, Roussopoulos and Faloutsos [103].

R-tree Rectangular-based B-tree. By Guttman [28] .

RDBMS Relational DataBase Management System.

UML Unified Modeling Language.

VLSI Very-Large-Scale Integration.

W3C World Wide Web Consortium.

List of Figures

List of Algorithms

List of Tables

Bibliography

[1] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. The lowell database research self-assessment. *Commun. ACM*, 48:111–118, May 2005.

[2] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, Anhai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The claremont report on database research. *Commun. ACM*, 52:56–65, June 2009.

[3] Aitchison Alastair. *Beginning Spatial with SQL Server 2008.* Apress, Berkely, CA, USA, 1 edition, 2009.

[4] Jochen Albrecht. Universal analytical gis operations- a task-oriented systematization of data structure-independent gis functionality. pages 577–591, 1996.

[5] Chuan-Heng Ang and T. C. Tan. New linear node splitting algorithm for r-trees. In *Proceedings of the 5th International Symposium on Advances in*

Spatial Databases, SSD '97, pages 339–349, London, UK, 1997. Springer-Verlag.

[6] Rudolf Bayer and E. M. Mccreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pages 322–331. ACM, 1990.

[8] Elisa Bertino, Barbara Catania, and Luca Chiesa. Definition and analysis of index organizations for object-oriented database systems. *Inf. Syst.*, 23:65–108, April 1998.

[9] Bison. Bison 2.6.2. http://www.gnu.org/software/bison/manual/bison.html, 2010.

[10] Calpont. What is infinidb? http://infinidb.org/resources/what-is-infinidb, 2010.

[11] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, 2010.

[12] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[13] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[14] C. J. Date. *An Introduction to Database Systems, Seventh Edition*. Addison Wesley, 7 edition, July 1999.

[15] Edsger W. Dijkstra. On the nature of computing science. 1984.

[16] Etsy. The etsy shard architecture: Starts with s and ends with hard. http://www.percona.com/live/mysql-conference-2012/sessions/etsy-shard-architecture-starts-s-and-ends-hard, 2011.

[17] Facebook. Keeping up. http://blog.facebook.com/blog.php?post=7899307130, 2012.

[18] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *PODS*, pages 247–252, 1989.

[19] Flickr. The great map update of 2012. http://code.flickr.com/blog/2012/06/29/the-great-map-update-of-2012/, 2010.

[20] Flickr. Using, abusing and scaling mysql at flickr. http://code.flickr.com/blog/2010/02/08/using-abusing-and-scaling-mysql-at-flickr/, 2010.

[21] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[22] John F. Gantz, David Reinsel, Christopeher Chute, Wolfgang Schlichting, Stephen Minton, Anna Toncheva, and Alex Manfrediz. The expanding digital universe: An updated forecast of worldwide information growth through 2011. Technical report, IDC Information and Data, 2008.

[23] Yván J. García, Mario A. Lopez, and Scott T. Leutenegger. On optimal node splitting for r-trees. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 334–344, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[24] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book.* Prentice Hall, October 2001.

[25] Sergei Golubchik and Andrew Hutchings. *MySQL 5.1 Plugin Development.* Packt Publishing, 2010.

[26] Google. Introduction of usage limits to the maps api! http://googlegeodevelopers.blogspot.gr/2011/10/introduction-of-usage-limits-to-maps.html, 2010.

[27] Google. Mysql tools released by google. http://code.google.com/p/google-mysql-tools/, 2010.

[28] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[29] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *IN PROC. 21 ST INTERNATIONAL CONFERENCE ON VLDB*, pages 562–573, 1995.

[30] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. Technical Report 1274, University of Wisconsin at Madiso, 07 1995.

[31] David Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. *Mathematische Annalen*, 38:459–468, 1891.

[32] ISO. About iso. http://www.iso.org/iso/about.htm, 2010.

[33] ISO. Discover iso: Who standards benefit. http://www.iso.org/iso/about/discovers-iso_who-standards-benefits.htm, 2010.

[34] ISO. Iso in figures for the year 2009. http://www.iso.org/iso/about/iso_in_figures.htm, 2010.

[35] ISO. Iso members. http://www.iso.org/iso/about/iso_members.htm, 2010.

[36] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. pages 500–509, 1994.

[37] Vangelis Katsikaros. Special q&a with monty widenius. https://www.linux.com/news/enterprise/biz-enterprise/544438-special-qaa-with-monty-widenius, 2010.

[38] Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1973.

[39] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[40] Scott T Leutenegger. Scott t leutenegger's publication page. http://web.cs.du.edu/~leut/pubs.html, 2010.

[41] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

[42] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*. Springer, 1 edition, 2005.

[43] MariaDB. Compiling mariadb from source. http://kb.askmonty.org/en/compiling-mariadb-from-source/, 2010.

[44] MariaDB. Downloads source, binaries, and packages. http://downloads.mariadb.org/mariadb/, 2010.

[45] MariaDB. Getting the mariadb source code. http://kb.askmonty.org/en/source-getting-the-mariadb-source-code/, 2010.

[46] MariaDB. Maria in launchpad. https://launchpad.net/maria, 2010.

[47] MariaDB. Mariadb. http://kb.askmonty.org/en/mariadb/, 2010.

[48] Yoshinori Matsunobu. Using mysql as a nosql - a story for exceeding 750,000 qps on a commodity server. http://yoshinorimatsunobu.blogspot.com/2010/10/using-mysql-as-nosql-story-for.html, 2010.

[49] Microsoft. Delivering location intelligence with spatial data. Technical report, Microsoft, 2010.

[50] Microsoft. Information schema views (transact-sql). http://msdn.microsoft.com/en-us/library/ms186778.aspx, 2010.

[51] Microsoft. Ogc methods on geometry instances. http://msdn.microsoft.com/en-us/library/bb933960.aspx, 2010.

[52] Microsoft. Rebranding microsoft virtual earth to.... http://blogs.msdn.com/b/virtualearth/archive/2009/05/28/rebranding-microsoft-virtual-earth-to.aspx, 2010.

[53] Microsoft. Types of spatial data. http://msdn.microsoft.com/en-us/library/bb964711.aspx, 2010.

[54] MySQL. 13.1.13. create index syntax. http://dev.mysql.com/doc/refman/5.1/en/create-index.html, 2010.

[55] MySQL. 13.1.17. create table syntax. http://dev.mysql.com/doc/refman/5.1/en/create-table.html, 2010.

[56] MySQL. Gis functions. http://forge.mysql.com/wiki/GIS_Functions, 2010.

[57] MySQL. Mysql 5.0 reference manual :: 11.16 spatial extensions. http://dev.mysql.com/doc/refman/5.0/en/spatial-extensions.html, 2010.

[58] MySQL. Mysql 5.1 reference manual :: 5.1.4. server system variables. http://dev.mysql.com/doc/refman/5.1/en/archive-storage-engine.html, 2010.

[59] MySQL. Mysql 5.5 reference manual. http://dev.mysql.com/doc/refman/5.5/, 2010.

[60] MySQL. Mysql 5.5 reference manual :: 14. storage engines. http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html, 2010.

[61] MySQL. Mysql 5.5 reference manual :: 14.3. the innodb storage engine. http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html, 2010.

[62] MySQL. Mysql 5.5 reference manual :: 14.5. the myisam storage engine. http://dev.mysql.com/doc/refman/5.5/en/myisam-storage-engine.html, 2010.

[63] MySQL. Mysql 5.5 reference manual :: 14.6. the memory storage engine. http://dev.mysql.com/doc/refman/5.5/en/memory-storage-engine.html, 2010.

[64] MySQL. Mysql 5.5 reference manual :: 14.8. the archive storage engine. http://dev.mysql.com/doc/refman/5.5/en/archive-storage-engine.html, 2010.

[65] MySQL. Mysql 5.5 reference manual :: 22.1.2. the mysql test suite. http://dev.mysql.com/doc/refman/5.1/en/mysql-test-suite.html, 2010.

[66] MySQL. Mysql 5.5 reference manual :: 22.2. the mysql plugin api. http://dev.mysql.com/doc/refman/5.1/en/plugin-api.html, 2010.

[67] MySQL. Mysql customers by industry. http://www.mysql.com/customers/industry/, 2010.

[68] MySQL. Sun to acquire mysql. http://www.mysql.com/news-and-events/sun-to-acquire-mysql.html, 2010.

[69] MySQL. Virgin mobile implements mysql enterprise. http://www.mysql.com/news-and-events/generate-article.php?id=2008_02, 2010.

[70] P. Naur and B. Randell. Software engineering techniques: Report on a conference sponsored by the nato science committee. Technical report, NATO, Brüssel, 1969.

[71] NIST. Nist: Strengthening u.s. innovation and industrial competitiveness. http://www.nist.gov/public_affairs/factsheet/strengthen_innovation_competitiveness.cfm, 2010.

[72] Shlomi Noach. Sphinx & mysql: facts and misconceptions. http://code.openark.org/blog/mysql/sphinx-mysql-facts-and-misconceptions, 2010.

[73] OGC. About ogc. http://www.opengeospatial.org/ogc, 2010.

[74] OGC. Faqs - ogc process. http://www.opengeospatial.org/ogc/faq/process, 2010.

[75] OGC. Implementations by specification. http://www.opengeospatial.org/resource/products/byspec/?specid=149, 2010.

[76] OGC. Ogc kml. http://www.opengeospatial.org/standards/kml, 2010.

[77] OGC. Ogc members. http://www.opengeospatial.org/ogc/members, 2010.

[78] OGC. Opengis implementation standard for geographic information - simple feature access - part 1: Common architecture. Technical report, Open Geospatial Consortium Inc., 2010.

[79] OGC. Opengis implementation standard for geographic information - simple feature access - part 2: Sql option. Technical report, Open Geospatial Consortium Inc., 2010.

[80] OGC. Simple features swg. http://www.opengeospatial.org/projects/groups/sfswg, 2010.

[81] Oracle. Editors choice awards 2010: Delivering innovation. http://www.oracle.com/technetwork/issue-archive/2010/10-nov/o60eca-176293.html, 2010.

[82] Oracle. Mysql: The dolphins leap again. http://www.opensourceday.pl/download,70,pl.html, 2010.

[83] Oracle. Oracle buys sun. http://www.oracle.com/us/corporate/press/018363, 2010.

[84] Oracle. Oracle spatial 10g. Technical report, Oracle, 2010.

[85] Oracle. Oracle spatial and locator features. http://www.oracle.com/technetwork/database/enterprise-edition/spatial-locator-features-100445.html, 2010.

[86] OSM. Component overview. http://wiki.openstreetmap.org/wiki/Component_Overview, 2010.

[87] OSM. Apple attributes osm in iphoto. https://twitter.com/openstreetmap/status/198101512201834497, 2012.

[88] Dimitris Papadias, Yannis Theodoridis, Timos Sellis, and Max J. Egenhofer. Topological relations in the world of minimum bounding rectangles: A study with r-trees. pages 92–103, 1995.

[89] Michael P. Peterson. *International Perspectives on Maps and the Internet - Lecture Notes in Geoinformation and Cartography*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[90] PostGIS. Eu joint research centre. http://postgis.refractions.net/documentation/casestudies/jrc/, 2010.

[91] PostGIS. Institut géographique national, france. http://postgis.refractions.net/documentation/casestudies/ign/, 2010.

[92] PostGIS. What is postgis? http://postgis.refractions.net/, 2010.

[93] PostgreSQL. Postgresql 9.0: Geometric types. http://www.postgresql.org/docs/9.0/interactive/datatype-geometric.html, 2010.

[94] PostgreSQL. Postgresql 9.0: Implementation. http://www.postgresql.org/docs/9.0/interactive/gist-implementation.html, 2010.

[95] PostgreSQL. Postgresql 9.0: The information schema. http://www.postgresql.org/docs/9.0/static/information-schema.html, 2010.

[96] PostgreSQL. gist readme. http://archives.postgresql.org/pgsql-hackers/2011-01/msg02331.php, 2011.

[97] PostgreSQL. postgresql.git/history - src/backend/access/gist/readme. http://git.postgresql.org/gitweb?p=postgresql.git;a=history;f=src/backend/access/gist/README;h=6c90e508bfe6cab2304c9ce5bc24f54dca6ca20e;hb=refs/heads/REL9_0_STABLE, 2011.

[98] Monty Program. About us - monty program. http://montyprogram.com/about/, 2010.

[99] Raghu Ramakrishnan, Johannes Gehrke, Raghu Ramakrishnan, and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 3 edition, August 2002.

[100] John T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, SIGMOD '81, pages 10–18, New York, NY, USA, 1981. ACM.

[101] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. *SIGMOD Rec.*, 14:17–31, May 1985.

[102] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz, and Derek J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and Load-Balancing*. O'Reilly Media, 3 edition, 2012.

[103] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. pages 507–518, 1987.

[104] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. Multidimensional access methods: Trees have grown everywhere. In *VLDB*, pages 13–14, 1997.

[105] Mehdi Sharifzadeh and Cyrus Shahabi. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. *Proc. VLDB Endow.*, 3:1231–1242, September 2010.

[106] Shashi Shekhar and Sanjay Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.

[107] Shashi Shekhar and Hui Xiong, editors. *Encyclopedia of GIS*. Springer, 2008.

[108] SkySQL. Deutsche telekom ag. http://www.skysql.com/node/260, 2010.

[109] SQLite. Sqlite r*. http://www.sqlite.org/rtree.html, 2010.

[110] Ssolbergj. Svg map of europe. borders of nation states. http://commons.wikimedia.org/wiki/File:Location_European_nation_states.svg, 2012.

[111] Michael Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the Second International Conference on Data Engineering*, pages 262–269, Washington, DC, USA, 1986. IEEE Computer Society.

[112] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of postgres. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1990.

[113] Twitter. Big and small data at twitter: Mysql ce 2011. http://nosql.mypopescu.com/post/4687379038/big-and-small-data-at-twitter-mysql-ce-2011, 2012.

[114] W3C. Current members. http://www.w3.org/Consortium/Member/List, 2010.

[115] W3C. Member statistics. http://www.w3.org/Consortium/Member/stats.html, 2010.

[116] W3C. Standards faq. http://www.w3.org/standards/faq.html, 2010.

[117] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2010.

[118] Tian Xia and Donghui Zhang. Improving the r*-tree with outlier handling techniques, 2005.

[119] Yahoo. Mysql high availability at yahoo! http://developer.yahoo.com/blogs/ydn/posts/2010/08/mysql_high_availability/, 2010.

[120] Peter Zaitsev. Talking mysql to sphinx. http://www.mysqlperformanceblog.com/2009/04/19/talking-mysql-to-sphinx/, 2010.

[121] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.