The Price of Generality in Spatial Indexing

Bogdan Simion, Daniel N. Ilha, Angela Demke Brown, Ryan Johnson
Department of Computer Science, University of Toronto
{bogdan, demke, ryan.johnson} @cs.toronto.edu, daniel.n.ilha@gmail.com

ABSTRACT

Efficient indexing can significantly speed up the processing of large volumes of spatial data in many BigData applications. Many new emerging spatial applications (e.g., biomedical imaging, genome analysis, etc.) have varying indexing requirements, thus, a unified indexing infrastructure for implementing new indexing schemes without requiring knowledge of database internals is beneficial. However, designing a generic indexing framework is a challenging task. We study the issues with general indexing schemes, such as the GiST (used in PostGIS) and expose the tradeoff between generality and performance, showing that generality can be severely detrimental to performance if the abstractions are not carefully designed. Our experiments indicate that the GiST framework, as implemented in PostgreSQL/PostGIS, performs 4.5-6x slower for filtering records through the index, compared to a custom R-tree implementation. We also isolate the GiST-specific overhead by implementing the framework outside the DBMS, showing that the GiST-based R-tree is up to 2x slower than the raw R-tree algorithm that it uses internally. We conclude that although a generic framework for a wide range of spatial BigData application domains is desirable, implementers of new frameworks need to be careful in designing the abstractions to avoid paying a hefty performance penalty.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Performance, Measurement, Experimentation, Algorithms

Keywords

Spatial databases, BigData, Spatial indexing, General Indexing Framework, GiST, R-tree.

1. INTRODUCTION

Spatial BigData processing has gained increased importance in recent years, due to emerging new applications that operate on huge volumes of data, such as medical imaging, genome sequencing, land information surveys, or environmental impact assessment. With the growth of spatial data, a key performance factor in spatial processing is the use of a spatial indexing mechanism. With BigData, the amount of useful data to be retrieved from a massive dataset is generally fairly small. As a result, an efficient index is crucial in fast data retrieval, which means close attention must be paid to index design. Given that many scientists in various fields are rolling out custom data processing solutions for their BigData needs, a generic framework for indexing multiple data domains, such as biomedical images, genome sequence data, audio/video data, fingerprints, biomolecular data, etc., seems like a promising avenue of exploration. However, designing a framework to support indexes for any of the emerging BigData applications is not an easy task.

Traditional database management systems (DBMS) are specifically designed for storing and processing large volumes of information, and making use of efficient indexing to query huge amounts of data very fast. As a result, they are a suitable candidate for building spatial analysis applications. Due to the increasing variety of emerging application domains, database designers are faced with implementing new efficient index structures for each of these domains, and integrating them into a rather large DBMS codebase. Recognizing this problem, researchers have attempted to design more generalized indexing frameworks, such as the Generalized Index Search Tree (GiST) [5], that facilitates an easily pluggable index implementation for any new type of data domain. PostgreSQL/PostGIS is one of the early adopters of the GiST as a generic framework for building R-trees and B-trees.

While designing a framework for building and accessing a spatial index in a generic way simplifies the programmer task when adding new index implementations, the abstraction has two major problems to address. First, generalizing the operations for any conceivable type of index structure and its possible variations is not trivial. For example, although the GiST framework supports the use of B+trees and R-trees, it lacks flexibility in supporting variations like R*-trees, or bulk loading, which have desirable properties for speeding up searches in many spatial scenarios.

The second problem is that the abstraction involved in designing a generic one-size-fits-all framework for implementing indexes, can pose a considerable overhead that ends up hurting overall performance of spatial queries. If custom implementations of domain-specific indexes outperform by a wide margin their counterparts implemented through a general indexing framework, then the design needs to be re-evaluated.

In this work, we illustrate the pitfalls of a generic framework (GiST) for indexing large volumes of spatial data in a state-of-the-art spatial DBMS (PostGIS) and show there is a tradeoff between performance and generality that needs to be carefully considered when designing a new generic indexing framework.

Our main contributions are three-fold:

a. We conduct an analysis of PostgreSQL/PostGIS (the only open-source OGC-compliant DBMS), to show where time is spent in processing spatial queries involving spatial indexes. We find that the DBMS imposes a large overhead over a raw R-tree implementation, which is partly due to the rigidity of the GiST framework.

b. We isolate the GiST performance from the DBMS implementation-specific overheads, by implementing an optimized stripped-down version of the GiST, outside of PostgreSQL, using the same insertion-based strategy (NLA) used in PostgreSQL. We compare the performance of the external GiST framework against several raw R-tree implementations, both insertion-based and bulk-loaded. The GiST not only is 1.5-2x slower than its raw NLA strategy counterpart, but also 4.5-6x slower than bulk-loaded R-trees, on all page sizes.

c. We show that although a one-size-fits-all indexing framework is very desirable, the approach of the GiST is rather flawed. We point out that the key problems lie in the restrictive nature of the set of predefined-operations, which draw the abstraction line in the wrong place, causing a negative performance impact.

2. BACKGROUND AND RELATED WORK

2.1 Spatial processing and indexing

Spatial databases employ a two step evaluation mechanism. The first step is *filter*, which returns a superset of the candidate objects satisfying a spatial predicate, by comparing an approximation of actual objects (called the minimum bounding rectangle, or MBR). The second step is *refinement*, in which the actual geometry of the candidate objects are inspected. The filter step tries to eliminate as many objects as possible, since the refinement step uses time-consuming computational geometry algorithms. To avoid evaluating the MBRs for all records, the candidate space can be pruned using a spatial index. The MBR approximations are stored in spatially-aware data-structures such as R-trees, Quadtrees, etc, to only test the MBRs located in the spatial ranges that are relevant to the query.

2.2 R-tree strategies

The R-Tree is a disk-efficient data structure, which stores a directory of entries into its internal nodes (or pages) to index the MBRs. R-trees are similar to B+Trees, but support N-dimensional data, by storing the MBRs of geometries as keys in the internal node entries. This support comes at the cost of page overlap which may degrade search performance (due to multi-path probing) and is dependent on how internal nodes and leaves are grouped. To maximize the tree search performance, many heuristics and variations of the data structure exist that attempt to produce the best grouping of pages and leaves. These are divided into three types:

a. Page-splitting insertion heuristics. These heuristics apply to how nodes are split when a record is to be inserted in a node that is already at maximum capacity. These heuristics play a significant role on minimizing overlap of internal pages, but they tend to be limited on the improvement they can provide due to their susceptibility to insertion order. The *Quadratic* split [4] attempts to reduce internal page area (and thus overlap) by looking to group children that result in the smallest aggregate bounding box. The *New Linear Algorithm (NLA)* [1] attempts to reduce internal page overlap by grouping children in pages that are as far away from each other as possible (directly minimizes overlap).

b. Bulk-loading strategies. When the dataset is available a priori, a bulk-load heuristic can leverage the spatial data distribution to build better packed R-trees. As a result, bulk-loaded R-trees minimize overlap more efficiently than insertion-based heuristics. Bulk-loading is particularly reasonable for applications where the data is either read-only or does not change often. Such methods include the *Hilbert* heuristic [6], which uses the Hilbert space-filling curve and its good locality properties to sort pages according to the Hilbert distance metric and group them. The *Sort-Tile-Recursive (STR)* [7] strategy leverages the properties of *k-d* trees into the R-Tree structure, by sorting the records on each axis and splitting the space along each dimension into stripes of equal record count.

c. R-tree variants. The R*-tree [3] uses a custom split heuristic and a node dissolution policy which gets around the limitations of split heuristics and generates much better results, but has a more expensive insertion operation. Basically, when a node overflows, a portion of the entries are removed and reinserted, resulting in a better tree in terms of reducing node overlap.

2.3 Generic indexing frameworks

With several emerging application domains being brought into the relational database market there is an increasing need for custom indexing data structures for efficient information retrieval. Since implementing custom indexes in a large codebase is an onerous task, some have sought a generic index structure that could be customized to fit any data domain without any changes in (or knowledge of) database internals.

Several researchers looked into how we can generalize the properties of various indexes in such a way to create a unified indexing infrastructure for ordered data domains. A generic indexing framework eliminates the burden on the programmer to know the database internals, requiring only the implementation of a pre-defined interface which abstracts how the index operates. However, the generality of such a framework is not a trivial task. For example, B-trees and R-trees have different properties (large fanout, balanced by nature) from other space-partitioning trees like quad-trees or kd-trees (reduced fanout, often unbalanced). As a result, abstracting the operations on an index must be carefully considered.

An attempt at designing a general indexing framework is the Generalized Index Search Tree (GiST) [5], which aimed to allow an easily pluggable index implementation for any new type of data domain. The paper is mostly theoretical, and the framework only indicates how to integrate B-trees, R-trees and RD-trees, while variations like R*-trees, and bulk-loaded R-trees are unsupported, as are other space-partitioning trees. Aref and Ilyas [2] designed the SP-GiST, an extensible database index for supporting space-partitioning trees, such as quad-trees, kd-trees, tries and their variants. Unfortunately, the SP-GiST is far from generic as well.

Although generalization is not trivial, due to the variety of emerging application domains, a generic framework for indexing data from various origins is worth exploring. PostgreSQL/PostGIS was one of the early adopters of the GiST as a framework for building R-trees. It is also one of the only open-source OGC-compliant DBMS, therefore, we are using it to analyze the performance implications of a generalized indexing framework.

No prior work has looked into the overhead of the GiST or studied the performance impact of its abstractions. We find that when designing a generic framework for building indexes for any Big-Data application domain, the overhead of the abstraction must be considered, since it can cause a major performance hit.

3. THE GIST FRAMEWORK

The GiST framework aims to provide all the basic search tree logic, unifying distinct search structures, such as B+-trees, R-trees or RD-trees. The framework assumes that the canonical image of a database search tree is a balanced tree with high fanout and a given minimum fill factor. They also assume that the nature of any database search tree is that it splits the dataset into partitions, in such a way that each partition has a categorization criterion that holds for all the data in the partition. This criterion becomes the key for the subtree that contains the partition.

The GiST provides a flexible interface that requires the programmer to only implement a set of 6 methods that allow the GiST to behave like any search structure that meets these assumptions. The methods exposed to the user are as follows:

Consistent: Given an entry in the tree $E = (Key, Ptr)$ and a given search predicate P, it returns false if $P \wedge Key$ is guaranteed unsatisfiable, or true otherwise. This allows for multiple paths to be searched, if there is overlap between the keys (e.g., R-trees).

Union: Given a set of entries, returns a predicate that holds for all tuples stored below them. This function is used for splitting the data, such that a criterion holds within each partition.

Compress: Given an entry $E = (Key, Ptr)$ returns an entry (π, Ptr), where π is a compressed representation of Key. For R-trees, the compressed representation is the MBR.

Decompress: Returns the uncompressed version of an entry. For R-trees, this is the identity function.

Penalty: This method determines a domain-specific penalty for inserting an entry $E1$ into the subtree rooted at another entry $E2$. This is used in the split and insertion algorithms.

PickSplit: Given a set S of $M + 1$ entries, this methods splits this set into two sets of entries, S_1 and S_2, where each of them has at least size kM (ensuring the minimum fill factor).

Using solely these methods, the framework implements all the search tree operations, such as insert, search and delete. Therefore, the framework should be able to, in theory, simulate any search tree, as long as the methods are implemented for the specific data domain. Unfortunately, the GiST is restrictive in assuming a traditional insertion-based construction of a search tree. Due to the reinsertion mechanism, R*-trees are not supported, nor are bulk-loading methods because of their divergent bottom-up approach.

4. PROFILING DBMS PERFORMANCE

To analyze the performance of the GiST framework we use a PostgreSQL/PostGIS instance, running a spatial join query involving 6 million polyline records and 6 million polyline records, which uses a spatial index in the query plan. We profile the query execution to obtain a breakdown of execution time. We observe that the filtering step takes 48.41% of the query execution time, while the refinement step accounts for 47.71% (the remaining time is spent in query preparation and returning results). These numbers indicate that the much less computationally intensive filtering step is nonetheless comparable in execution time to the refinement step.

To determine where time is spent in the filtering step, we then use valgrind's callgrind tool, and collect profiling information at the index entry point. We show the breakdown in Figure 1, and identify the performance bottlenecks:

i) TOAST-ing/De-TOAST-ing. PostgreSQL uses a fixed page size of 8 kB, and does not allow tuples to span multiple pages. Therefore, PostgreSQL cannot store very large field values directly. Their solution is to use a technique named TOAST (The Oversized-Attribute Storage Technique), which compresses large field values and/or breaks them up into multiple physical rows. Since a general index may store variable-sized entries, and PostgreSQL includes support for variable length keys, GiST entries are stored in TOAST blocks. PostgreSQL spends roughly 30% of the time reconstructing (de-TOAST-ing) data for each index entry, including internal nodes, in function *pg_detoast_datum_slice*. This is an unnecessary overhead, because we do not need to store either MBRs, or the index

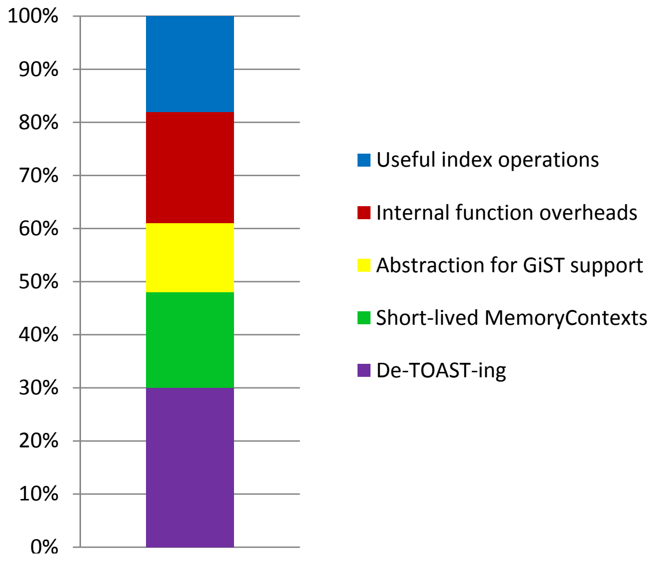

Figure 1: PostgreSQL overheads with the GiST spatial index.

pages themselves (which are both fixed size) in TOAST-ed blocks.

ii) Unnecessary memory management. To detoast index entries PostgreSQL has to allocate short-lived Memory Contexts (memory containers to manage internal allocated memory) for each detoasted key, on each traversal of every internal node. The allocation of Memory Contexts is included in the 30% detoasting overhead, but their *deallocation* accounts for another 18% of the filtering step.

iii) Abstraction overhead. PostgreSQL implements R-tree operations based on the GiST's 6-method abstraction. However, some of these methods are not really necessary for all the index structures supported, which can cause unnecessary function call overhead. For example, GiST's $Decompress$ operation is the identity function for R-trees, and has no effect in the index search, yet must be implemented to support the framework functionality. The overhead of this function is 13% of the filter step. Additionally, the way the abstraction operates restricts some compiler optimizations.

iv) Internal function overheads. Some of the functions implementing the operations described by the GiST account for a considerable overhead. For example, function $gistindex\_keytest$ accounts for 25% of the total filtering time, even though its internal call to $box2df\_overlaps$, which does the actual overlap check for MBRs, only takes ~3% of the time.

We conclude that 82% of the time spent in the index, is wasted in unnecessary overheads, either due to internal DBMS implementation choices, or the GiST abstraction, while only 18% of the time is spent traversing the tree and checking MBRs for overlap.

5. EVALUATING THE PERFORMANCE GAP

In this section, we evaluate the performance of the GiST framework. We show the performance penalty of the PostgreSQL DBMS, then test the GiST framework in isolation (stripped out of the DBMS), comparing it to several raw R-tree algorithms.

In our experiments, we use an Intel Core i7-2600 at 3.40GHz, 8MB cache, and 8GB RAM, running Ubuntu Linux 12.04 LTS, 64-bit 3.2.0 kernel, and gcc-4.6. We use a PostgreSQL 9.0.7 instance, with the PostGIS spatial extension. For all experiments, we use the TIGER dataset [8] for the state of Texas. The largest table contains roughly 6 million polylines, totalling a size of ~2GB, (including the spatial index). For the raw R-tree testing and the optimized GiST, we build the corresponding spatial indexes and then load the

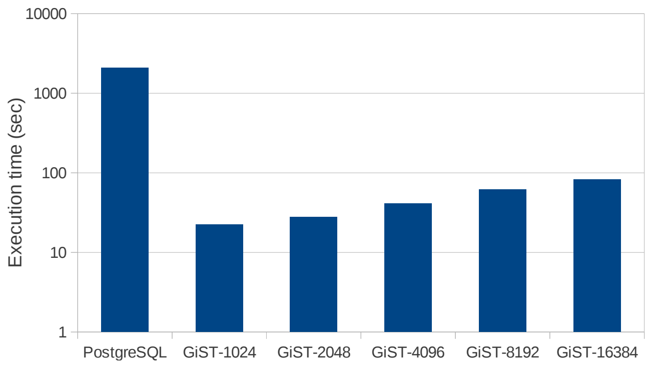

Figure 2: Comparison of PostgreSQL-based GiST with a stripped-down optimized GiST outside the DBMS engine. The Y-axis has logarithmic scale due to the large range of values.

indexes in memory. Since indexes are always considerably smaller than the data, they are generally memory-resident.

We first compare PostgreSQL to an optimized stripped-down GiST framework outside of the DBMS engine, followed by a comparison of the optimized GiST with several R-tree strategies, on a common spatial join query. A tradeoff exists in choosing the page size. Larger pages imply a lower tree height, which means probing the tree will take less time to get to a leaf. On the other hand, the linear traversal of the list of entries in internal nodes favors smaller pages. In our optimized GiST and raw R-trees, we vary the page size to determine its performance impact. PostgreSQL operates with a fixed page size of 8kB.

5.1 PostgreSQL vs GiST

To isolate the DBMS implementation-specific overhead, we implement the GiST framework outside of PostgreSQL. We implement the same NLA insertion-based R-tree algorithm that PostgreSQL uses for its GiST-based R-tree. We observe that in the absence of DBMS-specific abstractions or design choices (e.g., unnecessary de-TOAST-ing operations, transient Memory Context-related overheads, and other DBMS internal overheads), the stripped-down GiST-based R-tree performs much better than its PostgreSQL counterpart, as shown in Figure 2. The filtering step of a large spatial join on two relations of 6 million polylines each, finishes in 61.6 seconds for the stripped-down GiST, compared to almost 20 minutes for a PostgreSQL instance, amounting to a ~20x performance improvement, at the same 8K page size. The performance improvement of our flexible GiST implementation grows to 65x if we reduce the page size to 1kB. We observe that flexibility in selecting the page size to allow the index to be tuned for particular workloads would be desirable, since different queries may achieve their best performance at different page sizes. Unfortunately, PostgreSQL internally uses a fixed page size for operating with tables and indexes in a unified way, and does not allow for this flexibility.

5.2 GiST overheads over raw R-trees

In this section, we analyze the performance of raw R-tree implementations to determine the performance penalty of the GiST abstraction. We compare several strategies for R-trees, including bulk-loaded (Hilbert and STR), insertion-based (quadratic and NLA), and finally the optimized GiST which uses NLA under the hood. We show in figure 3 that when building the tree, the bulk-loading strategies are best, with NLA performing similarly at lower page sizes. We observe that bulk-loading algorithms are impacted less by the page size compared to insertion-based ones. The GiST, although using NLA under the hood, experiences some overhead compared to the raw NLA, because of the extra abstraction.

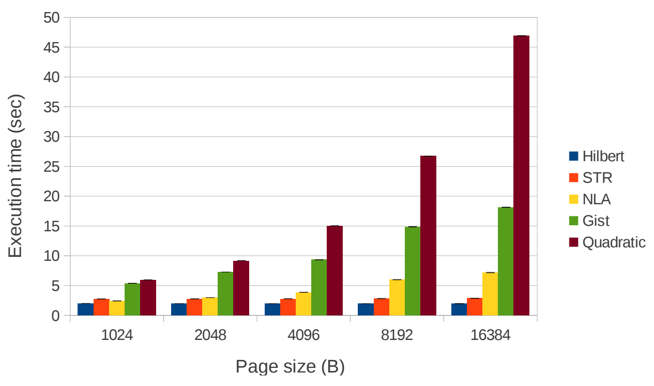

Figure 3: Index build time for GiST vs. raw R-tree algorithms.

We first examine the index build time, since one of the defining characteristics of BigData is its "velocity" (i.e., the rate at which new data is being generated); for high-velocity applications the time to build the index is important. In many spatial cases (e.g., maps, land information), spatial data is not frequently updated, so bulk-loaders have a clear edge because they have all the information up-front and output better R-trees with minimized overlap. However, in the case of frequently updated spatial data (coordinates from GPS devices or smartphones, etc.), the case for bulk-loading becomes less obvious. While we have not evaluated such cases, we note that the initial index construction is fast with bulk-loading (e.g., Hilbert bulk-loading is over 3x faster than the NLA insertion-based strategy used internally by Postgres at the 8k page size). As new data arrives, the index can be updated using insertion-based strategies. Should the index become sub-optimal over time, there is a minimal performance cost for periodically recreating the index using bulk-loading.

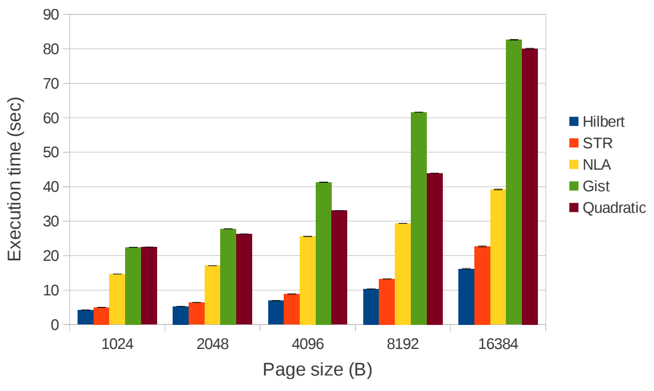

Figure 4: Spatial filtering for raw R-tree algorithms vs GiST.

The build strategy has a big impact on the performance of the R-tree for spatial filtering. Figure 4 shows the performance of the raw R-tree strategies and GiST on a full join filtering between two 6 million polyline relations. Bulk-loaded R-trees, such as the STR and Hilbert strategies outperform the insertion-based strategies by a wide margin. The GiST is outperformed by 4.5-6x by the bulk-loaders (which are not supported in GiST, as previously mentioned), on all page sizes. Furthermore, the GiST is 1.5-2x slower for the spatial join than the raw NLA it is based on. The GiST algorithm is even outperformed by the Quadratic strategy, due to the overhead of searches, even though the R-tree that results from the Quadratic strategy is much worse in terms of overlap.

5.3 Discussion

We observe that the generality of the GiST index compared to a custom solution comes at a high performance price. The reason is that GiST abstractions are restrictive, hindering (i) dedicated index optimizations, (ii) implementation of better index variants, and (iii) omission of unnecessary functionality. Although the custom index implementations have a performance advantage in isolation, we do not advocate such stand-alone solutions. There are advantages to the index being deeply-integrated into the database engine, in that a smart cost-based query optimizer can leverage the available indexes to determine efficient plans for query execution.

The lesson is that, in designing a better indexing framework for any BigData domain, we need to keep abstractions simple and decoupled from index functionality. A generic index interface within the database engine should describe how an index operates at a high-level, drawing the abstraction at the level of index operations (e.g., index a piece of data, retrieve an identifier for it, etc.), rather than how it should do them (e.g., the 6 methods of the GiST, with their restrictive assumptions on index properties).

The second observation we made is that although flexibility should be a first-class citizen when designing a new framework, sometimes restrictions can help avoid poor implementation decisions. For example, variable-length index entries may be necessary in some cases, but for the most part the entry reconstruction overhead is not worth it. While it is acceptable for a variable-size table field to be reconstructed when needed (presumably only a small portion of the data is retrieved anyway, if the associated index is efficient), an index on such data should be fast and cannot afford that cost. Therefore, indexing variable-sized data should be done using a compressed representation of it (e.g., a prefix for variable-length strings, an MBR for geometries, etc.).

Given the multitude of BigData application domains that can benefit from efficient indexing to prune the search space, exposing the tradeoff between generality and performance is an important lesson to any future designs of indexing frameworks for large volumes of data. We plan to use our findings to re-think such a general framework with less assumptions on what the structure of the index is, or how it operates, and only impose beneficial restrictions.

As future work, we plan to profile different types of queries which make use of a spatial index, as well as spatial datasets that have different properties and data distributions.

6. CONCLUSION AND FUTURE WORK

Spatial indexing plays a major role in fast retrieval of useful information from BigData, which means that an efficient index design is paramount. The variety of emerging spatial BigData applications are calling for new and improved indexing structures, which are often difficult to integrate in a large and rigid DBMS engine codebase. As a result, while a general indexing framework is desirable, the abstractions involved may cause considerable slowdown, if the design does not consider performance.

We studied the generality-performance tradeoff with general indexing schemes, such as the GiST. To identify the performance penalty imposed by the GiST framework, we analyzed PostgreSQL and showed where time is spent in a large-scale spatial query that uses spatial indexes. The GiST framework, as implemented in PostgreSQL/PostGIS, performs poorly for filtering records through the index. We implemented an optimized GiST framework outside of the DBMS, to show that even when tested in isolation, the GiST adds a 1.5-2x overhead over insertion-based R-tree strategies, and 4.5-6x overhead over bulk-loaded R-trees, at several page sizes.

We empirically showed that when designing a generic framework for indexing, drawing the abstraction line in the wrong place can cause a significant performance penalty, while also restricting the types of indexable data domains. As future work, we plan to isolate basic characteristics of spatial (and non-spatial) indexes and to implement a framework that makes no assumptions on the data or index properties, or the way the index should operate. Our aim is to provide index generality with minimal performance impact.

7. REFERENCES

[1] C.-H. Ang and T. C. Tan. New linear node splitting algorithm for R-trees. In *SSD*, pages 339–349, 1997.

[2] W. G. Aref and I. F. Ilyas. SP-GiST: An extensible database index for supporting space partitioning trees. *J. Intell. Inf. Syst.*, (2-3):215–240, 2001.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[5] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database system. In *VLDB*, pages 562–573, 1995.

[6] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *VLDB*, pages 500–509, 1994.

[7] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. Technical report, 1997.

[8] TIGER®. http://www.census.gov/geo/www/tiger.