INFORMATIK

BERICHTE

357 – 09/2010

Assessing Representations for Moving Object Histories

Christian Düntgen, Thomas Behr, Ralf Hartmut Güting

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
Postfach 940
D-58084 Hagen

Assessing Representations for Moving Object Histories

Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting

Abstract

This article proposes three different modes for the sliced representation of the histories of moving objects in databases: the Compact Representation, the Unit Representation and the Hybrid Representation. To this end, unit types are introduced as new DB data types to represent single data slices. The potential influence of different parameters on the performance of the data models is discussed. Based upon the BerlinMOD benchmark, the properties of the representations are considered and expectations for their performance for different categories of queries are formulated. After this, the benchmark is performed using the three representations, two different semantics and two clustering types for the sliced data. The observed results are compared with the expectations, and results for selected queries are discussed. The complete implementation of data types and operators, and all used query plans are freely available for further studies as algebra modules and query scripts for the SECONDO DBMS.

Index Terms

H.2.1: Data Models; H2.2.3: Database Semantics; H.2.4, H.2.m: Spatio-temporal Databases, Moving Objects, Trajectory Databases.

I. INTRODUCTION

Current database systems are able to store large sets of data. Besides standard data, more complex kinds of data may be stored, e.g. multimedia, spatial or spatio-temporal data. Whereas storing and efficient access of standard data is well unterstood, this is still a challenge for more complex data. In the last decade, tracking the location of mobile entities and geo-tagging has become easy using GPS devices, which still are getting even smaller, cheaper, and more precise. As a consequence, these devices have become popular and an impetus for the development of new applications, like location-aware services. Though collecting moving object data has become easy, its representation and processing in databases is still an open field for research.

Moving objects databases (MODBs) come in two flavors: (i) representing current movements, e.g. of a fleet of trucks, in real time, supporting questions about current and expected near future positions, and (ii) representing complete histories of movements, allowing for complex analyses of movements in the past. In this article, we focus on the second approach and present different ways of representing the histories of moving object data (*MOD*). Typical questions that can be answered using moving object history data are: “Where was object o at time t?”, “When did object o have a certain property?”, “Which vehicles have met each other at point p?”. To answer them efficiently, we need sophisticated methods to represent and access the histories. Though we focus on unconstrained linear time-sliced spatio-temporal data, our approach is general enough

The authors are with the Faculty of Mathematics and Computer Science, University of Hagen.
E-Mail: {christian.duentgen, thomas.behr, rhg}@fernuni-hagen.de

to be applicable also to other kinds of time-sliced data, like constrained moving objects (e.g. objects moving on a network) and non-linear movements, as well as non-spatial temporal data.

The contributions of this paper are:

- We introduce new respresentations for MOD.
- We introduce unit types as data types in a DBMS, which were regarded as an internal implementation issue and a formal concept by now.
- We provide a first comprehensive comparison of query processing on different MOD representations, which is useful for query optimization.
- We analyse the advantages of so-called summary fields in MOD representations.
- We exemplarily analyse equivalent query plans in order to explain the results.
- We make all query plans available for further analyses and experiments.

The remainder of the paper is organized as follows. After an overview on related work in Section II, and revisiting a formal data model for representing moving object histories in Section III, we present our suggested models of representing moving object data in Section IV. Subsequently, we introduce the BerlinMOD scenario as a benchmarking tool to assess the models and discuss the possible impact of different parameters on each representations' advantages and disadvantages in Section V. After some theoretical calculations on the expected behavior of the different representations regarding storage requirements and query runtimes, we present some experimental results in Sections VI and VII and finally conclude the article in Section VIII.

II. Related Work

A literature review by Pelekis et al. gives a good impression of the historic development of spatio-temporal data models [1]. First approaches were actually temporal data models applied to spatial data. They used *snapshots* to represent complete scenarios of spatial data at certain temporal instants [2], [3]. Later, *timestamping* was used to record discrete changes/events concerning single geometries in databases [4]–[7]: a moving object's history is stored as a set of pairs (l, i), each containing an observation instant i and the observation l, which may be a point or a complex geometry. The main problems with this *point-location management* called model are, that it only defines discrete observations (interpolation and extrapolation is not possible), a significant precision/resource problem arises (if many observations are stored), and hence, query processing becomes inefficient.

The *Moving Objects Spatio-Temporal model* (MOST) and the *Future Temporal Logic* (FTL) query language aim at modeling and querying moving object data such that questions regarding the current and (near) future positions of moving objects can be answered; this task is often called *tracking* [8]–[11]. Moving objects are modelled by *dynamic attributes*, which continuously change as a function of time. Each function is represented as a point and a motion vector, where the point represents the object's last known location and the motion vector its last known direction and velocity, thus allowing for predictive queries.

The first abstract model to include continuous movement of objects — time dependent geometries, i.e. geometries described as a function of time — is [12]. It views objects and their complete history as abstract data types, calling them *moving objects*. This has been mapped to a discrete, implementable model [13]: A *sliced representation* is used, which decomposes the continuous changes of moving objects into fragments, that are modelled by "simple" functions. Finally, the algorithms for the spatio-temporal operations introduced in [14] have been improved by adding auxiliary *summary fields* to the moving data types [15]. The moving object model allows to represent the complete evolution of a moving object as a single attribute within an object-relational or other data model. An introduction to this model is given in Section III.

The free and open source extensible DBMS SECONDO [16]–[18] provides a native implementation of the moving object model and we use it as a platform for the experimental evaluations presented in this article. Another system implementing the moving objects model is STAU [19], which extends the commercial Oracle Spatial DBMS with two data cartridges; one for temporal types, a second for moving object data types. HERMES [20]–[22], a descendant of this prototype, still uses the sliced representation, but allows for using variable simple interpolation functions (currently: polynomial of first/second degree, square root of polynomial of second degree, constant function). The used Oracle10g DBMS allows complex user-defined types (here: the moving object data types) to physically represent data of variable size using *nested tables*. A moving object attribute points to a nested table stored in a so-called store table, managed by the DBMS. The nested table contains a set of units. Each unit is formed by a temporal interval and a set of *unit functions*, e.g. one for each coordinate, each actually consisting of a flag indicating an interpolation function and a vector of parameters to this function. This general concept is similar to the storage mechanism provided by SECONDO. One difference is, that SECONDO dynamically decides whether LOB data are stored within a special store table (called FLOB file) or within the table file itself. The lastest version of HERMES now capture both, historic and dynamic moving object queries. HERMES has also been migrated to the PostgreSQL/PostGIS DBMS [23].

Instead of implementing moving objects as data types within a DBMS, they can be conceptually modelled by a relation controlled by a spatially extended standard DBMS storing time slice data as tuples $(id, t_1, x_1, y_1, t_2, x_2, y_2)$, where id identifies a moving object [24]. This is similar to the Unit Representation we will introduce in Section IV-B. The same holds for the model proposed in [25], where moving objects trajectories are represented as sequences of tuples $(i, (x, y), t, b)$, x and y being coordinates of the i-th waypoint reached by the object at instant t. The flag b indicates, whether waypoints i and $i+1$ are connected (forming a well-defined piece of the trajectory). Otherwise, a definition gap exists, or the following waypoint starts a new trip.

In most moving objects databases (also in SECONDO), movements are usually supposed to be piecewise linear (i.e. linear interpolation between sample data points is used to reconstruct the temporal evolution). The objective is a reduction of the complexity for spatio-temporal predicate and operator implementations. Becker et al. [26] describe an approach to use non-linear approximation functions (e.g. B-splines) for moving object data. Experimental results show, that using B-slines increases the cost for evaluation of temporal functions by 30–50%, compared to linear interpolation. Similar methods have been proposed in order to gain more accurate trajectories and to compress the data [27].

In [28] the authors deal with the question how to represent data from multiple sensors: inserting the readings of all sensors into a common single table, or into one table per sensor, one table per observation instant, or maintaining a single table with one column per sensor, where tuples give the readings for all sensors at a time. In an experimental performance study regarding four queries analysing RoboCUP sensor data (ball and players' position and heading), they showed that there is no general best choice. This work becomes relevant, if several moving object trajectories (readings of different sensors) should be managed and queried together in a database (e.g. several moving object data type attributes per tuple).

To assess the performance of data representations, benchmarks are appropriate tools. Most benchmarks proposed for moving objects databases in the literature are insufficient to the requirement of this study. Arguments and a discussion of their usability are given in [29]. The BerlinMOD benchmark proposed in that article focuses on vehicle tracking data and perfectly fits our needs. Hence, we use this benchmark throughout this article. The BerlinMOD benchmark supports two different approaches of moving object semantics. The first one, called *object based approach* (OBA), handles the complete history of a vehicle as a single moving object attribute. The second, *trip based approach* (TBA), decomposes each moving object's journey into a series of trips with shorter histories. Trips belonging to a given moving object are identified using a *moving object identifier* (**moid**) unique to that object, which may be considered as a system generated integer attribute. In the TBA, each history (journey) is therefore represented as a set of $Trip$ attribute values, each being a moving point represented like the complete journey within the OBA. An introduction to the BerlinMOD scenario is presented in Section V-A.

Though we focus on unconstrained 2D moving objects in this article, it is possible to generalize our approach and apply it to any other time slicing representation for moving object history data, e.g. for periodically moving objects [30], semantic trajectories [31], or network constrained moving objects [32].

III. Revisiting Data Structures for MOD

An abstract data model for representing moving objects has been proposed in [14]. The model introduces a type constructor $\underline{moving}$, that creates types for moving object data when applied to a base data type, e.g. applying the type constructor to the spatial type $\underline{point}$, it returns a type for moving point data: $\underline{moving}(\underline{point})$ = $\underline{mpoint}$. There are also other type constructors, like $\underline{range}$ to create interval set types from a base type. While the representation of moving point data is a major contribution of that article, also other kinds of data are covered, e.g. the types $\underline{moving}(\underline{bool})$ = $\underline{mbool}$, $\underline{moving}(\underline{integer})$ = $\underline{mint}$, and $\underline{moving}(\underline{real})$ = $\underline{mreal}$ are defined, which are capable of storing temporal evolutions of the particular base types.

Based on this abstract data model, a discrete data model has been presented in [13], using a *sliced representation* for the development of a moving object, which is expressed by a set of *units* (Figure 1). Each unit describes the behavior of a moving object o for a certain time interval (t_i, t_j), $t_i \leq t_j$ disjoint from all the other units' definition time intervals. A "simple" function is used to approximate the concrete value at each instant of time during that interval. For an $\underline{int}$ or $\underline{bool}$ unit for example, a constant function is applied, for a $\underline{real}$ unit a quadratic polynomial function or its square root is used, and for a moving $\underline{point}$, the function is represented by the static $\underline{point}$ values taken at the unit's starting instant t_i and ending instant t_j. Linear interpolation is used to evaluate the function between these values for any instant t, $t_i \leq t \leq t_j$.

In this work, we will use a subset of the type system proposed in [13], as shown in Table I. We use seven kinds of types: **BASE** types, which represent static simple data (like $\underline{integer}$, $\underline{real}$); **SPATIAL** types, representing static 2-dimensional spatial objects ($\underline{point}$, $\underline{points}$ (=multipoint), $\underline{line}$, and $\underline{region}$), and 2/3-dimensional rectangles ($\underline{rect}$, $\underline{rect3}$); **TEMPORAL**, with type $\underline{instant}$ to represent single instants of time; **RANGE** types, providing a collection of disjoint intervals defined on a base type or temporal type — they correspond to the types created using the $\underline{range}$ type constructor from [14]; **INTIME** types, representing a concrete value at a certain temporal instant (an $(instant,\ value)$ pair); **UNIT** types, which represent time slices of temporal

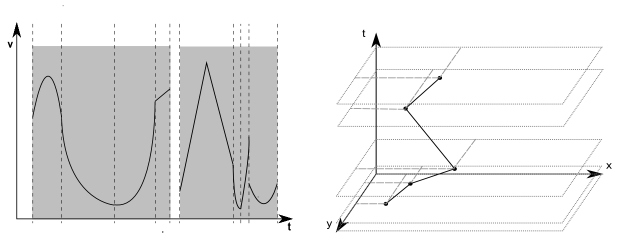

Fig. 1. Sliced Representations of a Moving Real and a Moving Point
The $\underline{mreal}$ shown in the left figure consists of 8 $\underline{ureal}$ units (time slices). The total definition time is shaded in gray. For instants outside the shaded area, the $\underline{mreal}$ is undefined. Each unit can be described by a simple function. The right figure depicts an $\underline{mpoint}$ consisting of 4 $\underline{upoint}$ units. Here, the lines between the 5 shown points represent the time slices, for which linear interpolation is used to approximate the object's trajectory.

BASE	=	$\{\underline{int}, \underline{real}, \underline{string}, \underline{bool}\}$
SPATIAL	=	$\{\underline{point}, \underline{line}, \underline{region}, \underline{rectangle2D}^{\dagger}, \underline{rectangle3D}^{\dagger}\}$
TEMPORAL	=	$\{\underline{instant}\}$
RANGE	=	$\{\underline{rint}, \underline{rreal}, \underline{rbool}, \underline{rinstant}^{\dagger}\}$
INTIME	=	$\{\underline{iint}, \underline{ireal}, \underline{ibool}, \underline{ipoint}, \underline{iregion}^{\dagger}\}$
UNIT	=	$\{\underline{uint}, \underline{ureal}, \underline{ubool}, \underline{upoint}, \underline{uregion}\}$
MOVING	=	$\{\underline{mint}, \underline{mreal}, \underline{mbool}, \underline{mpoint}, \underline{mregion}^{\dagger}\}$

TABLE I
TYPE SYSTEM

†InSECONDO, some types have different names: $\underline{periods}$ instead of $\underline{rinstant}$, $\underline{rect}$ instead of $\underline{rectangle2D}$, $\underline{rect3}$ instead of $\underline{rectangle3D}$, $\underline{intimeregion}$ instead of $\underline{iregion}$, $\underline{movingregion}$ instead of $\underline{mregion}$.

developments, as described in [13]; and **MOVING** types, which describe the complete history of a moving object by a collection of temporally disjoint units and correspond to the types created by the $\underline{moving}$ type constructor in [14];

Besides data types, we also require a set of operations on these types. Table II presents the subset of the operators introduced in [14] that is used within the BerlinMOD queries.

IV. MODELS

In this section, we address three different ways of representing the histories of moving object data (*MOD*) using unit types as primitives of a *sliced representation*, as introduced in Section III. To do so, we start using unit types as standalone data types. This is new, since in [14] unit types are not noted explicitly (though there is a single operator, **decompose**, that returns a set of "maximal continuous parts of temporal data types"), and in [13] the types introduced for kind **UNIT** are just used to formally define and implement the more complex **MOVING** types. However, values of **UNIT** types are conceptually capable to store simple temporal facts ($\underline{uint}$, $\underline{ubool}$, $\underline{ustring}$), functions ($\underline{ureal}$), and movements ($\underline{upoint}$, $\underline{uregion}$). We utilize this ability and assume that all unit types can be used as attribute types within relations managed by a DBMS. For our experiments, we have extended the SECONDO DBMS with all the unit types from Table I and operations on unit types according to Table II.[1] Hence, all requirements for using SECONDO during the experimental evaluations in Sections VI and VII are fulfilled.

[1]Implemented within the "TemporalUnitAlgebra" and the "TemporalAlgebra".

atinstant	restrict a mT/ uT to a certain instant
atperiods	restrict a mT/ uT to a certain rinstant
at	restrict a mT/ uT to the time intervals where it takes a certain T value
intersection	for a pair of values of compatible types, compute the intersection
inside	test, if the first value is contained in the second one
intersects	test, whether two values of compatible types intersect
trajectory	compute the 2D-spatial projection of a mT/ uT
deftime	return the definition time of a mT/ uT as an rinstant
speed	return the speed of an mpoint/ upoint as an mreal/ ureal
distance	compute the distance of two spatial objects
inst	return an iT value's instant
val	return an iT values's value of type T
initial	for a mT/ uT, return the iT corresponding to the initial instant
final	for a mT/ uT, return the iT corresponding to the final instant

TABLE II
A SELECTION OF OPERATIONS AS USED BY BERLINMOD

T ∈ BASE ∪ SPATIAL. rT, iT, mT and uT stand for according range, intime, moving, resp. unit types based on type T. The definition of "compatible" relies on common sense. Where applicable, operators will return moving types (e.g. **inside**: *mpoint* × *region* → *mbool*). Several operators (e.g. **at**, **atperiods**, **intersection**) return a set of units when applied to an unit type. A more detailed description of these operators and a formal definition of their semantics can be found in [14].

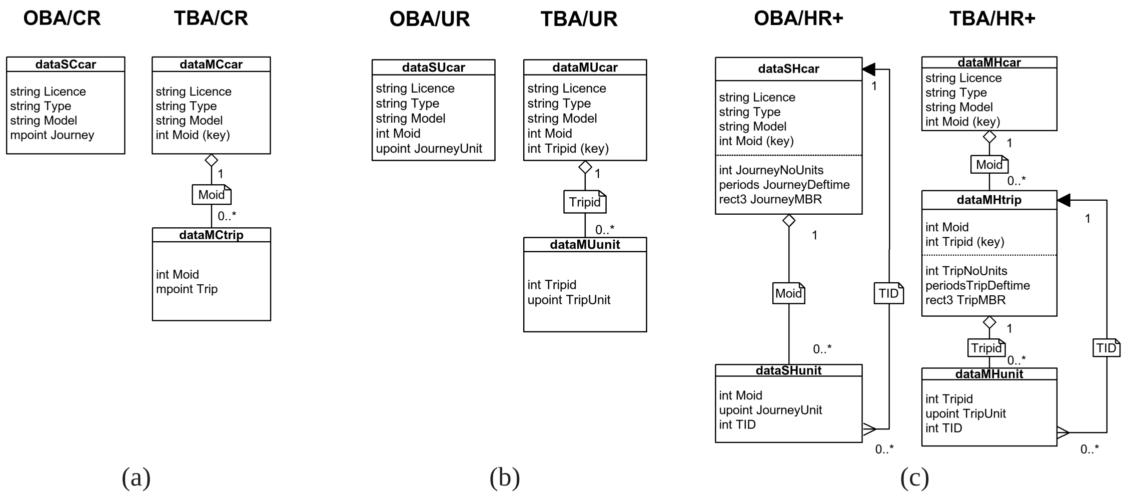

Fig. 2. BerlinMOD conceptual database schemas: Compact (a), Unit (b), and Hybrid Representation (c).

A. Compact Representation

The **compact representation (CR)** is the direct application of the MOD model from Section III: The complete history of a moving object is encapsulated by an abstract data type. Logically, all units of an object's history are organized into an array, that is embedded into the object representation. Hence, such a moving object can be used as an atomic attribute value within a tuple. Physically, the array of units is stored in a distinct file (LOB file) and only reference to that file and some summary information is kept within the object itself. Internally, units of each moving object are sorted by their definition time interval and hence allow for efficient access methods, like binary search. Additional aggregated information is stored in so-called *summary fields*: the number of components (units), definition time (interval from the first unit's beginning instant to the last unit's ending instant), duration (sum of interval lengths, where the object is defined), and further type specific summary information, as minimum and maximum value (for mint, mreal, mbool), or MBR (for the spatio-temporal types $\underline{mpoint}$, $\underline{mregion}$). The summary fields are directly kept within a moving object's root record (i.e. they are embedded into the tuple representation and need not to be loaded explicitly). They can be used in a preceding filtering step to select candidates for expensive geometric tests in selections and joins, which would otherwise require the object's units to be loaded from their external file. We assume, that at least the number of components, and for spatio-temporal moving data, the object's spatio-temporal MBR are saved as summary fields, which is the case in SECONDO.

Throughout this article, we will use both the moving object semantics (OBA and TBA) BerlinMOD [29] supports. Representing the OBA using the CR (abbreviated OBA/CR) is straightforward: Since complete histories of moving object data can be stored within single attributes, there is only a single relation `dataSCcar` containing one tuple per vehicle. The tuple contains a key attribute $licence$, some more attributes, plus one attribute of type $\underline{mpoint}$ called $Journey$, that contains the complete movement history for that object. To capture the TBA semantics in the CR (TBA/CR), each history (journey) is represented as a set of $Trip$ values, each being represented in a compact manner as a moving point. These history data are stored within a *trip relation* `dataMCtrip`, whereas the attributes regarding the complete object are stored within a *root relation* `dataMCcar`. Figure 2 (a) shows what the main data objects used in this benchmark look like, when using the CR to represent the MOD generated by BerlinMOD for both, the OBA and TBA. Note the 1:n relation between the root relation `dataMCcar` and the trip relation `dataMCtrip`; the additional attribute $Moid$ identifies all tuples containing history data for trips of the same moving object and can be used to create an index. There is also a functional dependency $Licence \rightarrow Moid$.

B. Unit Representation

In the **unit representation (UR)**, every moving object is represented by a set of tuples within a relation. In the OBA/UR, each tuple contains a single unit of a moving object (i.e. series of single units rather than complete histories are stored within the relation), a *moving object identifier* (**moid**, a key unique to the individual object, to which the unit belongs), and copies of all other attributes belonging to the moving object. This means, that all attributes get replicated once per unit. The moids allow to select the units forming the history of a certain moving object. If more than a single moving type attribute is used, the units need to be modified to ensure that the units contained by a tuple all have exactly the same definition time interval. This again increases the storage requirements, since original units are split, whenever definition time intervals of two different attributes' units overlap. Therefore, we restrict UR relations to contain only one moving object. Summary fields are not supported by this representation model.

For the OBA, only a single relation `dataSUcar` is used. Compared with the OBA/CR, the added attribute $Moid$ is used to identify all units belonging to the history of the same moving object. In the TBA/UR, each vehicle's journey is subdivided into several trips, each of which is handled as an own moving object. The units are kept within a separate *unit relation* `dataMUunit`, whereas the root records for each trip are kept within the *root relation* `dataMUcar`. The moid groups the tuples within the root relation belonging to a single vehicle. An additional unique trip identifier *Tripid* is added to all tuples within both relations. It connects each tuple of the root relation with the set of tuples within the unit relation whose units form the trip's history. The TBA/UR looks very similar to the TBA/CR schema; just instead of $\underline{mpoint}$ values, $\underline{upoint}$ values are stored, and — as the moid is used to group trips into journeys — an identifier $Tripid$ is used to form trips from sets of units. The main data objects for the BerlinMOD benchmark using the UR are shown in Figure 2 (b).

C. Hybrid Representation

In the **hybrid representation (HR)**, within an original tuple, every single moving object gets replaced by a unique *moving object identifier* (**moid**). Thus, we receive the *root relation*. As the moid is unique to every single moving object, it can be used to identify a moving object's units, which are stored within so-called *unit relations* (one per column/attribute containing a moving object), whose tuples contain each a unit of a moving object together with its moid and the *tuple id* (**tid**) of the referencing tuple in the root relation. This tid is a system-generated tuple identifier, that allows to access a tuple within the root relation in a direct and quick manner, i.e. without using an index. In comparison with the UR, we thus avoid the replication of non-moving attributes. Also, the HR doesn't suffer from the inflation inflicted by splitting units as observed with the UR when more than a single moving object is contained in a single tuple of the main relation. However, handling the additional unit relations increases the complexity of managing stored moving objects. While the moid can be used to find the related units in the unit relations, the tids can be used to identify the referring tuple within the main relation (i.e. the tuple "containing" the unit). In an attempt to save administrative overhead, one could use the main relation tids as moids of each contained moving object and omit the tid columns within the unit relation, but this would increase the complexity when creating result relations.

Modelling the OBA semantics, the HR database schema looks very similar to that for OBA/UR: The *root relation* `dataSHcar` contains one tuple per vehicle. $Moid$ is used to reference tuples in the *unit relation* `dataSHunit` that contain units forming the vehicle's MOD history. The tuple identifier TID points from each unit tuple back to the according root tuple. This allows to quickly access the root tuple without using an index. The HR for trip-based semantics is clearly the most complex of the representations proposed here, as is shown by Figure 2 (c). The TBA/HR schema has three levels. The uppermost *root relation* `dataHMcar` contains one tuple per vehicle. The intermediate *trip relation* `dataMHtrip` contains one tuple per trip belonging to the history of a vehicle from the root relation. $Moid$ is used to join these data. The lowermost *unit relation* `dataMHunit` contains the units forming all histories of vehicles from `dataMHcar`. The units are grouped by $Tripid$ to form trips. The tuple identifier TID refers from each unit tuple directly to its "containing" tuple in `dataMHtrip`.

For the HR, we further propose two variants regarding the presence of summary fields:

1) With Summary Fields (HR$^{\oplus}$): In this variant of the hybrid representation, summary information is kept as additional attributes within the tuples of the root relation (OBA), resp. the trip relation (TBA).

2) Without Summary Fields (HR$^{\ominus}$): Here, we abandon the information stored in summary fields. This variant we expect to benefit in the case of updates, as no summary data is needed to be computed on occasion of an update.

In Figure 2 (c), both HR variants are displayed all at once. For the variant HR$^{\ominus}$, we just drop the summary fields $TripNoUnit$, $TripDeftime$, and $TripMBR$ from relation `dataSHcar`, resp. `dataMHtrip`. In the figures representing these relations, the summary fields have been separated from the other attributes by a dotted line.

V. Evaluation Database

In the following sections, we will evaluate the performance of the proposed data models from section IV. We will use the BerlinMOD/R benchmark [29] for a standardized comparison of the results. We now briefly describe the BerlinMOD/R benchmark and motivate our choice.

A. Scenario

We assume, that a MOD database (in its compact representation) should contain at least one relation with a large cardinality (≥ 1000)[2], containing exactly one attribute of a moving spatial type per tuple. The moving objects stored should have a long history in average, i.e. their history consists of a large number of units. At scale factor 1.0, the BerlinMOD scenario tracks the motions of 2,000 vehicles within Berlin simulated for a period of 28 days.[3] 80% of these cars are “passenger cars”, 10% are “trucks” and the remaining 10% “buses”. For each car, commuting trips between the owner’s home and place of work are created, and additional “leisure time” trips are added. The simulation captures a vehicle’s positions whenever its velocity or direction changes significantly, every 2 seconds otherwise. With these assumptions, we get an average of about 963 units for each car and day, or 53,926,394 units in total at scalefactor 1.0.

B. Database Schema

The original BerlinMOD schema is adapted to fit our needs, resulting in the following database schema for the CR:

OBA only:

`dataSCcar`: **relation**$\{Licence$: $\underline{string}$, $Model$: $\underline{string}$, $Type$: $\underline{string}$, $Journey$: $\underline{mpoint}\}$ — relation of vehicle descriptions (car type, car model and licence plate number), including the complete position history as a single $\underline{mpoint}$ value per vehicle, where $Licence$ is a key.

TBA only:

`dataMCcar`: **relation**$\{Licence$: $\underline{string}$, $Model$: $\underline{string}$, $Type$: $\underline{string}$, $Moid$: $\underline{int}\}$ — relation of all vehicle descriptions (without position history).

`dataMCtrip`: **relation**$\{Moid$: $\underline{string}$, $Trip$: $\underline{mpoint}\}$ — relation containing all vehicles’ movements and pauses as single trips ($\underline{mpoint}$ values).

Here, $\{Moid\}$ is a key/ foreign key for `dataMCcar` and $\{Moid, Trip\}$ is a key for `dataMCtrip`.

[2]While 1000 is not considered to be a “large number” of tuples in a database system, we want to point out that we are talking about 1000 tuples, each containing a huge amount of data when it comes to long-term observation of moving objects. Each tuple may then contain tens of thousands of single observations.

[3]We will in the sequel refer to different scale factors by the notion of “$@SF$”, where SF is the scale factor.

Common Database Objects (OBA and TBA): A set of relations is used to control the queries by providing query points, periods, licences and regions. All these relations start with the prefix "`Query`".

This base schema can directly be used for the CR, whose main relations for the MOD are visualized in Figure 2 (a). From this basic database schema, the database schemas for the UR and HR can easily be deduced, having main relations according to Figure 2 (b) and (c).

C. Parameters

For setting up the experimental evaluation of different data representations, the analysis of variables influencing the performance is a preliminary step. It is important for choosing a realistic benchmarking workload and for the interpretation of the results. In this subsection we consider which factors should have what kind of impact on the performance of the described data models. We will also show, how these parameters are set up in the BerlinMOD benchmark.

1) Semantics of Moving Object Data: Moving Object Data for vehicles can be viewed in two different ways: The object-based (OBA), and the trip-based (TBA) approach. The first one keeps the complete history of the object together. The second approach rather observes a vehicle's single trips, which are kept separately in the database, together with a key (moid) that allows one to select all trips belonging to a single vehicle. Both views have advantages and disadvantages. While in the OBA semantics are always clear, in the TBA one needs to specify how to distinguish between subsequent "trips". The TBA applies some form of trajectory splitting [33] and thus reduces the size of MBRs to index and therefore also index selectivities. When interpreting the three presented models (CR, UR, HR) to be some kind of "archetypes" of representing MOD, the TBA diminishes the characteristics of these three models. E.g. in the TBA additional indexes to lookup all trips belonging to a single vehicle are required, because original keys (like "Licence" in BerlinMOD) loose their key property. The TBA/CR therefore looses its object semantic property, becoming more similar to the OBA/HR. As BerlinMOD provides data and queries for both, OBA and TBA, and since experiments [29] discovered several advantages of the TBA/CR regarding certain types of queries, we will cover both semantics in our experiments.

2) Amount of Objects: As many reasonable MOD applications will track data of many moving objects, to demonstrate the gain in performance achieved by using index structures, and to justify for using indexes on moid or primary keys, we choose to observe 447, 894, and 2,000 objects from the BerlinMOD data (at scale factors 0.05, 0.2, and 1.0) as "large" numbers of recorded object histories for the comparison of MOD representations.

3) Length of Histories: At scale factor 1.0, the BerlinMOD benchmark generates objects having an average length of 26,963 units per history. With that amount, we hope to avoid discrimination of any of the representations examined: Choosing too short histories results in an approximation of the CR to the UR; too large lengths will result in a growth of overhead from replicated data especially within the UR, and overly increase the answer times for spatial access to MOD in the CR. One could argue, that MOD should be compacted to contain only "interesting" data and therefore histories be shorter (e.g. only several 10 or 100 units, semantic trajectories [31]). But that would require a-priori knowledge of the meaning of "interesting". However, our model inherently compacts stored data, as "simple" changes (constant values, linear movements, etc.) can be expressed within a single unit, even for long periods of time. In BerlinMOD, the length l of generated MOD histories depends on the scale factor SF, as

$l \sim \sqrt{SF}$. We will use scale factors 0.05, 0.2, and 1.0 in our experiments. It is important to mention that the scale factor influences the amount of trips generated, not the average number of units per trip. This results in larger sizes of attribute $Journey$ within OBA/CR, but in higher cardinalities for all other representations.

4) Index Structures: In our experiments, we will use R-Trees [34] to index the MOD — 2D-R-Trees for spatial and temporal indexes, and 3D-R-Trees for spatio-temporal indexes. This choice is due to the availability of index structures in our database platform, namely SECONDO [16], [17]. Of course, there exist various specialized structures for indexing trajectories (e.g. the TB-tree [35]). But as our goal is to compare *general* properties of the three MOD representations, we think, that results can be transferred to more convenient index structures later on. To allow for fast access to stored MOD, we assume that for each representation model the following indexes are available for all spatial, temporal, and spatio-temporal attributes within the test databases:
(a) **Object Index (B-Tree)**: Key $\in \{Moid, Tripid, Licence\}$ — provide quick access to tuples holding the object data (CR), resp. units from the object's history (UR, HR).
(b) **Spatio-Temporal Index (3D-R-tree)**: Keys are 3D-MBRs of the indexed attribute — allows quick access to all tuples containing objects/ units whose 3D-MBR intersects a 3D-query window. For the CR, *multiple entry indexing* [36] is applied, i.e for each tuple within relation `dataSCcar`, its $\underline{mpoint}$ $Journey$ is decomposed into all contained units $u_1, \cdots, u_n$, and for each unit u_i a pair $(MBR(u_i),\ tid)$, where tid is the tuple identifier of the according tuple in `dataSCcar`, is inserted into the R-Tree.[4]
(c) **Temporal Index (2D-R-Tree)**: Keys are 2D-MBRs created from the indexed attribute's deftime — gives quick access to all tuples containing objects/ units whose deftime intersects a query instant or query time interval. Due to the index structures available in SECONDO, we apply a transformation approach to map temporal queries to spatial ones: Assuming, that each $\underline{instant}$ I can be translated into a corresponding $\underline{real}$ value using a function $\mathbf{getreal}$, it is possible to translate a temporal range value R, i.e. an $\underline{rinstant}$ value, into a 2D-point $(\mathbf{getreal}(\mathbf{initial}(R)),\ \mathbf{getreal}(\mathbf{final}(R)))$. Such a point can be used to generate keys for the temporal R-Tree index. To query such an index, the same transformation can be used to lookup temporal ranges in the index. To perform a temporal point query, the instant I is translated to the point P corresponding to the temporal interval $[I,\ I]$. If the $\underline{real}$ representation for the earliest, resp. latest representable instant is denoted by $BEGIN\_OF\_TIME$, resp. END_OF_TIME, and point S=($BEGIN\_OF\_TIME$,END_OF_TIME), the MBR of $\{P,\ S\}$ can be used to retrieve all temporal intervals possibly containing the query instant from the temporal index. For the CR, multiple entry indexing is used.
(d) **Spatial Index (2D-R-tree)**: Keys are 2D-MBRs of the indexed attribute resp. its MBR's spatial projection — grants quick access to all tuples whose object/ unit 2D-MBR intersects a given query rectangle. Again, multiple entry indexing is used for the CR.

Using R-Trees to index 2D-MOD with 3D-MBRs as keys raises a well-known problem: The temporal coordinates are usually represented on different scales than the spatial coordinates. Hence, direct use of these MBRs during R-Tree construction would lead to skewed indexes with unbalanced or even degraded index performance. The same may happen in the 2D case, if the sizes of the ranges regarding both dimensions are very different. As a simple solution to this problem, we normalize all spatial and spatio-temporal MBRs, so that all dimensions are treated

[4]Also see Section VI-A2 for a short example on multiple entry indexing.

fairly equal: Basically, each dimension is scaled such that the average interval size of all MBRs is the same for all dimensions.

5) Data Distribution: Obviously, the (spatial and temporal) distribution of data is crucial for index performance and therefore for query performance. Uniform distributions allow for easy analyses and are thus comfortable to the researcher. The same holds for Gaussian data distribution. However, both distribution schemas are not “natural” and have a limited expressiveness concerning real applications. Hence, we prefer to experiment with data being as “real” as possible. The BerlinMOD benchmark generates and uses representative, non-uniformly distributed data.

6) Universe and Query Window Dimensions: The universe, which is defined as the union of all possible objects’ 3D-MBRs, shall be large compared to the size of single query ranges. This reflects that often we collect huge amounts of data, but single queries can be restricted to relatively small ranges within the universe. BerlinMOD/R performs instant and point queries, as well as small and large range queries. This helps to balance the performance of the different temporal, spatial and spatio-temporal index types.

7) Standard Attributes and Tuple Size: Usually, queries are not concerning solely spatial, temporal or spatio-temporal aspects. Rather, additional information represented using standard data types is used. To demonstrate the properties of the different representations, we can use the additional attributes given to each vehicle in the BerlinMOD data. Namely, the string attribute $Licence$ is a key, and there are two more non-key string attributes, $Type$ (“bus”, “truck” or “passenger”) and $Model$ (manufacturer) of the car. Thus, tuples can be selected by different criteria (including non-indexed attributes) and the tuple size is increased without changing the size of the MOD. One could restrict the relation with the MOD to contain only a key as an additional attribute, and all other attributes to be stored separately in a second relation. This would transform each representation into something similar to the HR — but again, we want to demonstrate the general differences between the three representations.

8) Clustering: Within the relations, MOD can be stored clustered, i.e. sorted by some meaningful criterion. BerlinMOD creates data clustered primarily by object-id, with definition time being the secondary criterion. This is the only feasible way for the CR. For the UR and HR it makes sense to consider different orderings, e.g. storing the units by the their definition time, or spatial or spatio-temporal MBR (using Z-order). This might strongly affect the number of page reads required to process certain queries. Clearly, each clustering scheme will favor a certain kind of queries and discriminate some others. We will restrict our experiments to clustering relations by (i) first object-id and second definition time (**ID/TMP** schema), and by (ii) the spatio-temporal MBR of the MOD (**SPTMP** schema). This choice was made in order to keep the number of experiments manageable.

D. Queries

For our analyses, we simply use the 17 BerlinMOD/R benchmark queries, i.e. the range and point query subset, as proposed in [29]. The query set BerlinMOD/NN, focussing on complex nearest neighbor queries, is not considered, as SECONDO’s abilities to process these queries effectively — by now — depend on a certain UR like data representation, so these queries cannot contribute to the results of a fair comparison of the different data representations. The BerlinMOD/R queries can also be found in Appendix A.

1) Query Classification: To formulate our expectations and results within Sections VI and VII, we use the query categorization introduced in [29]. The categorization classifies query types as a combination of five dimensions: (a) **Object Identity** $\in$ {known, unknown} — Whether a query starts with a known object "Does object X ..." or without "Which objects do ...". (b) **Dimension** $\in$ {standard, temporal, spatial, spatio-temporal}) — This criterion refers to the dimension(s) used in the query. (c) **Query Interval** $\in$ {point, range, unbounded} — This property determines the presence/size of the query interval. (d) **Condition Type** $\in$ {single object, object relations} — Whether relationships between objects are subject of the query (requiring a join) or not ("single object"). (e) **Aggregation** $\in$ {aggregation, no aggregation} — Indicates whether the result is computed by some kind of aggregation. This property may depend on which approach is used ("(no) aggregation").

The BerlinMOD/R benchmark queries cover the 14 most interesting of all possible 96 combinations. Non spatio-temporal queries are covered twice (Q1–2); the combination (unknown, spatio-temporal, range, single, no aggr) is covered by three queries (Q13–15).

We further generalize these combinations into two more general and obvious categories of queries: (a) **Standard queries** — any query that does not involve a MOD attribute within its where clause and does not perform calculations on any MOD-attribute. From the BerlinMOD/R queries, only Q1 and Q2 belong to this category. (b) **MOD queries** — any query, that does involve a MOD attribute in its where clause or somehow calculates a MOD-attribute within the select clause. In BerlinMOD/R, Q3–17 belong to this category.

VI. Storage Requirements

In this section, we first formulate expectations on the relative storage performance by calculating the theoretically required disk space for all BerlinMOD relations and indexes within the different representations. Then, we compare the theoretical results to practical from experiments using the different models within the SECONDO DBMS. We have evaluated the following proposed representations for MOD following both, the OBA and TBA: CR, UR, HR$^{\oplus}$. To this end, we have used BerlinMOD to generate MOD at scale factors 0.05, 0.2, and 1.0 and then created all representations addressed in the previous sections. We also translated the benchmark queries from BerlinMOD/R to fit the different representations and started the benchmark for each representation and scale factor using the SECONDO DBMS. For the experiments, we used a standard PC configured as shown in Table III. As mentioned before (see Section V-A), we will use the notation of "$@SF$" in the sequel. It can be read as "at scale factor SF".

A. Expected Storage Requirements

In this subsection, we investigate how the three representations influence the storage size for the BerlinMOD benchmark data from a theoretical point of view, regarding relations as well as as the different indexes which could be useful in queries. We don't regard optimizer related information like histograms and so on.

For our computations, we use a hypothetical database similar to the concrete one used in the experiments. The main differences are: (a) All cars have the same (average) number of trips and units, and (b) computations are independent of the database system (assuming an optimal storage and query behavior). Otherwise, we use the original statistics on the BerlinMOD data @0.05, @0.2, and @1.0 (Table IV). For example, the table indicates, that @1.0, 2,000 vehicles with 292,693 trips (for the TBA), and 53,926,394 units are created by the BerlinMOD data generator.

CPU:	Intel Core 2 DUO 2.4 GHz
Memory:	2 GB
HDD:	2 x 500 GB, RAID 0
OS:	Open SuSe 10.2 (Kernel 2.6.18.2-34-default)
DBMS:	SECONDO 2.8.4

TABLE III
CONFIGURATION OF THE EVALUATION PLATFORM

	@0.05	@0.2	@1.0
Days	6	13	28
Vehicles	447	894	2,000
Trips	15,049	62,893	292,693
Units	2,744,069	11,658,013	53,926,394
Avg. Units/Vehicle	6138.857	13040.283	26963.197
Avg. Units/Trip	182.340	185.360	184.240
Avg. Trips/Vehicle	33.667	70.353	146.347

TABLE IV
BASIC STATISTICS ON BERLINMOD DATA

1) Relations: To determine the size of the relations, we start with calculating field sizes for all attributes. For most fields, as <u>*int*</u> or <u>*string*</u> attributes, the sizes are constant. Some types need more explanation: Each <u>*upoint*</u> value consists of a time interval, a starting and an ending position. For each position, we need two coordinates represented by the `double` type. We model time as a discrete representation of the $\mathbb{R}$. For a temporal resolution of 1 millisecond, we represent an <u>*instant*</u> value by a `long int` (8 bytes). Two `boolean` flags (1 byte each) indicate whether the starting and the ending boundary of the interval belong to the interval or not. Summarized we get $4 \cdot 8 + 2 \cdot 8 + 2 \cdot 1 = 50$ bytes per unit. The <u>*mpoint*</u> data type consists of a constant part containing data management and summary information, plus a variable part for all units contained — while both attributes, $Trip$ and $Journey$, are of type <u>*mpoint*</u>, they contain different numbers of units. The constant part consists of the summary fields; we have the number of contained components (`int`, 4 bytes), the deftime (two <u>*instant*</u>s, 8 bytes each), the duration (type <u>*duration*</u>, 8 bytes), and the MBR (a <u>*rectangle3D*</u> represented using 6 `double` values, 8 bytes each), giving a total of 76 bytes for the fixed-sized part. For the variable part, each contained unit contributes with 50 additional bytes, as calculated above. Obviously, the clustering of units within the unit relations for the UR and HR has no effect on the storage requirements.

The sizes for all attributes used within the relations for the representations introduced in Section V-B are shown in Figure 4. The upper part of the table shows the attributes with constant sizes. The lower part shows the varying average size of attributes $Journey$ and $Trip$, which depends on the average length of the vehicles' histories. Remember, that the size of $Trip$ does not depend on the scaling factor: With increasing scaling factor, within the vehicles' histories only the amount of trips — but not their length or duration — increases.

Now we can compute the tuple sizes for all used relations in a straightforward way. For the relations' cardinalities, we use the statistics on the number of vehicles, trips, and units as presented in Table IV. The cardinalities, tuple sizes, and total memory sizes for all the relations in Figure 2 are given in Table V. From that, we calculate the total relation sizes for all representations (see Table VI). As one could easily expect, the CR is the smallest one, and the UR the largest one in terms of disk space required.

2) Indexes: As in SECONDO only secondary indexes are available, we only provide calculations for secondary indexes. Secondary indexes yield tuple identifiers referring to tuples within the main relation (CR), resp. a unit or trip relation (UR, HR).

For all three representations, the temporal, spatial and spatio-temporal indexes (see V-C4) are created using multiple entry indexing [36]. This is, for each unit, a key is inserted into the

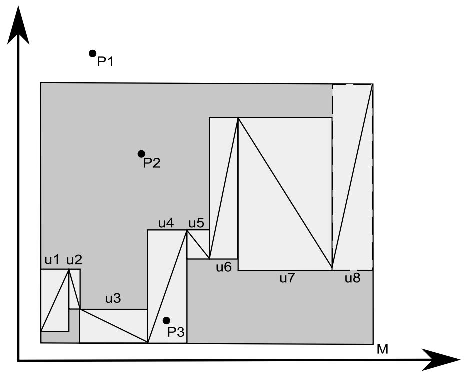

Fig. 3. Multiple entry indexing

Attribute	**Type**	**Size** [Byte]
Licence	string	12
Model	string	20
Type	string	10
Moid	int	4
Tripid	int	4
JourneyUnit	upoint	50
TripUnit	upoint	50
TripNoUnits	int	4
TripDeftime	rinstant	16
TripMBR	rect3	48
Tid	int	4
Journey	mpoint	
@0.05		307018.85
@0.2		652090.15
@1.0		1348235.85
Trip	mpoint	
@0.05		9343.75
@0.2		9343.75
@1.0		9288.08

Fig. 4. Hypothetical Attribute Sizes

	Cardinalities			**Tuplesizes [B]**			**Relation Sizes [MB]**		
Relation	**@0.05**	**@0.2**	**@1.0**	**@0.05**	**@0.2**	**@1.0**	**@0.05**	**@0.2**	**@1.0**
Calculated Sizes									
dataSCcar	447	894	2000	307018.850	652132.150	1348277.850	130.880	555.998	2571.636
dataMCcar	447	894	2000	46.000	46.000	46.000	0.020	0.039	0.088
dataMCtrip	15049	62893	292693	9193.024	9347.752	9292.077	131.937	560.673	2593.733
dataSUcar	2744069	11658013	53926394	96.000	96.000	96.000	251.227	1067.323	4937.109
dataMUcar	15049	62893	292693	50.000	50.000	50.000	0.718	2.999	13.957
dataMUunit	2744069	11658013	53926394	54.000	54.000	54.000	141.315	600.369	2777.124
dataSHcar$^{\oplus}$	447	894	2000	114.000	114.000	114.000	0.049	0.097	0.217
dataSHcar$^{\ominus}$	447	894	2000	46.000	46.000	46.000	0.020	0.039	0.088
dataSHunit	2744069	11658013	53926394	58.000	58.000	58.000	151.783	644.841	2982.837
dataMHcar	447	894	2000	46.000	46.000	46.000	0.020	0.039	0.088
dataMHtrip$^{\oplus}$	15049	62893	292693	76.000	76.000	76.000	1.091	4.558	21.214
dataMHtrip$^{\ominus}$	15049	62893	292693	8.000	8.000	8.000	0.115	0.480	2.233
dataMHunit	2744069	11658013	53926394	58.000	58.000	58.000	151.783	644.841	2982.837
Observed Sizes									
dataSCcar	447	894	2000	1035912.233	2188675.007	4474039.30	315.375	1337.914	8533.553
dataMCcar	447	894	2000	180.000	180.000	168.00	0.094	0.176	0.320
dataMCtrip	15038	62879	292669	21267.471	21646.254	21320.61	330.730	1407.137	5950.815
dataSUcar	2742033	11670697	53388403	296.000	296.000	296.00	825.977	3515.492	15070.884
dataMUcar	15038	62879	292669	180.000	180.000	180.00	2.816	11.738	50.240
dataMUunit	2742033	11670697	53388403	128.000	128.000	128.00	357.930	1523.391	6517.139
dataSHcar$^{\oplus}$	447	894	2000	348.000	348.000	348.00	0.168	0.328	0.664
dataSHcar$^{\ominus}$	—	—	—	—	—	—	—	—	—
dataSHunit	2742033	11670697	53388403	144.000	144.000	144.00	397.695	1692.652	7331.781
dataMHcar	447	894	2000	168.000	168.000	168.00	0.086	0.160	0.320
dataMHtrip$^{\oplus}$	15038	62879	292669	192.000	192.000	192.00	2.953	12.320	57.316
dataMHtrip$^{\ominus}$	—	—	—	—	—	—	—	—	—
dataMHunit	2742033	11670697	53388403	144.000	144.000	144.00	397.695	1692.652	7331.781

TABLE V
CALCULATED VS. OBSERVED CARDINALITIES, TUPLE SIZES, AND RELATION SIZES FOR BERLINMOD

Representation	Total Relations [MB]			Total Indexes [MB]			Total Database [MB]		
	@0.05	@0.2	@1.0	@0.05	@0.2	@1.0	@0.05	@0.2	@1.0
Calculated Sizes									
OBA/CR	131	556	2572	376	2055	9075	506	2611	11647
OBA/UR	251	1067	4937	463	2558	10888	714	3625	15825
OBA/HR$^{\oplus}$	152	645	2983	406	2177	9634	557	2822	12617
OBA/HR$^{\ominus}$	152	645	2983	405	2177	9634	557	2822	12617
TBA/CR	132	561	2594	376	2056	9080	508	2617	11674
TBA/UR	142	603	2791	406	2181	9650	548	2784	12441
TBA/HR$^{\oplus}$	153	649	3004	408	2187	9691	561	2836	12695
TBA/HR$^{\ominus}$	152	645	2985	406	2179	9643	558	2824	12628
Observed Sizes									
OBA/CR	315	1338	6112	599	2522	12573	914	3859	18685
OBA/UR	826	3515	16082	657	2756	13623	1483	6272	29705
OBA/HR$^{\oplus}$	398	1693	7744	628	2639	13098	1026	4332	20842
OBA/HR$^{\ominus}$	—	—	—	—	—	—	—	—	—
TBA/CR	331	1407	6447	599	2524	12580	930	3931	19027
TBA/UR	361	1536	7026	652	2745	13588	1013	4280	20614
TBA/HR$^{\oplus}$	401	1705	7801	659	2771	13665	1060	4476	21465
TBA/HR$^{\ominus}$	—	—	—	—	—	—	—	—	—

TABLE VI
CALCULATED VS. OBSERVED TOTAL RELATION, INDEX, AND DATABASE SIZES FOR BERLINMOD

indexes. If one would build, e.g. the indexes for the CR, based on the objects' total MBRs, this would result in drastically smaller indexes, which would in turn have a bad performance, since almost all MBRs would overlap, producing a drastically high percentage of false alarms at each index access. The idea of trajectory splitting for multiple entry indexing is visualized in Figure 3: A moving point M consists of 8 units $u_1, \ldots, u_8$. The figure shows $trajectory(M)$ and compares the 2D-MBR of the complete moving object with those for the single units. A spatial index built with MBR(M) returns the whole object $\{M\}$ for query points P_2 and P_3, while an index built using the *multiple entry indexing* policy will create separate entries for MBR$(u_i), 1 \leq i \leq 8$. The index returns $\emptyset$ for query point P_2 and $\{u_4\}$ for P_3.

We further assume, that an R*-tree [37] implementation is used for 2D (3D)-R-trees, and set the page size to the widely used standard of 4kB. Each R-tree entry contains an MBR (16 bytes per dimension) and a page/ tuple identifier (4 bytes), making up 36 bytes (2D), resp. 52 bytes (3D) per entry. This allows for 113 (78) entries per page. We force node fillings of not less than 40% and therefore use R-trees of orders (46, 113) for 2D-R-trees, and (32, 78) for 3D-R-trees. Assuming an average filling degree of 69%, we get an average fan-out of $f = 77$ ($f = 53$), resulting in height $h = \log_f N$. For N, we have 53,926,000 units resp. 292,693 trips or 2,000 vehicles. Assuming a B$^+$-tree implementation is used for B-trees and page size = 4kB, we get a maximum fan-out of 512 for attribute $TripMoid$ (256 for $Licence$). With an average filling degree of 74% for inner nodes and leaves, we get an average fan-out of $f = 378$ ($f = 189$).

The calculated index sizes (in terms of 4kB pages and height of the trees) for the different relations and scale factors are presented in Table VII. The total index sizes (in MB) for the different representations are listed in Table VI.

3) Databases: The expected total database size for each representation is calculated as the sum of its total relation size and its total index size. The results are presented in Table VI.

	Relation	**Calculated [pages/height]**					**Observed [pages/height]**					
		R-tree Indexes		**B-tree Indexes**			**R-tree Indexes**			**B-tree Indexes**		
		2D	3D	Moid	Tripid	Licence	2D spat.	2D temp.	3D	Moid	Tripid	Licence
@0.05	dataSCcar	41567/3	54584/3	—	—	3/1	49478/3	45592/3	58299/4	—	—	6/2
	dataSUcar	41567/3	54584/3	7638/2	—	14708/2	49478/3	45592/3	58299/4	7429/2	—	7432/2
	dataSHcar$^{\oplus}$	6/1	9/1	2/1	—	3/1	15/1	10/1	17/1	5/2	—	6/2
	dataSHcar$^{\ominus}$	—	—	2/1	—	3/1	—	—	—	—	—	—
	dataSHunit	41567/3	54584/3	7638/2	—	—	49478/3	45592/3	58299/4	7429/2	—	—
	dataMCcar	—	—	2/1	—	3/1	—	—	—	5/2	—	6/2
	dataMCtrip	41567/3	54584/3	40/1	—	—	49478/3	45592/3	58299/4	71/2	—	—
	dataMUcar	—	—	40/1	40/1	80/1	—	—	—	71/2	77/2	92/2
	dataMUunit	41567/3	54584/3	—	7638/2	—	49478/3	45592/3	58299/4	—	13265/3	—
	dataMHcar	—	—	—	—	3/1	—	—	—	5/2	—	6/2
	dataMHtrip$^{\oplus}$	273/2	337/2	40/1	40/1	—	723/2	462/2	765/2	71/2	77/3	—
	dataMHtrip$^{\ominus}$	—	—	40/1	40/1	—	—	—	—	—	—	—
	dataMHunit	41567/3	54584/3	—	7638/2	—	49478/3	45592/3	58299/4	—	13265/2	—
@0.2	dataSCcar	157332/3	368840/4	—	—	5/1	188365/4	193048/4	264094/4	—	—	10/2
	dataSUcar	157332/3	368840/4	31220/2	—	97404/3	188365/4	193048/4	264094/4	30072/2	—	30076/2
	dataSHcar$^{\oplus}$	12/1	17/1	3/1	—	5/1	17/1	15/1	20/1	7/2	—	10/2
	dataSHcar$^{\ominus}$	—	—	3/1	—	5/1	—	—	—	—	—	—
	dataSHunit	157332/3	368840/4	31220/2	—	—	188365/4	193048/4	264094/4	30072/2	—	—
	dataMCcar	—	—	3/1	—	5/1	—	—	—	7/2	—	10/2
	dataMCtrip	157332/3	368840/4	167/1	—	—	188365/4	193048/4	264094/4	605/3	—	—
	dataMUcar	—	—	167/1	167/1	522/2	—	—	—	605/3	316/3	452/3
	dataMUunit	157332/3	368840/4	—	31220/2	—	188365/4	193048/4	264094/4	—	55715/3	—
	dataMHcar	—	—	—	—	5/1	—	—	—	7/2	—	10/2
	dataMHtrip$^{\oplus}$	894/2	1240/2	167/1	167/1	—	2643/3	1693/2	2846/3	605/3	316/3	—
	dataMHtrip$^{\ominus}$	—	—	167/1	167/1	—	—	—	—	—	—	—
	dataMHunit	157332/3	368840/4	—	31220/2	—	188365/4	193048/4	264094/4	—	55715/3	—
@1.0	dataSCcar	1156876/4	1166357/4	—	—	11/1	776849/4	886984/4	1554721/4	—	—	20/2
	dataSUcar	1156876/4	1166357/4	143041/2	—	321046/3	776849/4	886984/4	1554721/4	134510/2	—	134526/2
	dataSHcar$^{\oplus}$	26/1	38/1	6/1	—	11/1	31/1	29/1	39/1	12/2	—	20/2
	dataSHcar$^{\ominus}$	—	—	6/1	—	11/1	—	—	—	—	—	—
	dataSHunit	1156876/4	1166357/4	143041/2	—	—	776849/4	886984/4	1554721/4	134510/2	—	—
	dataMCcar	—	—	6/1	—	11/1	—	—	—	12/2	—	20/2
	dataMCtrip	1156876/4	1166357/4	1153/2	—	—	776849/4	886984/4	1554721/4	2017/2	—	—
	dataMUcar	—	—	1153/2	1153/2	1738/2	—	—	—	2017/2	1458/3	2033/2
	dataMUunit	1156876/4	1166357/4	—	143041/2	—	776849/4	886984/4	1554721/4	—	254442/3	—
	dataMHcar	—	—	—	—	11/1	—	—	—	12/2	—	20/2
	dataMHtrip$^{\oplus}$	3879/2	8332/3	1153/2	1153/2	—	6306/3	7309/3	8019/3	2017/2	1458/3	—
	dataMHtrip$^{\ominus}$	—	—	1153/2	1153/2	—	—	—	—	—	—	—
	dataMHunit	1156876/4	1166357/4	—	143041/2	—	776849/4	886984/4	1554721/4	—	254442/3	—

TABLE VII
CALCULATED VS. OBSERVED INDEX SIZES FOR BERLINMOD

Only one common value is listed for calculated sizes of the spatial and temporal 2D indexes, as in our calculations, we use the same parameters for both. As mentioned within the text, pagesize is 4 kB.

For the OBA, we observe, that the CR has the smallest expected relation and index sizes, while the UR has always the largest expected relation and index sizes. The UR is expected to be +35.9% larger than the CR (+92.0% larger relation, +20.0% larger indexes). HR$^{\oplus}$ has an overhead of +8.3% (relations: +16.00%, indexes +6.2%); HR$^{\ominus}$ is only slightly better with +8.3% (relations: +15.99%/, indexes: +6.2%).

For the TBA, we also expect the CR to be the most compact format. Here, the HR$^{\oplus}$ is expected to be worst with respect to required space. However, it only has an overhead of +8.75% (relations: +15.82%, indexes: +6.7%). The HR$^{\ominus}$ is slightly better with +8.18% (relation: +15.09%, indexes: +6.2%). The UR is second place with expected +6.57% in total (relations: +7.6%, indexes +6.3%).

B. *Experimental Results*

The observed relation sizes are presented in Table V. We observe slight differences in the cardinalities (relative error $\leq 1\%$). For the tuple sizes, we observe significant differences between expected and observed values: SECONDO uses 1.58 to 3.91 times as much memory as the calculated minimum requirement. This directly affects the relation sizes (factors 2.38 to 3.29). The cause for this "wasted" space is the simple persistency mechanism applied by SECONDO, that does not use compression. As expected, the CR has the most compact, and the UR the largest relations. The OBA/HR requires about 1.12 times, and the OBA/UR 1.59 times the space of the OBA/CR. For the TBA, the differences are smaller: Here the TBA/HR requires about 1.13, the TBA/UR only 1.08 times the space required for the TBA/CR.

The observed statistics for the indexes are shown in Table VII. Several indexes seem to need less space than expected. There are two reasons for this good performance: (1) As we are interested in querying historic MOD, we use a bulkload mechanism to create the R-Trees. The algorithm is a variant of [38]. It creates packed trees by sequentially filling all leaf nodes (unless a key to insert is too far "off" the leaf node's current MBR) and creating upper inner nodes only when required. This might leave the rightmost path with a very low filling degree (nodes may even underflow), but this is acceptable, since the index buildup time is reduced drastically and updates are not required. Entries are fed into the bulkload after sorting them by means of a Z-ordering and therefore can be expected to show a good query performance. The 2D-R-trees used for spatial and temporal indexes all have order (33, 83), the 3D-R-trees (24, 62). For the 3D-Index @1.0, we achieve a net storage utilization of 54.19% (capacity = 99,502,144 keys: 1554721 pages, 64 keys each; occupied by the entries for 53,926,394 units). (2) SECONDO employs BerkeleyDB as a storage manager. B-Trees are implemented directly using BerkeleyDB's B-Trees, which seem to apply some optimization techniques to increase the storage performance.

The total relation, index, and database sizes for all representations are listed in Table VI.

VII. BERLINMOD/R PERFORMANCE

Due to different relation sizes and methods applied to access the data in the different representations, we expect them to behave differently executing the BerlinMOD/R benchmark queries. We compare our expectations with the practical benchmark results on SECONDO, using the same configuration as described in Section VI. Finally, we analyze selected queries in detail. The benchmark queries are listed in Appendex VIII.

A. Expected Runtimes

1) Compact Representation: In the CR, moving object data is primarily clustered by object identity. Time is always the secondary clustering criterion, as the units are sorted by their time intervals internally; using different clustering patterns is impossible. With respect to the cardinalities and relation sizes, the CR is minimal, hence we clearly expect it to be the best model for standard queries (Q1–2). For MOD queries, we expect advantages whenever data are accessed using the identity of the vehicles prior to further processing, as in queries Q3, and Q16. We also expect an advantage for the CR with any combination of dimensions with (Object Identity Known, Temporal; Q3, Q8), where the relatively small B-tree indexes can be used to find the moving objects and binary search be applied to the internal unit array. Worse for the CR in this category are queries for dimensions (Object Identity unknown) along with "Spatial" (Q4, Q7), or "Temporal" (Q9), as they can use the fine grained spatial/ temporal index, but nonetheless must scan the complete MO history. Since the indexes will trigger "false alarms" due to high rates of overlapping object MBRs — the temporal index will be worst, as the entry for each unit will overlap with one MBR for each vehicle at least — a linear search within each returned object's unit array is required to check for spatial conditions. While the coarse summary fields can be used to perform simple tests on the moving object (e.g. whether the moving object is possibly defined at a given instant, or the moving spatial object takes a value within a given range), complex tests are required to compute temporal or spatial predicates. On the other hand, aggregations over selected units within a single moving object should be quite fast.

2) Unit Representation: In the UR, we can cluster the data within the units relation by any criterion we want, e.g. by object identity, by the units' time intervals, or by their temporal values (resp. their MBR for spatial base types). A major disadvantage is the massive replication of non-moving attribute data, which drastically increases the size of the relation. As a side effect, maintaining keys (resp. attributes with a key property) is unfeasible within the UR. Hence, we expect the UR to perform worst on standard queries (Q1–2), as duplicates must always be dealed with. For MOD queries within dimension "spatial" (Q4–5, Q7), and "temporal" (Q3, Q8–9), we expect the UR to perform well. Here, the UR should benefit from the 1:1 relation of index entries to units within the relation. As all non-moving attributes of an object are contained in each unit tuple, we have immediate access to these data. However, for "aggregation" queries (Q9, Q17), grouping candidate tuples by moid is required, reducing the performance. As for differences between the two employed clustering schemas — ID/TMP (object identity, then temporal) and SPTMP (spatio-temporal) — we expect the SPTMP clustering to benefit queries applying spatio-temporal indexes (Q4, Q7, Q12 and Q17). The ID/TMP clustering will have its advantage, if temporal access or ordering/ grouping is used (Q3, Q5 and Q8).

3) Hybrid Representation: In both variants of the HR, clustering the unit relation by any criterion is possible. The representation needs more space than the CR (as referencing moids and tids have to be stored in the relations), but the requirements are far less than for the UR. The internal references lead to another main drawback, as the construction of indexes and relations becomes more complex compared to the CR or UR. Otherwise, the hybrid representation is expected to combine the advantages of both other forms of representation. In standard queries (Q1–2), the root relation can be used to access the units using the moid and the object index to retrieve the MOD if necessary. The behavior should be similar to the CR's here. As for MOD queries in the UR, the 1:1 mapping from index entries to tuples of the unit relation should make the HR fast for queries having dimensions "spatial" (Q4–5, Q7) and "temporal" (Q3, Q8–9). As spatio-temporal and further, non-moving attributes are kept separated in different relations, disk-accesses may be reduced in some cases. In other cases, data from both relations need to be joined (using moid with an object index, resp. the tid as the join attributes). When summary fields are maintained within the root relation, they can be used for simple prefiltering tests (e.g. to restrict complex tests to promising candidates). Again, we expect the SPTMP clustering schema to support queries using spatio-temporal indexes (Q4, Q7, Q12 and Q17), and the ID/TMP being superior with temporal access and grouping (Q8–9).

4) Semantics: OBA vs. TBA: As noted in Section V-C, the TBA implements a form of trajectory splitting and hence is expected to show some kind of "intermediate" behavior: While in the TBA trajectories are shorter than in the OBA, some queries require trip data to be joined with data from the root table, or duplicates being removed. Therefore, we expect the UR and HR to benefit less from using the TBA than the CR.

B. *Evaluation of BerlinMOD/R Running Times*

To execute the BerlinMOD/R queries on the SECONDO platform, we hand-translated the SQL queries into so-called SECONDO executable queries, which represent concrete query plans, i.e. we did not utilize the SQL query optimizer. For most queries, several equivalent translation variants were tried, but only the fastest response times were included into the results tables. The experiments are conducted using database scripts. Each script contains all executable queries for Q1 – Q17 as a series for each combination of: (1) semantic approach (OBA/TBA), (2) representation (CR/UR/HR), and (3) clustering Schema (ID/TMP and SPTMP)[5]. The experiments for both, HR$^{\oplus}$ and HR$^{\ominus}$ (HR with/without summary fields), have been combined in common scripts. The query scripts can be downloaded from the BerlinMOD web site [39].

We did not evaluate buffer effects explicitly, because (1) the scripts already employ a good mix of data access paths, and (2) it seems unreasonable to explicitly rule out buffering, because buffering effects are actually intended in operational DBMS. Since the workload is well-defined and the sequence of executed queries fixed, the results remain comparable. However, a strong buffer effect has been observed and explicitly examined for Q12 (see Section VII-B1).

The response times for all queries are listed in Table VIII for the OBA and Table IX for the TBA. As stated above, for the HR we have only created the HR$^{\oplus}$ representation, but formulated queries once not using summary fields (HR$^{\ominus}$ style) and once using them (HR$^{\oplus}$ style), where this seemed appropriate. Where the HR$^{\oplus}$ style did not seem appropriate at all, a "$\longleftarrow$" is printed in the HR$^{\oplus}$ columns of the result tables, meaning that the HR$^{\ominus}$ result is also used for the HR$^{\oplus}$. We continue with a brief analysis of the experimental results.

1) Semantics — OBA vs. TBA: In the OBA, the expectations for standard queries (Q1–2), have been confirmed. The CR and HR execute them fast, but the UR suffers from the replication of the standard attributes. For MOD queries, with increasing scale factor (i.e. increasing length of histories), UR and HR gain advantages regarding many queries, namely for Q4, Q7, and Q10–15. All these queries deal (a) with either unknown object identities or a possible relation between known objects, and (b) spatial or spatio-temporal dimension. In the temporal point query Q3, the CR takes advantage of a rapid selection by $Licence$ and a subsequent binary search within the array of units. The CR maintains an advantage in Q5, where the UR and HR require for additional aggregation steps. Another query preferring the CR is Q8, which requires to process the complete trajectory of candidate vehicles. Surprisingly, the CR also seems to have a major advantage with respect to Q9. The construction of the temporal selection and subsequent spatial projection for all objects' MOD seems to be much less expensive than selecting units using the temporal index and recreating the total trajectory applying grouping and aggregation. Also, the CR is clearly superior with regard to Q16. This again stems from the requirement to aggregate the units of candidates by $Licence$, here in order to allow for a parallel scan of the candidate pair's trajectories instead of checking all pairs of candidates' units.

In the TBA, the MOD histories within the CR are split into shorter trips. Therefore, the overhead for searching the history becomes drastically smaller, diminishing the CR's main disadvantage. As a consequence TBA/CR is faster than OBA/CR for some queries, e.g. Q13–15. TBA-queries Q11–14, and Q16 run fastest using the CR. For Q5, the CR's superiority becomes even clearer. This leads to the TBA/CR prepresentation being the global "winner" for 8 of the

[5]In the result tables, the columns for ID/TMP variants are marked with ^{id}, those for SPTMP are labeled with ^{3d}.

17 queries (namely Q3, Q5, Q8, Q11–14, and Q16) in our benchmark experiments. Also the TBA/UR using SPTMP clustering is quite good, scoring 3 best global results (Q4, Q7, Q12).

Regarding the majority of the benchmark queries, the TBA is superior to the OBA when using the CR or HR. However, there are some remarkable exceptions: Q5 runs very slown on the TBA/HR, and Q9 on the TBA/CR. As expected, the UR cannot generally profit from using the TBA: While the OBA is favored by 10 queries, the TBA is so by only 7. Among the latter are the standard queries (Q1–2), since in the TBA/UR the root relation can be used instead of the much larger unit relation. Q12 seems to be much faster in the TBA than in the OBA, but closer inspection proves this to be an artifact: (1) By mistake, we accidently translated the OBA variants in an improper way, using the spatial index and the **at** operator instead of the spatio-temporal index and the **atinstant** operator, penalizing the OBA variants. (2) When setting up the query scripts, we missed to notice that Q12 has exactly the same data access patterns as Q11 and the amount of data is relative small. Hence, the operating system's file caches allow for high cache-hit rates. Since with the TBA smaller amounts of data are loaded after the prefiltering step, cache hits are much higher here than with the OBA. This skews the performance results. To demonstrate these effects, we re-translated the OBA queries, now using the spatio-temporal index and the **atinstant** operator, and executed all queries with cleared system caches. The results[6] show, that the difference not only between OBA/CR@0.2 and TBA/CR@0.2, but even between all variants becomes almost insignificant: OBA/CR: 4.515 sec, TBA/CR: 4.685 sec; OBA/UR: 4.756 sec, TBA/UR: 4.435 sec; OBA/HR$^{\ominus}$: 4.671 sec, TBA/HR$^{\ominus}$: 4.411 sec. This is exactly what one should expect for this kind of simple spatio-temporal point queries.

2) Clustering Schemas — ID/TMP vs. SPTMP: Comparing the two clustering schemas for the unit relation in the UR and HR, the ID/TMP pattern is superior for Q1–3, Q5–6, Q8, Q10, and Q15, whereas the SPTMP pattern is better for Q4, Q7, Q9, Q12–14, Q16–17.[7] Only for Q11 the differences between both patterns are insignificant, probably, because the used spatio-temporal predicate is very specific. The experiments show, that the ID/TMP pattern is superior for standard queries (having an advantage during grouping and id-based duplicate removal operations), and if candidates are being selected using vehicle ids. Spatial, temporal or spatio-temporal data access benefits from using the SPTMP clustering pattern. These properties of the two clustering methods generally hold for both, UR and HR.

3) Summary Fields — HR$^{\ominus}$ vs. HR$^{\oplus}$: For the HR, considerable performance differences between queries using (HR$^{\oplus}$) and not using summary fields (HR$^{\ominus}$) have been established. In this analysis we focus on experiments with the full scale database (SF 1.0). For the OBA we observe, that only Q9 and Q16 (when using ID/TMP clustering), resp. Q3 (SPTMP clustering) profit from using summary fields. This is different regarding the TBA. Here, Q3, Q8–9, and Q16 are enhanced by using HR$^{\oplus}$ for both clustering schemas. There are two general explanations for the different behavior between OBA and TBA: (1) In the TBA, summaries exist for each trip, so that e.g. tests for overlapping MBRs can speed up processing significantly. (2) In the TBA one may restrict oneself to inspect fewer units if either a query object's identity is known or the search can be restricted to certain trips (i.e. certain trips could be selected using prefiltering on the trip summaries).

[6]Said results for ID/TMP clustering are also displayed in Table VIII, and Table IX.
[7]In the result tables, ID/TMD results are marked ^{id}, SPTMP results are labeled ^{3d}.

C. Exemplary Query Analyses

In this subsection, we analyze three of the queries to show, how the query plans relate to observed query response times. We use the following abbreviations: `QL` for `QueryLicences`, `QP` for `QueryPeriods`, and `QI` for `QueryInstants`. We also shorten all relation names starting with “data”, so `dataXY` becomes `XY`.

1) Q3: We used the following plans for this temporal point query on known objects: *OBA/CR:* For `QL1`, select according tuples from `SCcar` using a B-tree index. Then for each instant from `QI1` calculate the vehicles’ position. *OBA/UR:* For `QL1`, select the JourneyUnits from `SUcar` using a B-tree index. Join each unit with all instants from `QI1` selecting those pairs, where the unit is present at the given instant. *OBA/HR$^{\ominus}$:* For `QL1`, select trips from `SHcar`, then units from `SHunit` using the B-tree indexes, joins units with `QI1`, selecting TripUnits present at an query instant and evaluate the temporal function. *OBA/HR$^{\oplus}$:* For `QL1`, select trips from `SHcar` and join with `QL1`, selecting pairs whose query instant is contained by the TripDeftime summary (but as the summary is for the complete journey, this is always the case). Then fetch all units from `SHunit` to refine the temporal containment and evaluate the temporal function. *TBA/CR:* For `QL1` retrieve all Moids from `MCcar` using a B-tree index, then retrieve all TripUnits belonging to each found Moid from `MCtrip` using a B-tree index. Join with all instants from `QL1`, where the TripUnit is present at the given instant. *TBA/UR:* For `QL1` retrieve all TripIds from `MUcar` and then all TripUnits belonging to the according TripIds (both times using B-tree indexes). Join with all instants from `QL1`, where the TripUnit is present at the given instant. *TBA/HR$^{\ominus}$:* For `QL1` select the Moids from `MHcar`, then retrieve all according TripIds from `MHtrip` and finally the TripUnits from `MHunit` using the three according B-trees. Then join with all instants from `QL1`, where the TripUnit is present at the given instant. *TBA/HR$^{\oplus}$:* For `QL1` select all Moids from `MHcar` and get all according trips descriptions from `MHtrip` using two B-trees. Then join with the instants from `QL1` selecting TripIds whose TripDeftime contains the given query instant. Retrieve the TripUnits belonging to the remaining TripIds using a B-tree and restrict to those tuples, where the TripUnit is present at the query instant (this is necessary as a trip within `MHtrip` theoretically may contain definition gaps).

Q3 shows some interesting results. First, it is simple to understand, that the CR is superior to the UR and HR, as for each given licence, one just extracts the car’s position at each of the query instants. Second, as basically the same access sequence (first id, then time) is also used for the UR and HR, also the superiority of the ID/TMP to the SPTMP is clear. Third, using summary fields is good for all HR-variants except the OBA/HR, where the prefiltering adds complexity, but does not drop any candidates (because prefiltering is always positive).

2) Q6: The plans used for this unbounded spatio-temporal join query with unknown identities and without aggregation are: *OBA/CR:* Select trucks from `SCcar`, selfjoin if minimum distance ever becomes $<$10m. *OBA/UR:* Create a materialized view with all trucks’ JourneyUnits from `SUcar`. Create a spatio-temporal index (3D R-Tree) for the view, with keys buffered by 5m in both spatial dimensions. Do a spatial selfjoin on this R-Tree, selecting pairs of units from different trucks and that getting nearer than 10m. Remove duplicates. *OBA/HR$^{\ominus}$:* Select trucks from `SHcar`, retrieve according units from `SUunit` using the moid B-tree. Join with all units that intersect with their distance-buffered MBR using the spatio-temporal index. Drop pairs of units belonging to the same truck, evaluate the spatio-temporal condition and drop pairs with non-truck join partners. *OBA/HR$^{\oplus}$:* Select trucks from `SHcar`, use the spatio-temporal index for a distance-buffered self-join on the MBR summaries, filtering pairs of different trucks. Retrieve units for the first and second truck truck from `SHunit`, using the moid index. Do a distance-buffered spatial join between them and evaluate the spatio-temporal condition. Remove dublicates. *TBA/CR:* Select trucks from `MCcar`, retrieve all their trips from `MCtrip` and do a buffered spatial selfjoin select pairs that have different licences and whose trips come closer than 10m and remove duplicates. *TBA/UR:* Select all trucks from `MUcar` and retrieve the according TripUnits from `MUunit` using a B-tree, store them as a materialized view. Create a spatio-temporal index (3D R-Tree) with keys buffered by 5m in both spatial dimensions. Do a spatial selfjoin on this R-Tree, filter pairs of units, that have different moids and belong to different trucks and ever get nearer than 10m. Finally, remove duplicates. *TBA/HR$^{\ominus}$:* An R-tree-indexed relation with distance-buffered MBRs of all trucks' units is created which is used to evaluate a spatial selfjoin as a prefilter. Then the spatio-temporal condition is applied and duplicate results are removed. *TBA/HR$^{\oplus}$:* Create a relation with the extended TripMBRs for all trucks' trips. Apply a selfjoin to find all trip pairs with different Moids having overlapping distance-buffered TripMBRs. Then, join in the units for each pair of trips using the B-tree index. Filter the tuples having overlapping buffered MBRs and the evaluate the spatio-temporal predicate. Remove all duplicate results.

Q6 strongly benefits the TBA/CR (small product size for selfjoin, short mpoint values for parallel scan), and the OBA/UR (high selectivity for the spatial selfjoin). TBA is superior to OBA (single trips can be retrieved instead of the complete history of an object, reducing the cardinality of intermediate results by orders of magnitude). Maybe surprisingly, the clustering schema clearly is the dominating factor in the TBA, ID/TMP being superior to SPTMP (because unit access is always ordered by Moid (TripId)).

3) Q9: This is a temporal range query for unknown objects using an aggregation. *OBA/CR:* Create the product of `SCcar` and `QP`. Extend each tuple with the travelled distance of Journey restricted to the query period. Group by query period to extract the maximum travelled distance. *OBA/UR:* Retrieve all tuples from `SUcar` whose JourneyUnit is present at a query period from `QP` using the temporal R-tree index. Extend each tuple with the length of JourneyUnit restricted to the query period. Group by Moid sum up the total travelled distance for each vehicle. Finally, group by query period to extract the maximum total distance. *OBA/HR$^{\ominus}$:* Use the temporal index to retrieve all units from `SHunit` present at a given Period from `QP`. Restrict the units to the query period and group by Moid and query period to calculate the travelled distance for each vehicle and period. After that, grouping is used to select the maximum travelled distance for each period. *OBA/HR$^{\oplus}$:* Join `QP` with all cars from `SHcar` selecting pairs where the JourneyDeftime summary field intersects the query period. Then load the journey units using the Moid B-tree from `SHunit`. Apply the refined temporal condition, restrict the units to the query periods and sum up the total length for each combination of query period and Moid. Group by query period to extract the maximum value. *TBA/CR:* For each query period from `QP` load all trips from `MCtrip`, whose deftime intersect the query period (using the temporal index). Calculate the length of the Trip restricted to the query period. Group by Moid to sum up the total travelled distance for each vehicle. The group by query periods to extract the maximum travelled distance. *TBA/UR:* For each query period from `QP` load all units from `MUunit`, whose deftime intersect the query period (using the temporal index). Calculate the length of the TripUnit restricted to the query period. Group by Moid to sum up the total length for each vehicle. Group by query period to extract the maximum travelled distance. *TBA/HR$^{\ominus}$:* For each query period from `QP` load all trips from `MCtrip`, whose deftime intersect the query period (using the temporal index). Calculate the length of the Trip restricted to the query period. Group by Moid to sum up the total travelled distance for each vehicle. The group by query periods to extract the maximum travelled distance. *TBA/UR:* For each query period from `QP` load all units from `MHunit`, whose deftime intersect the query period (using the temporal index). Calculate the length of the TripUnit restricted to the query period. Group by TID to sum up the total length for each trip. Retrieve the trip tuple from `MHtrip` "containing" the grouped units dereferencing the TID link. Group by Moid to calculate the total length for each vehicle. Group by query period to extract the maximum travelled distance. *TBA/HR$^{\oplus}$:* For each query period from `QP` load all trips from `MHtrip`, whose deftime intersect the query period (using the temporal index). Using the Tripid B-tree index, load all according units from `MHunit` using the TripId B-tree. Calculate the length of the TripUnit restricted to the query period. Group by Moid to sum up the total length for each vehicle. Group by query period to extract the maximum travelled distance.

The query clearly favours the OBA/CR, because during query processing, cardinalities remain small and the temporal restrictions are fast and easy to perform. Only a single grouping is required to get the result. All other variants are more than 6 times slower. The second rank is for the TBA/HR$^{\oplus}$ with IDTMP clustering, where candidate trips can be selected fast using the TripDeftime summary. Then, the unit data is accessed ordered by Tripid and time, perfectly matching the clustering schema. Using SPTMP more than doubles the responce time, This also holds for the remaining TBA variants. While still existent, the benefits of using the Deftime summary becomes much smaller for OBA, since the definition time summary field is more accurate for single trips. For the OBA/UR and OBA/HR$^{\ominus}$ variants, the SPTMP clustering is better than the ID/TMP due to the access pattern created by the result sequence returned by the temporal index, which does not reflect the object identity.

VIII. Conclusions and Research Perspectives

We have proposed three different representations for storing time-sliced moving object data. Regarding queries and general settings for a database and application domain, we analyzed the factors having impact on their query and space performance, and using the BerlinMOD/R benchmark, we formulated expected differences between the representations. We implemented data types and algorithms required to create and evaluate the BerlinMOD data for each of the representations in the Secondo extensible DBMS and evaluated all models in this concrete database system.

As expected, the UR and HR showed to have some advantages with simple point and range queries, where no further processing is required after the selection (Q11–15). For queries considering the complete history of a moving object (Q5–6, Q8–9), the CR is the better choice.

In the comparison of the UR and HR, the summary fields within the HR$^{\oplus}$ have proved to be only of marginal use. They are only useful within queries employing very rough selections like "Was object X present at instant I" (Q3), "Which objects were present between instants t_1 to t_2", or "Which objects ever moved within area R", where they can only be used as an initial filtering step. This seems to be more useful in the according TBA queries than in the corresponding OBA queries. Otherwise, it is more convenient to directly use the more fine grained spatial, temporal or spatio-temporal indexes to retrieve candidates within point and range queries.

	Query	OBA/CR	OBA/UR^{id}	OBA/UR^{3d}	OBA/HR$^{\ominus,id}$	OBA/HR$^{\oplus,id}$	OBA/HR$^{\ominus,3d}$	OBA/HR$^{\oplus,3d}$
@0.05	Q1	0.262	0.001	1.906	0.001	←	0.122	←
	Q2	0.027	0.001	38.752	0.004	←	0.004	←
	Q3	1.285	0.899	7.913	0.901	4.915	22.256	5.540
	Q4	34.587	0.239	7.437	0.280	←	16.119	←
	Q5	5.090	13.488	53.652	9.787	←	6.851	←
	Q6	31.506	38.149	58.402	132.517	4244.040	118.437	4403.280
	Q7	38.236	1.818	2.572	0.992	←	0.990	←
	Q8	0.453	2.263	24.823	2.244	6.803	2.318	7.446
	Q9	189.092	1014.490	1077.650	1524.180	3867.880	1546.830	4052.970
	Q10	213.984	35.522	29.817	37.401	←	41.392	←
	Q11	1.347	0.888	0.580	0.455	←	0.843	←
	Q12[†]	0.710	0.079	0.069	0.051	←	0.051	←
	Q13	29.304	42.408	38.900	37.392	←	40.272	←
	Q14	0.645	0.900	0.630	0.848	←	0.900	←
	Q15	2.446	1.679	1.583	1.816	←	1.534	←
	Q16	42.110	250.423	140.360	244.644	98.393	221.559	181.749
	Q17	2.675	18.330	7.824	30.684	←	21.727	←
@0.2	Q1	0.190	4.885	4.074	0.103	←	0.106	←
	Q2	0.004	141.508	194.426	0.005	←	0.005	←
	Q3	2.377	3.197	25.212	3.452	13.127	77.717	13.793
	Q4	163.173	61.755	26.465	105.537	←	53.633	←
	Q5	7.250	24.320	100.302	21.172	←	20.678	←
	Q6	220.035	196.260	220.671	701.332	57082.800	679.447	57645.500
	Q7	201.235	109.870	50.657	145.098	←	69.025	←
	Q8	0.711	5.796	74.813	4.497	18.129	5.343	78.907
	Q9	450.832	18955.500	12064.000	15456.300	16132.400	11704.100	19133.800
	Q10	1264.260	160.557	216.539	174.406	←	232.595	←
	Q11	4.010	1.412	1.253	1.390	←	1.290	←
	Q12[†]	35.965	13.289	5.800	19.775	←	7.682	←
	Q12[‡]	4.515	4.756	—	4.671	←	—	—
	Q13	60.725	126.367	94.153	146.240	←	97.531	←
	Q14	2.045	4.072	5.884	3.605	←	3.401	←
	Q15	24.988	18.313	14.506	19.996	←	10.586	←
	Q16	65.539	513.499	328.596	520.173	193.745	411.003	440.091
	Q17	46.104	91.857	13.317	133.918	←	92.517	92.517
@1.0	Q1	0.187	6.840	7.469	0.110	←	0.307	←
	Q2	0.014	793.578	990.027	0.009	←	0.009	←
	Q3	4.132	6.158	221.999	6.657	26.925	228.287	28.847
	Q4	1540.350	401.612	190.306	421.459	←	177.656	←
	Q5	14.950	63.320	574.382	56.889	←	485.887	←
	Q6	1892.430	819.940	825.850	6102.880	1183340.000	6012.200	1249690.000
	Q7	4293.671	442.659	215.382	899.849	←	333.711	←
	Q8	5.314	9.603	226.742	8.246	35.620	14.190	230.862
	Q9	2174.680	85226.900	55128.100	118095.000	76242.100	51634.500	183316.000
	Q10	8871.060	714.355	882.736	881.063	←	848.341	←
	Q11	10.538	3.183	3.253	3.122	←	3.163	←
	Q12[†]	906.701	62.633	28.775	125.353	←	47.576	←
	Q13	356.500	144.867	89.181	200.313	←	76.835	←
	Q14	13.604	7.738	11.987	8.332	←	6.622	←
	Q15	253.536	37.465	17.150	58.318	←	16.524	←
	Q16	59.455	719.583	606.589	733.218	385.471	206.785	869.483
	Q17	1539.700	413.274	203.257	882.076	←	313.049	←

TABLE VIII
OBSERVED RESULTS FOR BERLINMOD OBA-QUERIES USING SECONDO: TOTAL RESPONSE TIMES [SEC]

[†]: Q12 was run using a wrong query translation. [‡]: Therefore, Q12 was reevaluated using corrected translations @0.2 and ruling out cache effects.

	Query	TBA/CR	TBA/UR^{id}	TBA/UR^{3d}	TBA/HR$^{\ominus,id}$	TBA/HR$^{\oplus,id}$	TBA/HR$^{\ominus,3d}$	TBA/HR$^{\oplus,3d}$
@0.05	Q1	0.046	0.186	0.150	0.081	←	0.089	←
	Q2	0.003	0.047	0.034	0.004	←	0.004	←
	Q3	0.327	1.725	65.319	5.595	1.197	76.122	4.227
	Q4	57.370	28.338	4.307	30.996	←	4.940	←
	Q5	4.314	5117.194	5306.014	1680.635	←	1855.058	1855.058
	Q6	4.028	28.616	144.718	33.909	66.540	200.463	68.227
	Q7	24.332	3.436	2.693	4.744	←	3.180	←
	Q8	0.349	6.397	8.592	6.804	789.272	9.092	1.422
	Q9	411.694	4849.780	5138.690	1040.500	789.272	1196.930	1051.900
	Q10	51.171	31.166	32.076	24.156	←	24.810	←
	Q11	0.508	1.207	1.105	0.739	←	0.851	←
	Q12†	0.124	0.090	0.098	0.111	←	0.116	←
	Q13	13.626	53.317	45.301	45.931	←	40.192	←
	Q14	0.246	0.847	0.704	1.246	←	0.122	←
	Q15	1.550	2.108	1.587	1.658	←	1.744	←
	Q16	4.697	160.765	243.615	23.806	7.271	20.143	8.918
	Q17	1.444	21.603	5.632	2.388	←	5.349	←
@0.2	Q1	0.079	0.394	0.369	0.082	←	0.077	←
	Q2	0.003	0.215	0.168	0.005	←	0.005	←
	Q3	0.486	3.485	347.377	14.619	2.273	372.418	17.100
	Q4	267.167	109.938	10.570	118.496	←	11.718	←
	Q5	6.561	27580.610	28138.400	5188.582	←	6083.815	←
	Q6	6.464	99.577	951.223	118.378	280.166	1169.331	984.615
	Q7	104.564	104.432	14.861	120.182	←	17.099	←
	Q8	0.462	16.564	353.958	17.988	5.392	108.600	120.058
	Q9	2921.800	8976.480	15407.900	18876.000	2942.880	50984.200	43824.200
	Q10	482.898	86.946	161.593	74.452	←	417.820	←
	Q11	0.389	1.421	1.130	1.069	←	1.303	←
	Q12†	0.153	0.116	0.136	0.135	←	0.160	←
	Q12‡	4.685	4.435	—	4.411	←	—	—
	Q13	54.636	150.869	88.704	131.321	←	92.782	←
	Q14	0.925	4.519	6.079	4.460	←	3.737	←
	Q15	27.672	21.540	5.825	23.915	←	8.143	←
	Q16	6.547	441.481	572.390	46.820	14.709	51.617	20.222
	Q17	128.523	102.351	4.044	110.043	←	11.807	←
@1.0	Q1	0.097	1.181	1.667	0.200	←	0.140	←
	Q2	0.005	1.160	0.724	0.008	←	0.008	←
	Q3	0.954	6.375	1313.900	28.109	3.177	1281.970	26.258
	Q4	2230.440	459.191	25.071	435.355	←	31.176	←
	Q5	10.091	86319.730	88602.890	7303.940	7303.940	9675.540	←
	Q6	23.815	374.373	9700.021	454.592	2140.877	8849.633	53314.719
	Q7	2236.707	475.536	45.178	464.588	←	59.698	←
	Q8	2.198	38.937	1352.870	34.920	5.416	931.084	267.802
	Q9	15012.300	26656.700	70687.700	82092.600	13998.700	427516.000	291820.000
	Q10	3983.280	741.830	2220.020	708.417	←	2120.840	←
	Q11	3.550	2.977	3.580	3.127	←	3.093	←
	Q12†	0.172	0.167	0.161	0.166	←	0.196	←
	Q13	166.185	169.501	123.163	140.526	←	92.806	←
	Q14	4.425	7.619	6.609	7.585	←	5.605	←
	Q15	108.037	46.862	18.632	49.670	←	16.011	←
	Q16	8.095	1067.479	1089.416	83.854	17.746	89.493	24.300
	Q17	2035.960	467.890	39.538	435.972	←	33.767	←

TABLE IX

OBSERVED RESULTS FOR BERLINMOD TBA-QUERIES USING SECONDO: TOTAL RESPONSE TIMES [SEC]

†: Q12 was run using a wrong query translation. ‡: Therefore, Q12 was reevaluated using corrected translations @0.2 and ruling out cache effects.

We also evaluated the impact of using different clustering schemas for the unit stores required by the UR and HR. Both examined variants showed to have their domain: If standard predicates dominate the queries, the ID/TMP should be preferred, for spatio-temporal queries, the SPTMP clustering leads to a significantly increased query performance.

One aspect of future work may aim at identifying the impact of additional clustering schemas for the unit relations. Other aspects are about how to integrate the presented results into query optimization. As the queries for the UR and especially the HR representations showed to be quite complex, translation rules need to be defined. A challenging optimization perspective is automatic schema migrations in order to migrate the MOD representation schema according to the observed working sets of queries.

Acknowledgments

This work was partially supported by grant Gu 293/8-2 from the Deutsche Forschungsgemeinschaft (DFG), project “Datenbanken für bewegte Objekte”.

References

[1] N. Pelekis, B. Theodoulidis, I. Kopanakis, and Y. Theodoridis, “Literature review of spatiotemporal database models,” *The Knowledge Engineering Review*, vol. 19, no. 3, pp. 235–274, 2005.

[2] G. Langran, “A framework for temporal geographic information,” *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 25, no. 3, 1988.

[3] Gail Langran, *Time In Geographic Information Systems*. London: Taylor and Francis, 1992.

[4] G. J. Hunter and I. P. Williamson, “The development of a historical digital cadastral database,” *International Journal of Geographic Information Systems*, vol. 4, no. 2, 1990.

[5] D. Peuquet and E. Wentz, “An approach for time-based spatial analysis of spatio-temporal data,” in *Advances in GIS Research, Proceedings 1*, 1994, pp. 489–504.

[6] D. Peuquet and N. Duan, “An event-based spatio-temporal data model (ESTDM) for temporal analysis of geographical data,” *International Journal of Geographical Information Science*, vol. 9, no. 1, pp. 7–24, 1995.

[7] C. Claramunt and M. Thériault, “Managing time in GIS: an event-oriented approach,” in *Temporal Databases*, ser. Workshops in Computing, J. Clifford and A. Tuzhilin, Eds. Springer, 1995, pp. 23–42.

[8] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, “Querying the uncertain position of moving objects,” *Lecture Notes in Computer Science*, vol. 1399, pp. 310–337, 1998.

[9] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez, “Cost and imprecision in modeling the position of moving objects,” in *ICDE*, 1998, pp. 588–596.

[10] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, “Moving objects databases: Issues and solutions,” in *Statistical and Scientific Database Management*, 1998, pp. 111–122.

[11] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha, “Updating and querying databases that track mobile units,” *Distributed and Parallel Databases*, vol. 7, no. 3, pp. 257–387, 1999.

[12] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis, “Spatio-temporal data types: An approach to modeling and querying moving objects in databases,” *GeoInformatica*, vol. 3, no. 3, pp. 269–296, 1999.

[13] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, “A data model and data structures for moving objects databases,” in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM Press, 2000, pp. 319–330.

[14] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis, “A foundation for representing and quering moving objects.” *ACM Trans. Database Syst.*, vol. 25, no. 1, pp. 1–42, 2000.

[15] J. A. C. Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, “Algorithms for moving objects databases.” *Comput. J.*, vol. 46, no. 6, pp. 680–712, 2003.

[16] S. Dieker and R. H. Güting, “Plug and play with query algebras: SECONDO - a generic DBMS development environment,” in *Proc. of the International Symposium on Database Engineering & Applications*, 2000, pp. 380–392.

[17] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle, “Secondo: An extensible DBMS platform for research prototyping and teaching.” in *ICDE*, 2005, pp. 1115–1116.

[18] R. H. Güting, T. Behr, and C. Düntgen, “Secondo: A platform for moving objects database research and for publishing and integrating research implementations,” *IEEE Data Eng. Bull.*, vol. 33, no. 2, pp. 56–63, 2010.

[19] N. Pelekis, “STAU: A Spatio-Temporal Extension to Oracle DBMS,” UMIST, Department of Comptation, Tech. Rep. Ph.D. Thesis, 2002.
[20] N. Pelekis and Y. Theodoridis, “Boosting location-based services with a moving object database engine,” in *MobiDE ’06: Proceedings of the 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access*. New York, NY, USA: ACM, 2006, pp. 3–10.
[21] N. Pelekis, Y. Theodoridis, S. Vosinakis, and T. Panayiotopoulos, “Hermes - a framework for location-based data management,” in *In Proceedings of 10th International Conference on Extending Database Technology (EDBT’06)*, 2006, pp. 1130–1134.
[22] N. Pelekis and Y. Theodoridis, “An Oracle data cartridge for moving objects,” Information Systems Lab., University of Piraeus, Piraeus, Hellas, Tech. Rep. UNIPI-ISL-TR-2007-04, December 2007.
[23] S. Boulahya, “Représentation et interrogation de données spatio-temporelles: Cas d’étude sur PostgreSQL/PostGIS,” Master’s thesis, Université Libre de Bruxelles, Faculté des Sciences, D’epartement d’Informatique, 2009.
[24] Y. E. Qajary, “Design and implementation of a moving object database for truck information systems,” in *Map Middle East 2008 (MME 2008)*, 2008.
[25] M. Vazirgiannis and O. Wolfson, “A spatiotemporal model and language for moving objects on road networks,” in *SSTD ’01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*. London, UK: Springer-Verlag, 2001, pp. 20–35.
[26] L. Becker, H. Blunck, K. Hinrichs, and J. Vahrenhold, “A framework for representing moving objects.” in *DEXA*, 2004, pp. 854–863.
[27] B. Yu, S. H. Kim, T. Bailey, and R. Gamboa, “Curve-based representation of moving object trajectories,” in *IEEE International Database Engineering and Applications Symposium*, 2004, pp. 419–425.
[28] J. Eisenstein, S. Ghandeharizadeh, C. Shahabi, G. Shanbhag, and R. Zimmermann, “Alternative representations and abstractions for moving sensors databases,” in *Proceedings of the Tenth International Conference in Information and Knowledge Management (CIKM)*. ACM, 2001, pp. 5–10.
[29] C. Düntgen, T. Behr, and R. H. Güting, “BerlinMOD: a benchmark for moving object databases,” *The VLDB Journal*, vol. 18, no. 6, pp. 1335–1368, 2009.
[30] T. Behr, V. T. de Almeida, and R. H. Güting, “Representation of periodic moving objects in databases,” in *GIS’06: Proceedings of the 14th annual ACM International Symposium on Advances in Geographic Information Systems*. New York, NY, USA: ACM, 2006, pp. 43–50.
[31] A. Frihida, D. Zheni, H. H. B. Ghézala, and C. Claramunt, “Modeling trajectories: A spatio-temporal data type approach,” in *DEXA Workshops*, A. M. Tjoa and R. Wagner, Eds. IEEE Computer Society, 2009, pp. 447–451.
[32] R. H. Güting, V. T. de Almeida, and Z. Ding, “Modeling and querying moving objects in networks,” *The VLDB Journal*, vol. 15, no. 2, pp. 165–190, 2006.
[33] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento, “A trajectory splitting model for efficient spatio-temporal indexing.” in *VLDB*, 2005, pp. 934–945.
[34] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *International Conferenece on Management of Data*. ACM, 1984, pp. 47–57.
[35] D. Pfoser, C. S. Jensen, and Y. Theodoridis, “Novel approaches in query processing for moving object trajectories.” in *VLDB*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan Kaufmann, 2000, pp. 395–406.
[36] V. T. de Almeida, R. H. Güting, and C. Düntgen, “Multiple entry indexing and double indexing,” in *IDEAS 2007*. IEEE Computer Society, 2007, pp. 181–189.
[37] 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 Conference*, 1990, pp. 322–331.
[38] I. Kamel and C. Faloutsos, “On packing R-trees,” in *CIKM ’93: Proceedings of the Second International Conference on Information and Knowledge Management*. New York, NY, USA: ACM, 1993, pp. 490–499.
[39] “BerlinMOD Benchmark Web Site,” http://www.informatik.fernuni-hagen.de/import/pi4/secondo/BerlinMOD/BerlinMOD.html, 2010.

APPENDIX

We provide the BerlinMOD/R OBA queries as a quick reference. Full query texts are available from [29]. Several relations and attributes have been renamed for the sake of disambiguation. We only list the queries in common English and the SQL-queries for the OBA/CR. The SQL-queries for other representations can be formulated in an analog way.

Q1: What are the models of the vehicles with licence plate numbers from `QueryLicence`?

```
SELECT DISTINCT LL.Licence AS Licence, Model FROM dataSCcar, QueryLicence LL
WHERE Licence = LL.Licence;
```

Q2: How many vehicles exist that are “passenger” cars?

```
SELECT COUNT(Licence) FROM dataSCcar WHERE Type = ”passenger”;
```

Q1: What are the models of the vehicles with licence plate numbers from `QueryLicence`?

```
SELECT DISTINCT LL.Licence AS Licence, Model FROM dataSCcar, QueryLicence LL
WHERE Licence = LL.Licence;
```

Q2: How many vehicles exist that are “passenger” cars?

```
SELECT COUNT(Licence) FROM dataSCcar WHERE Type = ”passenger”;
```

Q4: Which licence plate numbers belong to vehicles that have passed the points from `QueryPoints`?

```
SELECT PP.Pos AS Pos, C.Licence AS Licence FROM dataSCcar C, QueryPoints PP
WHERE C.Journey passes PP.Pos;
```

Q5: What is the minimum distance between places, where a vehicle with a licence from `QueryLicences1` and a vehicle with a licence from `QueryLicences2` have been?

```
SELECT LL1.Licence AS Licence1, LL2.Licence AS Licence2,
       distance(trajectory(V1.Journey),trajectory(V2.Journey)) AS Dist
FROM dataSCcar V1, dataSCcar V2, QueryLicences1 LL1, QueryLicences2 LL2
WHERE V1.Licence = LL1.Licence AND V2.Licence = LL2.Licence AND V1.Licence <> V2.Licence;
```

Q6: What are the pairs of licence plate numbers of “trucks”, that have ever been as close as 10m or less to each other?

```
SELECT V1.Licence AS Licence1, V2.Licence AS Licence2 FROM dataSCcar V1, dataSCcar V2
WHERE V1.Licence < V2.Licence AND V1.Type = ”truck” AND V2.Type = ”truck”
      AND sometimes(distance(V1.Journey,V2.Journey) <= 10.0);
```

Q7: What are the licence plate numbers of the “passenger” cars that have reached the points from `QueryPoints` first of all “passenger“ cars during the complete observation period?

```
SELECT PP.Pos AS Pos, V1.Licence AS Licence FROM dataSCcar V1, QueryPoints PP
WHERE V1.Journey passes PP.Pos AND V1.Type = ”passenger”
      AND inst(initial(V1.Journey at PP.Pos)) <= ALL
      ( SELECT inst(initial(V2.Journey at PP2.Pos)) AS FirstTime
        FROM dataSCcar V2 WHERE V2.Journey passes PP.Pos AND V2.Type = ”passenger” );
```

Q8: What are the overall travelled distances of the vehicles with licence plate numbers from `QueryLicences1` during the periods from `QueryPeriods1`?

```
SELECT V1.Licence AS Licence, PP.Period AS Period,
       length(V1.Journey atperiods PP.Period) AS Dist
FROM dataSCcar V1, QueryPeriods1 PP, QueryLicences1 LL
WHERE V1.Licence = LL.Licence AND V1.Journey present PP.Period;
```

Q9: What is the longest distance that was travelled by a vehicle during each of the periods from `QueryPeriods`?

```
SELECT PP.Period AS Period, MAX(length(V1.Journey atperiods PP)) AS Dist
FROM dataSCcar V1, QueryPeriods PP WHERE V1.Journey present PP.Period GROUP BY PP.Period;
```

Q10: When and where did the vehicles with licence plate numbers from `QueryLicences1` meet other vehicles (distance $<$ 3m) and what are the latters’ licences?

```
SELECT V1.Licence AS QueryLicence, V2.Licence AS OtherLicence,
    (V1.Journey atperiods(deftime((distance(V1.Journey, V2.Journey) <= 3.0)
    at TRUE))) AS Pos
FROM dataSCcar V1, dataSCcar V2, QueryLicences1 LL
WHERE V1.Licence = LL.Licence AND V2.Licence <> V1.Licence
    AND sometimes(distance(V1.Journey, V2.Journey) <= 3.0);
```

Q11: Which vehicles passed a point from `QueryPoints1` at one of the instants from `QueryInstants1`?

```
SELECT C.Licence AS Licence, PP.Pos AS Pos, II.Instant AS Instant
FROM dataSCcar C, QueryPoints1 PP,  QueryInstants1 II
WHERE val(C.Journey atinstant II.Instant) = PP.Pos;
```

Q12: Which vehicles met at a point from `QueryPoints1` at an instant from `QueryInstants1`?

```
SELECT PP.Pos AS Pos, II.Instant AS Instant, C1.Licence AS Licence1, C2.Licence AS Licence2
FROM dataSCcar C1, dataSCcar C2, QueryPoints1 PP, QueryInstants1 II
WHERE val(C1.Journey atinstant II.Instant) = PP.Pos
      AND val(C2.Journey atinstant II.Instant) = PP.Pos;
```

Q13: Which vehicles travelled within one of the regions from `QueryRegions1` during the periods from `QueryPeriods1`?

```
SELECT RR.Region AS Region, PP.Period AS Period, C.Licence AS Licence
FROM dataSCcar C, QueryRegions1 RR, QueryPeriods1 PP
WHERE NOT(isempty(((C.Journey atperiods PP.Period) at RR.Region)));
```

Q14: Which vehicles travelled within one of the regions from `QueryRegions1` at one of the instants from `QueryInstants1`?

```
SELECT RR.Region AS Region, II.Instant AS Instant, C.Licence AS Licence
FROM dataSCcar C, QueryRegions1 RR, QueryInstants1 II
WHERE val(C.Journey atinstant II.Instant) inside RR.Region;
```

Q15: Which vehicles passed a point from `QueryPoints1` during a period from `QueryPeriods1`?

```
SELECT PO.Pos AS Pos, PR.Period AS Period, C.Licence AS Licence
FROM dataSCcar C, QueryPoints1 PO, QueryPeriods1 PR
WHERE NOT(isempty(((C.Journey atperiods PR.Period) at PO.Pos)));
```

Q16: List the pairs of licences for vehicles, the first from `QueryLicences1`, the second from `QueryLicences2`, where the corresponding vehicles are both present within a region from `QueryRegions1` during a period from `QueryPeriod1`, but do not meet each other there and then.

```
SELECT PP.Period AS Period, RR.Region AS Region, C1.Licence AS Licence1,
       C2.Licence AS Licence2
FROM dataSCcar C1, dataSCcar C2, QueryRegions1 RR, QueryPeriods1 PP,
     QueryLicences1 LL1, QueryLicences2 LL2
WHERE C1.Licence = LL1.Licence AND C2.Licence = LL2.Licence
      AND LL1.Licence <> LL2.Licence AND (C1.Journey at PP.Period) passes RR.Region
      AND (C2.Journey at PP.Period) passes RR.Region
      AND isempty((intersection(C1.Journey, C2.Journey) atperiods PP.Period)
          at RR.Region);
```

Q17: Which points from `QueryPoints` have been visited by a maximum number of different vehicles?

```
CREATE VIEW PosCount AS SELECT PP.Pos AS Pos, COUNT(C.Licence) AS Hits
   FROM QueryPoints PP, dataSCcar C WHERE C.Journey passes PP.Pos  GROUP BY PP.Pos;
SELECT Pos FROM PosCount AS N WHERE  N.Hits = (SELECT MAX(Hits) FROM PosCount);
```