VIII Brazilian Symposium on GeoInformatics, Campos do Jordão, Brazil, November 19-22, 2006, INPE, p. 35-52.

Exploiting Type and Space in a Main Memory Query Engine

Thomas Schwarz, Matthias Grossmann, Daniela Nicklas, Bernhard Mitschang

Universität Stuttgart, Institute of Parallel and Distributed Systems (IPVS)
Universitätsstrasse 38, D-70569 Stuttgart, Germany

thomas.schwarz@ipvs.uni-stuttgart.de

Abstract. *More and more spatial data is accessible over the web or through portals of wireless service providers. In this context the main selection criteria for the data are the type of the requested data objects and their position in the real world. Integration and performance issues are challenged by the need to process ad hoc queries in an interactive fashion. In this paper we investigate how a main memory query engine can be used to meet these requirements. It has the added benefit of being easily deployable to many components in a large-scale data integration system. Hence, we analyze how such a query engine can best exploit the query characteristics by employing an index structure that leverages spatial and type dimensions.*

In order to support query processing in the best possible way we investigate a specific multi-dimensional main memory index structure. Compared to the straightforward approach using separate indexes on type and position we can increase the performance up to almost an order of magnitude in several important usage scenarios. This requires to tweak the mapping of type IDs to values in the type dimension, which we discuss extensively. This enables the overall system to be used interactively, even with large data sets.

1. Introduction

In the upcoming areas of location-based services and ubiquitous computing new data-intensive applications emerge, which support their users by providing the right information at the right place, i.e., providing on demand what fits best to the user's current situation. Usually, the user's position and the application he is currently using determine the relevant information, so most information requests issued by the application contain spatial predicates and predicates restricting the type of the data. In this paper, we present a dedicated main memory query engine that is tailored to this environment and that supports application-specific processing capabilities. In particular, we analyze which index structures are best suited to maximize its performance.

The idea for this query engine emerged from the experiences with a data and service provisioning platform for context-aware applications. Data providers manage spatially referenced data, e.g., rooms, facilities, and sensors in a building, or the map data of a city. There, several data management systems that are specialized to the characteristics of the managed data (i.e., update rate and selection usage) [9] have been developed. In order to combine the data of multiple providers an integration middleware [24] has been developed. It achieves a tight semantic integration of the data instances using an extensible integration schema [16]. A plug-in concept allows to employ domain-specific functionality in the middleware like detecting duplicates, merging multiple representations, or aggregating and generalizing (map) data. The platform is used by various location-based applications like a city guide (a tourist application) or a digitally assisted scaven-

ger hunt (a multiplayer mixed reality game) [17]. According to our experience, applications get by with simple selection queries.

Our query engine is also of interest to others. It can be directly integrated into implementations of the OGC Catalogue Services standard [18] or within the FGDC clearinghouse [15], which both offer a discovery mechanism for digital geospatial data. Similarly, implementations of geographic information systems may profit from our query engine. Furthermore, grid metadata catalog services [26] or discovery services in a service-oriented architecture [2] can apply our approach in order to optimize their engines that select different types of resources or services based on given restrictions.

1.1. Contribution

In this paper, we describe the design and implementation of a main memory query engine employing an index structure that leverages spatial dimension and type dimension, such that location-conscious queries are most efficiently supported. The focus is not on indexing and index structures, but on configuring the query engine's internal data structures to exhibit the best possible index organization. In order to do this, we evaluate two different approaches to organize an index structure that combines a spatial dimension and a type dimension. We detail on three different variants to map the type information (type IDs) to values in the type dimension. This has a substantial impact on the performance of the query engine, but has not been considered previously. We also point out how to determine the best range for the mapped values.

Many components in a large-scale information system may profit from the proposed query engine. Therefore, we describe a solution architecture for such an information system and introduce four different usage scenarios for four of its components, each having different characteristics. In order to achieve a sub-second response time of the overall system (including network latencies, (de)serialization and other processing overhead) the individual query engines have to process a typical query returning about 1,000 objects in 10 milliseconds, as a rough estimate. Therefore, we emphasize a main memory approach in order to achieve fast response times and allow for an easy deployment.

We run a substantial number of experiments and assess the suitability of the various techniques specifically for each scenario. Compared to an approach using separate indexes on type and space we can increase the performance up to almost an order of magnitude in certain cases. Our goal is to enable the reader to apply our insights profitably to his problem at hand.

The remainder of this paper is structured as follows. In Section 2 we introduce the typical data managed by our query engine and the typical queries issued. In Section 3 we characterize its usage scenarios. We describe the different approaches to organize the index structures used by our query engine in Section 4. In Section 5 we describe the conducted experiments and analyze their results. We give an overview on the related work in Section 6. Finally, we conclude the paper and indicate future work in Section 7.

2. Data and Queries

Typically, applications in the domain of location-based or context-aware applications operate on object-structured data, see Figure 1. In the GIS world, objects are also called "features". The schema consists of a collection of types. An object is associated to a type

which determines the name and data types of the attributes that an object of this type may use to store information. Types are structured in a is-a-hierarchy, see Figure 1 for a typical example.

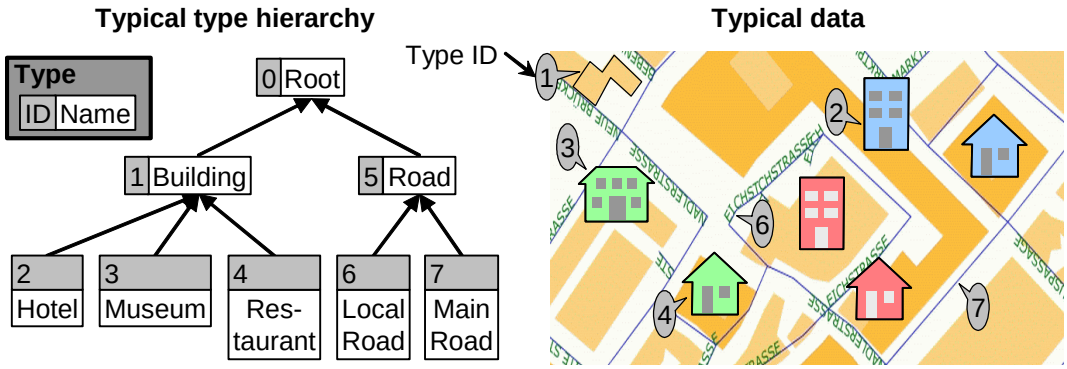

Figure 1: Simplified excerpt of a typical type hierarchy (schema) and typical data

We assign a unique number called type ID to each type. Using an optimal assignment (termed linearization in [14]) we are able to determine for each type a continuous interval that contains exactly the type's own ID and the IDs of the subtypes of this type. This works always for type hierarchies with single inheritance [14].

We assume that in the targeted application domains every object has a position or an extent so that already the root type of the hierarchy comprises a generic geometry attribute that we exploit for indexing purposes. Examples for such schemas are the TIGER/Line data model [27], augmented world models like the one used in [16], or the upcoming standard for city models, CityGML [11]. Objects have linestrings and polygons as geometries, which all can be approximated by bounding boxes from an indexing point of view. Typical data sets comprise various kinds of roads (local road, main road, highway, ...), buildings, points-of-interest (museum, church, viewpoint, ...), and so on, see Figure 1 for some ideas.

Expressed in natural language, typical queries are "Give me all roads (no matter what kind) in the given rectangle", "Give me all major roads in the corridor between my current and my target position", or "Give me all French restaurants within 1 mile". All these queries have in common that they have a spatial predicate restricting the position of the result objects and a type predicate restricting the type of the result objects. Usually, the query addresses also all subtypes of the sought type. Therefore, we strive for exploiting this commonality by supporting such queries with a tailored index approach.

3. Usage Scenarios

An information system can employ the proposed query engine in various ways and places. We focus on location-based and context-aware systems that integrate data dynamically from many data providers ranging from web sites over digital libraries and geo-information systems to sensors and other stream-based sources. Figure 2 shows a typical architecture for such systems.

The processing model is as follows. An application on a *mobile device* issues a query for data relating to the user's vicinity. The query is first processed by the local query engine. A query for the missing data is issued to the *integration middleware*.

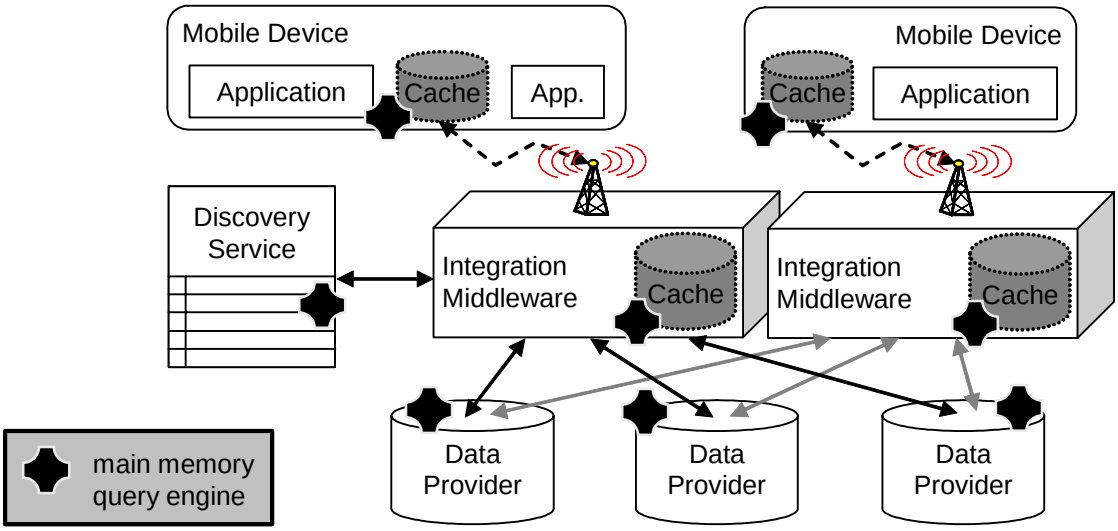

Figure 2: Architecture of a location-conscious data provisioning system integrating data from various providers

There, the query engine of the middleware processes the query. For retrieving the missing data the integration middleware determines the relevant providers using the *discovery service*, which itself runs a query engine. The integration middleware requests the data from these *data providers*. They evaluate location-based queries using a query engine as well. When integrating their results a query engine supports the integration middleware in evaluating additional predicates and performing location-based data merging. Finally, the integrated result is sent to the application. On all levels query processing can be considerably enhanced by means of data caches (cf. Figure 2).

As mentioned before, those query engines basically consist of a specific index structure that supports predicate-based queries that predominantly consist of location and type predicates. Hence, the efficiency of the query engine is mostly determined by the performance of the supporting index structure. Obviously, all of the above four types of components (mobile device, integration middleware, discovery service, and data provider) running such query engines do benefit from the index structures we investigate in this paper.

However, each component manages a different piece of the data, has different typical queries, and updates or exchanges the data in a different way and at a different frequency. Hence, we analyze the experiments in Section 5 individually for each component. Table 1 summarizes and quantifies these characteristics, which have been derived from the experiences with our service provisioning platform for context-aware applications [24].

The term selectivity factor (SF) refers to the ratio of objects qualifying for the result set to the total number of objects (the data set size). If a predicate has a low SF then only few objects qualify for the result set, and vice versa. The Spatial SF refers to the ratio of the area of the query window to the area of the data set's universe, which is given by the convex hull around the geometries of all objects in a data set. Update rate counts the number of objects that are updated between two consecutive queries. E.g., an update rate of 0.1 means that only one object is changed during a period where ten queries are processed.

Table 1: Selectivity factors (SF) and update rates (number of updated objects per number of queries) of the usage scenarios

Usage Scenario	Spatial SF	Type SF	Update rate
Data Provider	1% - 20%	20% - 100%	0.01
Discovery Service	1% - 5%	1% - 20%	0.1
Integration Middleware	10% - 50%	1% - 20%	10
Mobile Device	10% - 50%	10% - 100%	100

3.1. Summary of the Requirements

From the previous discussion we can devise the following requirements to our location-conscious query engine:

- Simple query capabilities suffice. Applications get by with predicate-based selection queries.
- Combine Type and Space. Typical queries contain a spatial predicate and a type predicate.
- Cope with different workloads. The selectivity factors of the typical queries and the update rate depend on the usage scenario.
- Fast response times. In order to allow for interactive applications that are backed by a complex information system, the individual query engine has to respond in the order of 10 milliseconds.

Concerning the query capabilities there is a nice analogy of our approach to the well-known one of XPath processing. Our dedicated query engine compares to a full blown SQL engine in a similar way as an XPath engine compares to an XQuery engine. Likewise, its performance is determined by the performance of its internal index structures, again similar to the XPath engine, whose performance depends on its internal data structures. We emphasize a main memory approach in order to achieve fast response times and to allow for an easy deployment. Hence, we need to reflect on how to organize main memory data structures and add location information to them in order to bring out the best performance. For this, the next steps are to describe our data structures and how the query engine uses them to process queries in Section 4, analyze the performance in Section 5, and review previous approaches in Section 6.

4. Index Structures

In this section we give details about the different approaches that we investigated to build an index structure that combines the spatial dimension and the type dimension. This combination is a natural consequence from the observation in Section 2 that the majority of queries involves selection predicates on at least these two dimensions. For each approach we explain how the query engine processes a query step by step.

We refer to the spatial dimension as a single dimension, although it actually involves two dimensional coordinates. Also, we will abstract from the details of particular spatial index structures (e.g., R*-Tree, Grid-File, or MX-CIF Quadtree, see [8] for a survey) because the underlying spatial index structure can be easily exchanged without significantly shifting the relative performance of the presented approaches. We focus on

how to combine existing well-known index structures in new ways to best solve the problem at hand.

The following approaches are designed to work in main memory, which is a requirement to achieve reasonable response times. Partitioning techniques can be applied to split the data into chunks that a single system can maintain in main memory. Furthermore, this allows us to flexibly deploy our query engine to any component in the entire system with very little administrative overhead in contrast to deploying a full fledged database system.

All approaches use hash data structures that map a type's name to its ID, and this ID to a list of the IDs of all sub or super types in constant time. The size and contents of these lookup tables depends only on the type hierarchy so that they are small in size compared to the entire data set and they are not affected by updates.

4.1. Separate Indexes (SEP)

The Separate Indexes approach maintains two distinct data structures, see Figure 3. The first data structure uses a spatial index to organize the objects solely by their geometry. The second data structure uses an array containing for each type a separate list of the corresponding objects. Objects are inserted into both data structures, which have to be in sync at all times.

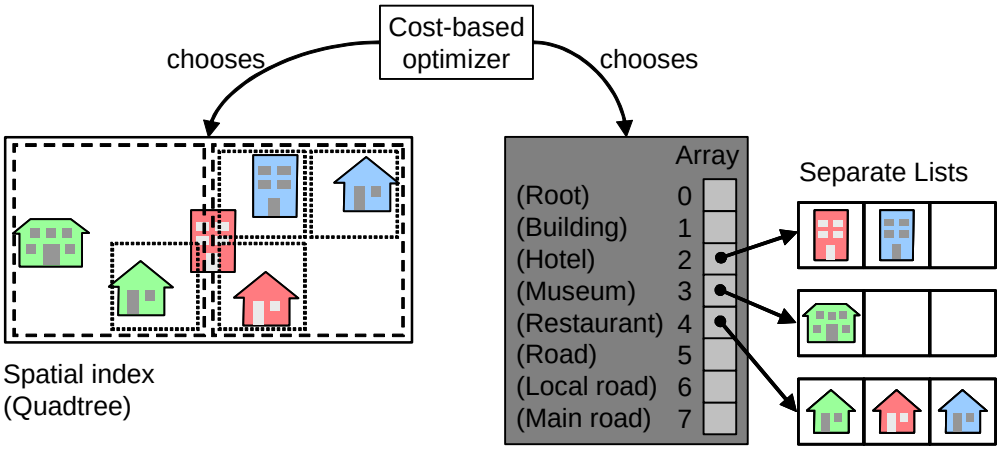

Figure 3: Data structures in the Separate Indexes (SEP) approach

For answering a query a cost-based optimizer assesses the selectivity of the spatial predicate and the type predicate. Then, the more selective predicate (lower selectivity factor) is used to generate a list of candidates using the corresponding data structure. Finally, these candidates are filtered using the remaining predicate.

This approach gives us the bottom line of the least achievable performance as it uses only standard database technology without any problem-specific improvements. This approach tends to be slow because it exploits an access path for at most one dimension and filters all candidates along the other dimension.

4.2. Real 3D Index (R3D)

The Real 3D Index approach is especially tailored to the typical queries introduced in Section 2. It maintains a single spatial index that involves three orthogonal dimensions, see Figure 4. Two dimensions are used to store the bounding boxes of the objects' geometries. The third dimension is used to store the objects' type ID. Each object is inserted into this index only once.

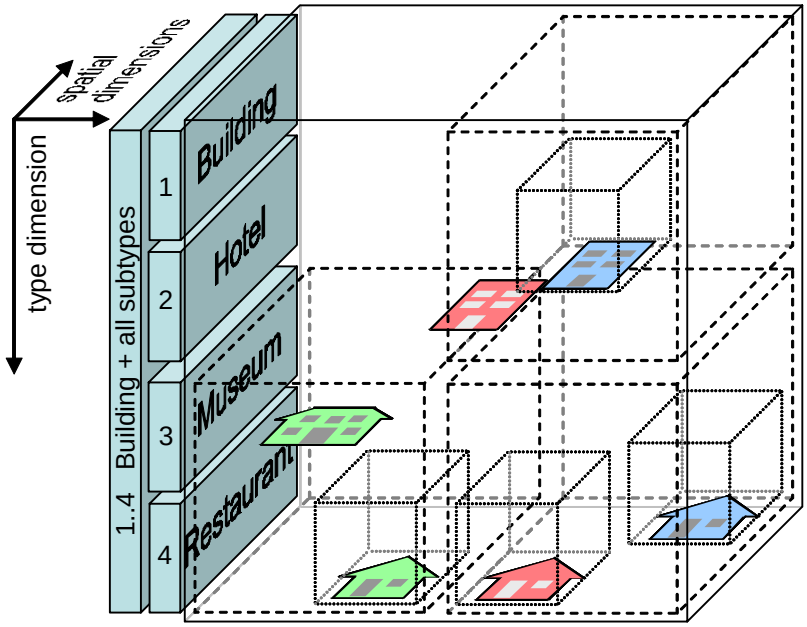

Figure 4: Data structures in the Real 3D Index (R3D) approach

In this approach a query involving a spatial predicate and a type predicate can be expressed as a three dimensional bounding box where the range in the type dimension comprises the ID of the sought type and the IDs of all of its subtypes. Thus, each query can be translated into a single bounding box that is used for a single traversal of the Real 3D Index.

The mapping of types to values in the type dimension is the critical aspect in this approach. As shown in Figure 5 the space between the mapped values of two adjacent types influences the clustering of objects and child nodes in the inner nodes of the index tree. If there is a large gap between the mapped values of two types (wide spacing, left part of Figure 5), then objects are grouped by their type value rather than by their position in the spatial dimension. In the example objects with three different positions and only two different types are grouped in the same inner index node. If the mapped values of two types are close to each other (narrow spacing, right part of Figure 5), then it is vice versa. Inner index nodes store objects with only two different positions but three different types. This is due to the fact, that most indexing methods try to keep the bounding boxes of the inner nodes as squarish as possible. As we will see in Section 5, the spacing of the values in the type dimension has a huge impact on the performance of the index. It determines if whole branches can be pruned away when traversing the index tree. With wide spacing, subtrees with the wrong type can be skipped quite early. With narrow spacing, the same goes with subtrees having too distant positions.

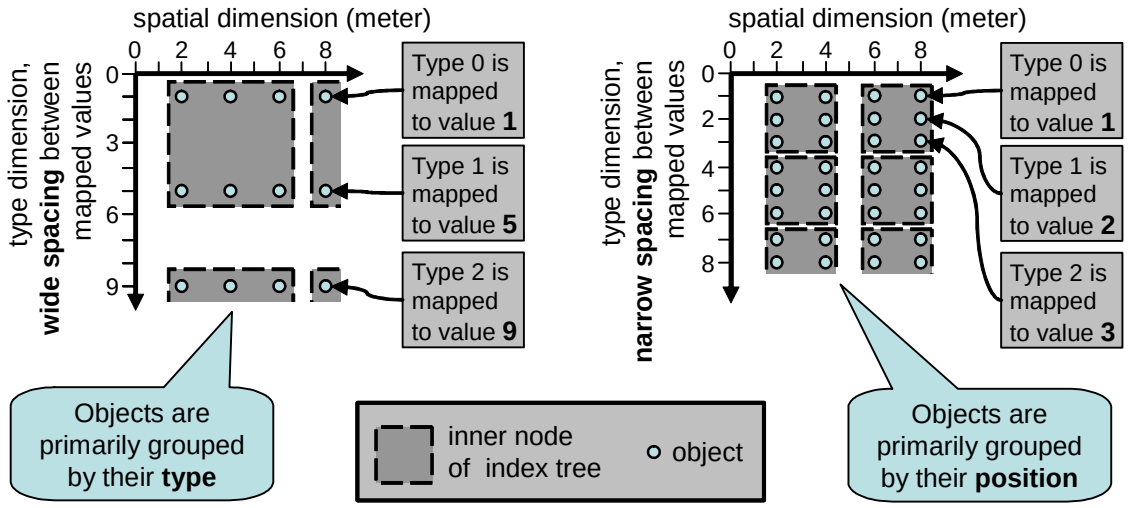

Figure 5: Effects of the spacing between mapped type values in the type dimension on the clustering of objects in the inner index nodes

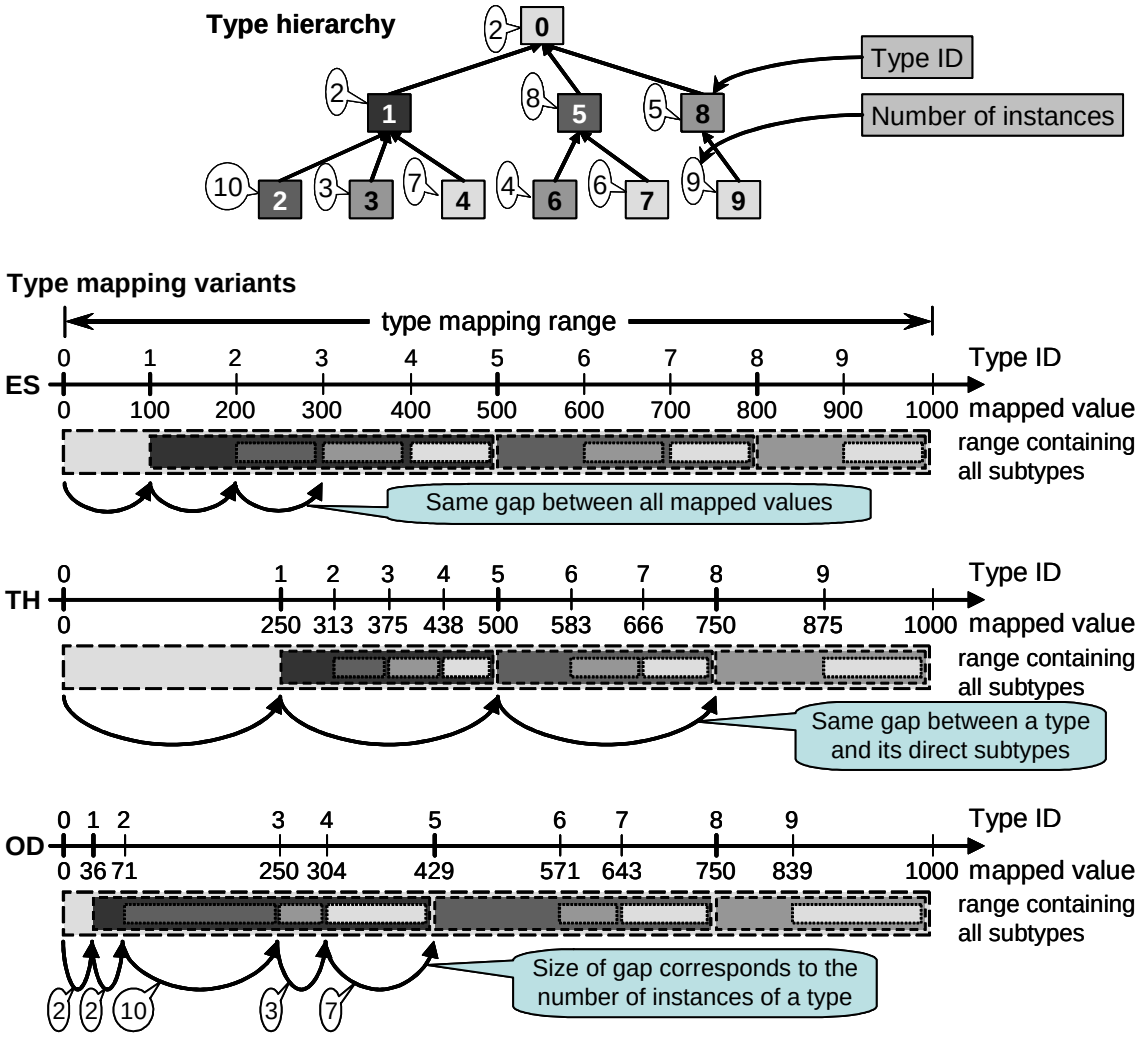

Figure 6: Computing the mapped type values for the different type mapping variants

We have investigated three different kinds of spacing, which are visualized in Figure 6. In all variants, the mapped values are scaled to span a predetermined type mapping range, which is about as large as the average distance between neighboring objects multiplied by the number of types in the type hierarchy, see Section 5.2 for more details. The dashed boxes indicate the range containing the mapped values of a type and all of its (transitive) subtypes.

- Bottom-up **equal spread** (ES): Each type's value in the type dimension equals the type's ID multiplied by a constant scaling factor. This is the simplest variant which disregards the type hierarchy to a large extent. It is a close relative to the approach pursued in [14]. In the example in Figure 6 we have 10 types and the type mapping range is 1000 units, so that the distance between two mapped values is 100 units.
- Top-down biased by **type hierarchy** (TH): On each level of the type hierarchy the available mapping range is evenly distributed among the current type and its subtypes. In Figure 6, type 0 has three direct subtypes (1, 5, and 8), so that the available mapping range of 1000 units is split into four segments. Type 5, in turn, has to split its range (500 to 749) into three segments because it has two subtypes. This variant preferredly groups objects having the same supertype in the inner tree nodes.
- Top-down biased by **object distribution** (OD): On each level of the type hierarchy the available mapping range is distributed among the current type and its subtypes based on the number of instances each type has. In Figure 6, we have a total of 56 object instances. Two object instances have type 0, so that the gap to type 1 is $\frac{2}{56} \cdot 1000 \approx 36$ units. Note that this method requires additional statistical knowledge on the object distribution. This variant groups objects in the inner tree nodes according to the actual distribution of the objects on the types. Frequent types get their own subtree already close to the root of the index tree. Objects having rare types are predominantly grouped by their geometry and the index tree splits up by type only very close to the leaves of the index tree, see also Figure 5.

5. Experiments

In order to assess the performance of the approaches we implemented all of them in Java. We used the MX-CIF quadtree [23] implementation of the JTS Topology Suite [31] as our spatial index structure and adapted it to cope with more than two dimensions. We added some optimizations so that in the end it was faster than the XXL library's [30] R*-Tree implementation for both inserting and querying objects. However, we point out that the actually used spatial index method has only a marginal influence on the relative performance of the different approaches.

We used a dual processor Dell workstation having 2GHz Intel Xeon processors and 2 gigabytes of RAM, half of which was assigned to the Java virtual machine. All experiments were run on a single processor while the other one was idle to minimize disturbances caused by the operating system. In order to get a reasonable precision when measuring sub-millisecond response times we used the high resolution timer package [22].

We conducted the experiments using a subset of the TIGER/Line 2003 data sets [27]. In particular, we ran queries against the data sets of nine counties in California, see

Table 2: Data sets used in the experiments

Abbre-viation	County	Size	Universe (in km)		Number of objects
			Width	Height	
#1	Yuba	small	33.2	27.6	11923
#2	Glenn		42.6	70.8	16839
#3	San Francisco		15.0	78.4	22666
#4	Alameda	medium	37.6	49.7	46285
#5	Santa Clara		42.1	42.8	53727
#6	Sacramento		49.2	45.5	71743
#7	Riverside	large	59.0	26.5	151489
#8	Kern		98.6	17.0	175082
#9	San Diego		90.6	114.8	203122

Table 2 for their characteristics. We extracted the linestring and polygon based features leading to data sets comprising between 12k and 200k objects.

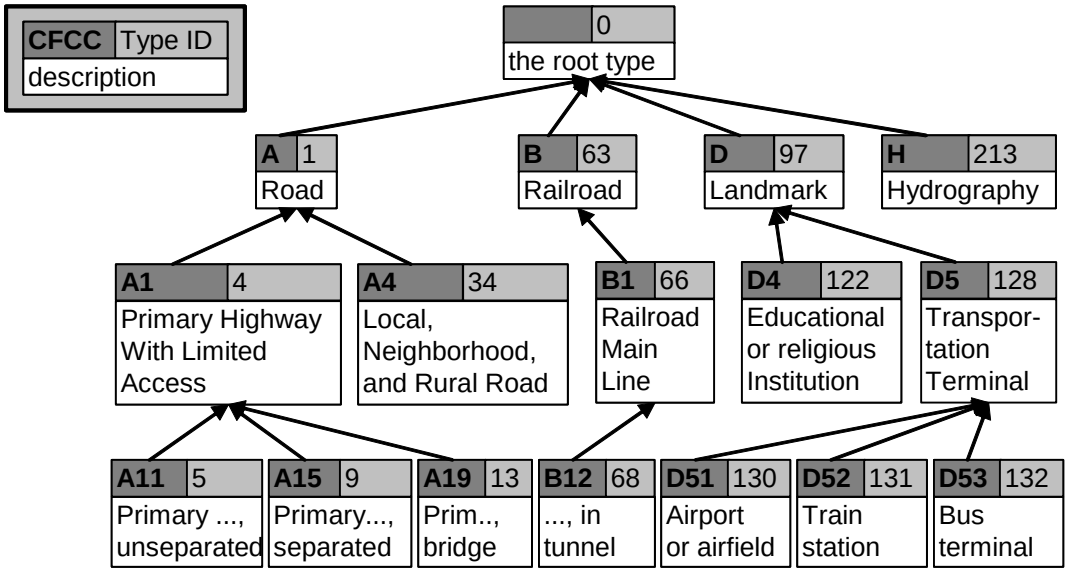

Figure 7: Excerpt of the type hierarchy based on the CFCC feature codes used in the TIGER/Line data sets

The data sets contain data about roads, railroads, other ground transportation, landmarks, hydrography, property boundaries, etc. We interpreted the CFCC feature codes [28] as type names and built up a type hierarchy with four levels, see Figure 7. The single lettered CFCC codes constitute the direct children of the hierarchy's root node. The codes having two letters make up their children, and so on. In total, the type hierarchy consists of 258 nodes.

5.1. Computing the total average query and update response time (TAQURT)

As a system using our query engine cannot switch between different indexing approaches on the fly, we do not compare individual measurements. Instead, we compute a weighted average response time for each combination of indexing approach, usage sce-

nario, and data set. For this, we picked a representative subset of all types and a set of ten differently sized query areas. For each combination of type (fixed type SF) and query area (fixed spatial SF) we measured the response time, leading to a grid of measurements. We interpret the measurements as support points for a piecewise linear surface function having the spatial SF and the type SF as the independent variables. For each approach, usage scenario, and data set we compute the weighted average query response time by computing the weighted integral of the surface function along the axes of the independent variables. The usage scenarios provide the parameters describing typical workloads. Their minimum and maximum SFs (see Table 1) define the integration limits. The surface function is weighted by the reciprocal of the multiplied selectivity factors. The rationale behind this is that small queries retrieving only few objects are more frequent than large queries. Calculating the weighted average response time by integrating the piecewise linear surface function has the benefit of allowing to arbitrarily set the minimum and maximum selectivity factors. Furthermore, the integral allows to calculate a more meaningful average value that takes each measurement's SF into account.

Finally, the usage scenario also determines the frequency of updates. The total average query and update response time (TAQURT) is the sum of the average insertion cost per object weighted by the update rate and the average query response time:

```
TAQURT = averageQueryResponseTime
         + (averageInsertionCostPerObject * updateRate)
```

Thus, we get a TAQURT for each combination of indexing approach, usage scenario, and data set. By aggregating the query and update performance into a single figure we can evaluate the approaches from a "total cost of ownership" perspective and by concentrating on the four usage scenarios we keep the experiments clear.

5.2. Comparing the type mapping variants of the Real 3D Index

In this section, we compare the three mapping variants equal spread (ES), type hierarchy (TH), and object distribution (OD) introduced in Section 4.2 in order to determine the best one. The mapping variants differ in the spacing in the type dimension between the mapped values of two adjacent type IDs. Taking the type IDs themselves as the mapped values in the type dimension (1:1), as proposed in [14], is a very bad idea that leads to an average performance loss between 37% and 418%, see Figure 9.

We vary the range of the mapped type values in five steps, denoted A, B, C, D, and E, see Table 3. The sizes of the selected mapping ranges are around the order of magnitude of the anticipated optimal mapping range, which is approximated by calculating the average distance of the objects of one type along one of the spatial axes (4200 meter in our data sets) and multiplying it by the number of types in the hierarchy (258). This way objects are clustered equally along the spatial dimension and the type dimension in the inner nodes of the index tree. The selected mapping ranges are quite representative as in most cases one of the medium ranges shows the best performance, see Figure 8.

Table 3: Type mapping ranges

mapping range	A	B	C	D	E
total range (in kilometer)	15	150	1,500	15,000	60,000

We compute the columns displayed in Figure 8 as follows. We group the measurements by usage scenario and data set. For each combination of mapping variant and mapping range we compute its relative response time as the ratio of its TAQURT to the minimal TAQURT in each group. Then, we average the relative response time across all data sets.

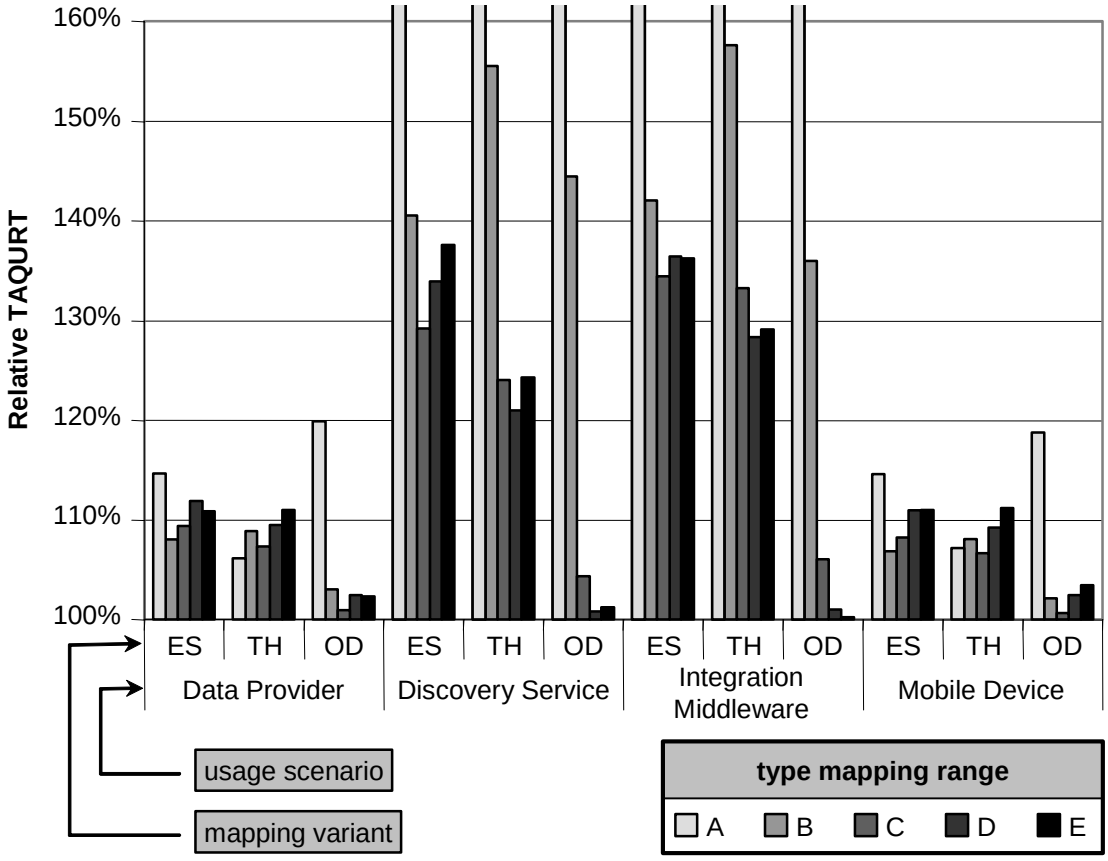

Figure 8: Relative total query and update performance of the Real 3D Index mapping variants for each scenario

The first observation is that the data provider and mobile device usage scenarios are less sensitive to the mapping range than the other two scenarios. This is due to them having a high type SF. The main insight of this figure is that it suffices to get close to (within an order of magnitude) the best possible mapping range to get reasonable performance (5% worse than best possible). However, if you are far off the mark (mapping range A) then the performance degrades considerably. Unfortunately, the best mapping range differs depending on the usage scenario and mapping variant. Averaged across all scenarios, the ES variant achieves the best performance with mapping range C. The TH and OD variants work best with mapping range D.

Figure 9 displays a subset of the results shown in Figure 8. For each mapping variant in each usage scenario only the column corresponding to the mapping range that yields the fastest relative TAQURT is displayed. Additionally, they are compared to the 1:1 mapping variant and to the Separate Indexes approach (SEP).

Figure 9 clearly shows that the OD variant delivers the best performance. The data provider and mobile device usage scenarios both have a high type SF and the perfor-

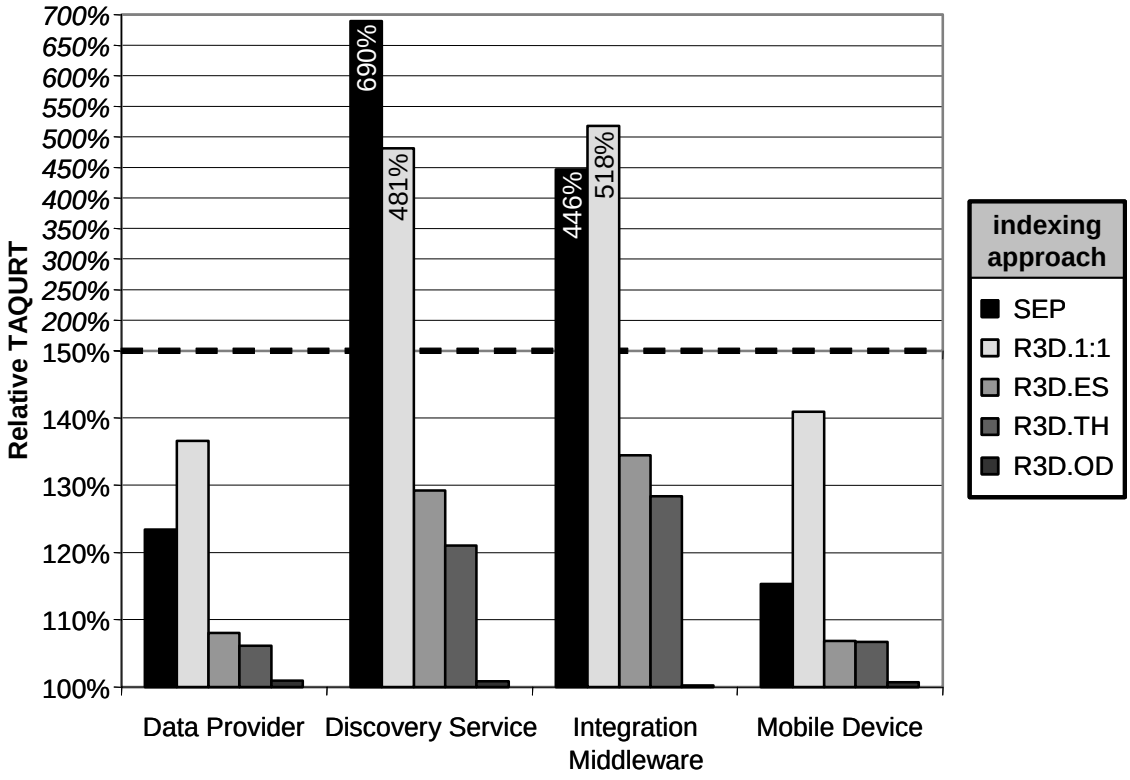

Figure 9: Comparing the best mapping ranges of Figure 8 with the 1:1 mapping variant and the SEP indexing approach for various usage scenarios

mance differences are less pronounced there. The OD variant leads there by about 7%. In the other two scenarios the lead is at least 20%. However, the OD variant has the disadvantage that it needs advance knowledge about the distribution of the objects on the types of the type hierarchy.

The best alternative to the OD variant is the TH variant, which is always between 1% and 8% better than the ES variant. The 1:1 variant is far behind and in most cases it is even worse than the SEP approach. This highlights the importance of choosing an adequate type mapping range. Doing so gives the R3D approach a comfortable lead over the SEP approach in the usage scenarios having a high type SF. When the type SF is low the R3D approach really outperforms the SEP approach by almost an order of magnitude. In the discovery service usage scenario, where spatial SF and type SF are both low, even the 1:1 mapping variant of the R3D approach is faster than the SEP approach.

5.3. Index construction

In this section we analyze the costs involved with maintaining each index structure. The costs are divided into the average time needed to insert an object into the index and the average memory occupied by each object (see Figure 10). In both figures the total values can be determined by multiplying the per object values with the number of objects in the data set. In our further considerations we approximate the cost of updating an object with the cost of inserting one. Thus, we can figure out both the initial cost for setting up the entire index from scratch and the running costs involved with processing updates.

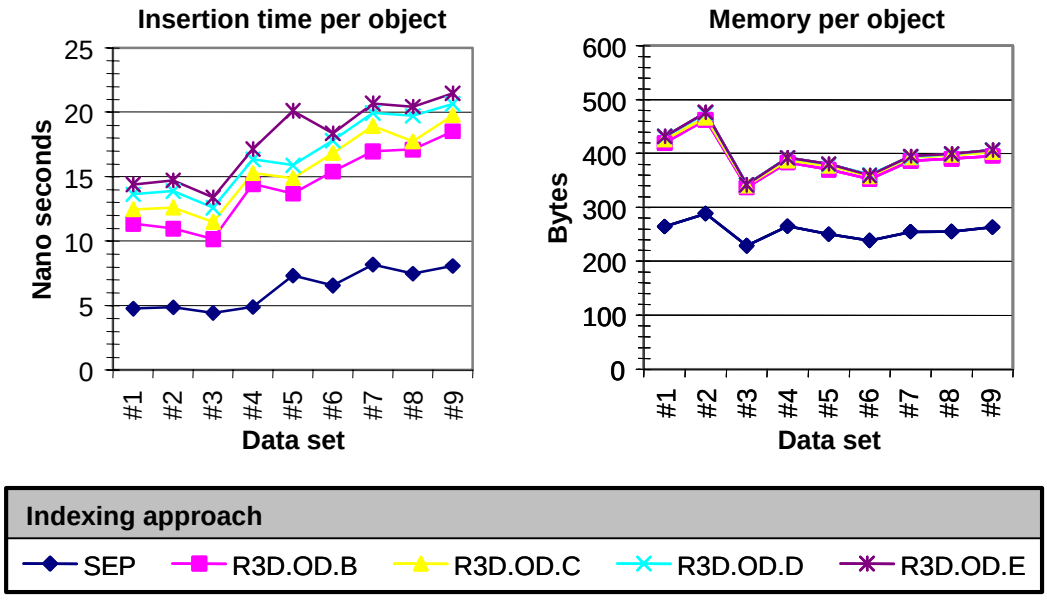

Figure 10: Average time for inserting a single object into the index, and average memory occupied by a single object (including geometry, bbox, type ID, and indexing overhead)

Figure 10 shows for the Separate Indexes approach (SEP) that the time per object increases only slightly with an increasing data set size, so that the approach scales quite well. The Real 3D Index (R3D) is 2.4 to 3.2 times slower than the SEP approach for small data sets. For large data sets it is only 2.2 to 2.6 times slower than the SEP approach. As shown in Figure 10, the type mapping range (B to E) has a significant impact on the index creation time. The larger the type mapping range is, the more time is needed to build the index. The other type mapping variants (ES and TH) not shown in Figure 10 behave similarly.

The average amount of memory (including indexing overhead) occupied by a single object is about the same for all data sets, see Figure 10. Large data sets do not lead to worse memory utilization. The least memory is occupied by the SEP approach. All R3D variants have about the same memory footprint. They occupy about 50% more memory than the SEP approach.

To put it in a nutshell, the SEP approach clearly outperforms the R3D approach in terms of only index maintenance costs. It uses less memory and is faster in building up the index.

6. Related Work

As indicated in the introduction, OGC Catalogue Services [18] (OGC CS) and Grid Metadata Catalog Services [26] (MCS) can benefit from the proposed query engine. Currently, vendors like ESRI, Galdos, or Ionic offer geodata catalog services complying with OGC CS Implementation Specification [18] and geodata servers complying with OGC's Web Feature Service Implementation Specification [19]. However, they focus on building systems that leverage existing ORDBMS and rely on their query engines. Thus, they have little influence on the query engine and they do not offer one that is deliberately customized to the typical query load described in Section 2. Also research projects like

the GDI NRW Testbed [3] are more concerned with designing the overall architecture and interaction patterns than with optimizing the underlying data structures.

In [32], the implementation of an OGC CS using a Grid MCS is investigated. While the authors do not address the internal implementation of the service, it emphasizes the importance and impact of our work. Recent research in the MCS area is concerned with managing extensible sets of arbitrary attributes [6], but not with optimizing index structures or exploiting class hierarchies. In [7], the authors describe how to leverage class hierarchies in ebXML Registries, but the authors do not detail on an efficient implementation or index support.

IBM's Cloudscape and Oracle's TimesTen main memory databases are pure relational systems and do not offer any support for type hierarchies or spatial indexes. MonetDB allows for geographic extensions [4], but still has no support for hierarchical relationships. Current research in this area focuses on minimizing CPU branch mispredictions [21] and developing CPU cache conscious data structures [13] to alleviate the main memory access bottleneck. This research is complementary to ours.

6.1. Spatial indexes

A detailed overview on the most important spatial index structures is given in [8]. As we focus on building a query engine rather than on developing a new index structure we pick an existing index structure that assumedly best fits out requirements of performing well with real data sets and coping with high update rates. We chose the MX-CIF quadtree for meeting these requirements and being simple to implement. We do not recommend the best spatial index structure, but we analyze how to utilize any existing index structure to combine spatial dimension and type dimension.

In [29], several approaches to combine spatial indexes and text indexes are presented in order to build a geographical search engine on the web. They aim at enhancing the search precision and recall. However, in the text part of web pages there are considerably more distinct words than we have types in our type hierarchy. No hierarchical relationships between the words are defined or exploited. Finally, their data is stored in files on disk.

6.2. Object-oriented databases

In the area of object-oriented databases (OODB) related indexing problems have been discussed [10, 12, 14, 20]. Indexes combine an object's type with one or several of its attributes. However, attributes may only have a single value instead of a range as in the spatial dimension. Therefore, only point access methods are considered whereas we need to deal with spatially extended geometries requiring spatial access methods. We provide an extensive analysis using real (spatial) data sets. Also, we consider the effort to accommodate changes in the index in order to get a total-cost-of-ownership assessment of the performance in real usage scenarios.

The Multikey type (MT) index [14] is basically similar to our Real 3D index approach using the type hierarchy based mapping variant. However, [14] concentrates on providing an optimal linearization for type hierarchies having multiple inheritance. Scaling the type dimension is not discussed at all, which proves to have a significant impact in our experiments.

6.3. Object-relational databases

Conceptually, in an ORDBMS each type corresponds to a table and each attribute to a column. While some systems (e.g., PostgreSQL) store objects as a row in the table for its type, other systems (e.g., DB2) store all objects in a single hierarchy table [5] which has an additional column for storing the type of an object. In the first case, many tables have to be queried if the sought type is a non-leaf type in the type hierarchy. In the latter case, DB2 internally creates a two-dimensional index on a given column and on the type column. However, it uses a point access method to accomplish this combination, which is inadequate as we have discussed in Section 6.2.

If we have an explicit type column and separate indexes on this column and the geometry column then we can either fetch the qualifying tuple IDs from both indexes and intersect these sets before fetching the remaining object data [1]. Alternatively, we can pick the more selective index to determine a set of candidates, and filter them afterwards [25]. Our experiments have shown, that the latter approach, which we address in Section 4.1, is always more efficient in main memory.

7. Conclusion

In this paper, we have presented a main memory location-conscious query engine that exploits the characteristics of typical queries, which contain spatial predicates and predicates restricting the type of the data. The query engine can be deployed to many components in a large-scale information system and contribute to let the whole system be usable interactively. Both reasons advocate for a main memory approach.

As our experiments have shown, the Real 3D Index approach offers the best overall performance. However, it is crucial to determine an adequate type mapping range or otherwise performance will degrade considerably. We have investigated three different variants to map type IDs to values in the type dimension. The variant relying of object distribution statistics offers the best performance, however the statistics have to be collected beforehand. Both aspects have not been discussed previously. In the usage scenarios with a high type selectivity factor the Real 3D Index approach beats the Separate Indexes approach, which uses conventional database technology, by about 20%. If the type selectivity is low, then the lead increases to almost an order of magnitude. However, the Separate Indexes approach uses only two thirds of the memory and builds up its index at least twice as fast compared to the Real 3D Index approach.

In future work, we will investigate the deployment of the query engine to all components of our data and service provisioning platform for context-aware applications and assess its impact on the overall system. We intend to optimize the overall processing performance under changing workloads (mobile users) and changing data (high level context information derived from sensor data). Thus, our main memory query engine paves the way for virtualizing the whole query processing task by facilitating the distribution of query capabilities across several components based on resources, load, cache contents, etc. Additionally, we plan to extend the index by further dimensions such as valid time or measurement time.

References

[1] M.M. Astrahan, D.D. Chamberlin: *Implementation of a Structured English Query Language,* Communications of the ACM, 18(10), 1975

[2] D.K. Barry: *Web Services and Service-Oriented Architectures,* Morgan Kaufmann Publishers, 2003

[3] L. Bernard: *Experiences from an implementation Testbed to set up a national SDI,* 5th AGILE Conf. on Geographic Information Science, Palma, Spain, 2002

[4] P.A. Boncz, W. Quak, M.L. Kersten: *Monet And Its Geographic Extensions: A Novel Approach to High Performance GIS Processing,* Proc. of the 5th Intl. Conf. on Extending Database Technology (EDBT), Avignon, France, 1996

[5] M. Carey et al.: *O-O, What Have They Done to DB2?* 25th Intl. Conf. on Very Large Data Bases (VLDB), 1999

[6] E. Deelman et al.: *Grid-Based Metadata Services,* 16th Intl. Conf. on Scientific and Statistical Database Management (SSDBM), Santorini Island, Greece, 2004

[7] A. Dogac, Y. Kabak, G.B. Laleci: *Enhancing ebXML Registries to Make them OWL Aware,* Journal on Distributed and Parallel Databases, 18(1), July 2005

[8] V. Gaede, O. Günther: *Multidimensional Access Methods,* ACM Computing Surveys, 30(2), June 1998

[9] M. Grossmann et al.: *Efficiently Managing Context Information for Large-scale Scenarios.* 3rd IEEE Conf. on Pervasive Computing and Communications (PerCom), Kauai Island, Hawaii, March 8-12, 2005

[10] W. Kim, K.C. Kim, A. Dale: *Indexing techniques for object-oriented databases,* In: W. Kim, F.H. Lochovsky (eds.): "Object-Oriented Concepts, Databases, and Applications", Addison-Wesley, 1989

[11] T.H. Kolbe, G. Gröger, L. Plümer: *CityGML – Interoperable Access to 3D City Models,* 1st Intl. Symp. on Geo-Information for Disaster Management (GI4DM), Delft, The Netherlands, 2005

[12] C.C. Low, B.C. Ooi, H. Lu: *H-trees: A Dynamic Associative Search Index for OODB,* ACM SIGMOD Intl. Conf. on Manangement of Data, San Diego, California, 1992

[13] S. Manegold, P.A. Boncz, M.L. Kersten: *Optimizing database architecture for the new bottleneck: memory access,* VLDB Journal, 9(3), Dec 2000

[14] T.A. Mueck, M.L. Polaschek: *A configurable type hierarchy index for OODB,* VLDB Journal, 6(4), 1997

[15] D. Nebert: *Information Architecture of a Clearinghouse,* WWW Conference, 1996, http://www.fgdc.gov/publications/documents/clearinghouse/clearinghouse1.html

[16] D. Nicklas, B. Mitschang: *On building location aware applications using an open platform based on the NEXUS Augmented World Model.* Software and Systems Modeling, Vol. 3(4), 2004

[17] D. Nicklas et al.: *Design and Implementation Issues for Explorative Location-based Applications: the NexusRallye.* VI Brazilian Symposium on GeoInformatics (GeoInfo), 2004

[18] Open GIS Consortium: *Catalogue Service Implementation Specification,* Version 2.0.1, Document 04-021r3, 2004-08-02

[19] Open GIS Consortium: *Web Feature Service Implementation Specification,* Version 1.1, Document 04-094, 2005-05-03

[20] S. Ramaswamy, P.C. Kanellakis: *OODB Indexing by Class-Division,* ACM SIGMOD Intl. Conf. on Manangement of Data, San Jose, 1995

[21] K.A. Ross: *Selection Conditions in Main Memory,* ACM Trans. on Database Systems, 29(1), March 2004

[22] V. Roubtsov: *My kingdom for a good timer! Reach submillisecond timing precision in Java,* JavaWorld, 2003, http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html

[23] H. Samet: *The Design and Analysis of Spatial Data Structures,* Addison-Wesley, Reading, MA, 1990

[24] T. Schwarz et al.: *Efficient Domain-Specific Information Integration in Nexus.* Proc. of the 2004 VLDB Workshop on Information Integration on the Web (IIWeb), Toronto, Canada, 2004

[25] P.G. Selinger et al.: *Access Path Selection in a Relational Database Management System,* ACM SIGMOD Intl. Conf. on Management of Data, Boston, Massachusetts, 1979

[26] G. Singh et al.: *A Metadata Catalog Service for Data Intensive Applications,* ACM/IEEE Conf. on Supercomputing (SC), Phoenix, Arizona, 2003

[27] U.S. Census Bureau: *TIGER/Line Files,* http://www.census.gov/geo/www/tiger/

[28] U.S. Census Bureau: *TIGER/Line Files Technical Documentation,* April 2002, http://www.census.gov/geo/www/tiger/tigerua/ua2ktgr.pdf

[29] S. Vaid, C.B. Jones, H. Joho, M. Sanderson: *Spatio-textual Indexing for Geographical Search on the Web,* 9th Intl. Symp. on Spatial and Temporal Databases (SSTD), Angra dos Reis, Brazil, 2005

[30] J. Van den Bercken et al.: *XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries,* 27th Intl. Conf. on Very Large Data Bases (VLDB), Roma, Italy, 2001

[31] Vivid Solutions: *JTS Topology Suite,* http://www.vividsolutions.com/jts/jtshome.htm

[32] P. Zhao et al.: *Grid Metadata Catalog Service-Based OGC Web Registry Service,* 12th annual ACM Intl. workshop on Geographic information systems (ACM GIS), Washington DC, 2004