Multidimensional Interfaces for Selecting Data within Ordinal Ranges

Ruben Taelman, Pieter Colpaert, Ruben Verborgh, and Erik Mannens

Data Science Lab (Ghent University - iMinds)
firstname.lastname@ugent.be

Abstract. Linked Data interfaces exist in many flavours, as evidenced by subject pages, SPARQL endpoints, triple pattern interfaces, and data dumps. These interfaces are mostly used to retrieve parts of a complete dataset, such parts can for example be defined by ranges in one or more dimensions. Filtering Linked Data by dimensions such as time range, geospatial area, or genomic location, requires the lookup of data within ordinal ranges. To make retrieval by such ranges generic and cost-efficient, we propose a REST solution in-between looking up data within ordinal ranges entirely on the server, or entirely on the client. To this end, we introduce a method for extending any Linked Data interface with an n-dimensional interface-level index such that n-dimensional ordinal data can be selected using n-dimensional ranges. We formally define *Range Gates* and *Range Fragments* and theoretically evaluate the cost-efficiency of hosting such an interface. By adding a multidimensional index to a Linked Data interface for multidimensional ordinal data, we found that we can get benefits from both worlds: the expressivity of the server raises, yet remains more cost-efficient than an interface providing the full functionality on the server-side. Furthermore, the client now shares in the effort to filter the data. This makes query processing becomes more flexible to the end-user, because the query plan can be altered by the engine. In future work we hope to apply Range Gates and Range Fragments to real-world interfaces to give quicker access to data within ordinal ranges.

Keywords: Linked Data, indexing, dimensional data, Linked Data Fragments, SPARQL

1 Introduction

Different types of Linked Data interfaces can be used to publish data to the Web, such as data dumps, subject pages, Triple Pattern Fragments (TPF), and SPARQL endpoints. Each of these interfaces have a certain expressivity, ranging from high expressive query interfaces to low expressive data dumps. The idea of ranking interfaces by their expressivity to choose a Linked Data interface by discussing trade-offs, was introduced by Linked Data Fragments (LDFS) [10], as illustrated in Fig. 1. LDFS and TPF also advocate for Linked Open Data interfaces to follow the REST principles: next to having an identifier for resources, which Linked Data promises, clients should also automatically be able to discover the controls of an interface. This way, new features can be added to such an interface dynamically, enabling clients to exploit those when discovered and supported.

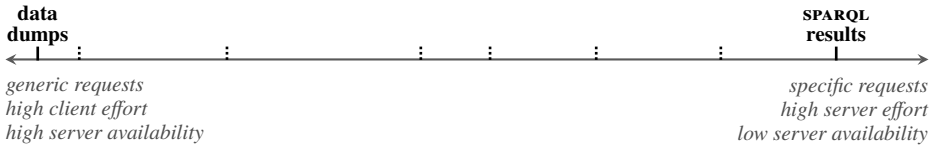

Fig. 1: The *Linked Data Fragments* axis illustrates that all HTTP interfaces offer data fragments, yet they differ in the specificity of the data they contain, and thus the effort needed to create them [10].

In order to evaluate Basic Graph Patterns, TPF was introduced, where the server only exposes a triple pattern interface. A significant amount of SPARQL queries however contain FILTER clauses [5] and because the regular TPF interface only allows clients to search by exact triple pattern matches, it requires these clients to evaluate all SPARQL FILTERS client-side. Therefore, an extension for filtering has been added to the TPF interface [8], which works for substring filtering. This extension thus improves the efficiency of SPARQL query evaluation within the TPF framework for queries with high FILTER selectivity at the cost of a large server-side index.

Filtering on *multidimensional ordinal values*, like numbers, dates, geospatial locations and even genomic ranges, yields the same problem as substring filtering. It is possible that a highly selective ordinal filtering is required by the query. Using the regular TPF interface, the client would have to first download all data before being able to apply a filter. This leads to slow query execution times for highly selective filters. To this extent, we introduce a method for publishing any set of triples in fragments which assist the client to perform the filtering. This way, we allow a generic client to query for data within any kind of ordinal range. In the same way SPARQL endpoint maintainers today would add indices on certain literal types, data publishers now would need to expose the data within the right fragments using our method. Compared to SPARQL endpoints, we now have a less expressive interface which reduces the potential load on the server, in turn reducing the number of possible server requests and makes filter-processing more cacheable.

In the next section, we discuss related research on which our solution is based. After that, Section 3 gives a generic overview of our proposed index structure. Next, Section 4 includes a theoretical analysis of the index for both client and server. Finally, Section 5 discusses the conclusions of this work with further research opportunities.

2 Related Work

Linked Data Fragments is a conceptual framework using which Linked Data publication interfaces can be compared. These interfaces are compared by the way they represent data, their controls to specify the data, and the metadata they use to describe this data. SPARQL interfaces for example allow very specific data to be selected using a generic control, the SPARQL language, which requires a high server effort. Data dumps, for example, have a very simple control over which the whole dump can be selected. This requires a high client effort to find specific data but a low server effort. TPF can be seen as a trade-off between these two extremes where it requires some work from both the client and server.

Triple Pattern Fragments (TPF) [10] interfaces allow clients to query data based on triple patterns. When such a request is performed, the server returns a Triple Pattern

Fragment, which consists of *data* triples that match the client's triple pattern. These TPFs are *pageable* resources to reduce response size, which means that when a lot of triples match with the client's triple pattern, the client will need to request multiple *pages* to completely retrieve the set of matching triples. These TPFs also include *metadata* and *controls* next to the *data*. The metadata includes a description of the interface and the TPF, while the hypermedia controls enable access to other TPFs on the interface and access to the next and previous pages of the current TPF. Indexing structures are typically used in data retrieval systems to reduce lookup times. The interface-level index structure we introduce is similar to tree-based indexes like B-Trees and B+Trees [2] that are used to store and lookup data with a certain order.

For specific dimensions such as time, the *Memento* protocol [6] can be used to expose access to historical versions of HTTP resources by extending HTTP with content negotiation in the datetime dimension. Memento adds a new link to resources in the HTTP header, named the *TimeGate*, which acts as the datetime dimension for a resource. It provides a list of temporal versions of the resource which can be requested. In this work we generalize this concept of a *Gate*, to provide access to a collection of resources.

In the area of distributed database systems [4], the concept of *data fragmentation* exists. Fragmentation is used to assign subsets of data to different database systems. This is useful when datasets become too large for one system or when subsets of data benefit from being assigned to certain systems. Two types of fragmentation are typically distinguished for relational data: a) *horizontal fragmentation* refers to splitting up the rows of a table, where groups of full tuples are assigned to the same fragments. b) *vertical fragmentation* refers to splitting up the columns of a table, where certain tuple elements of all tuples are assigned to the same fragments. In this paper, we will reuse the concept of horizontal fragmentation, which we will refer to as simply *fragmentation*.

3 Multidimensional index

In order to allow clients to navigate to specific parts of a dataset for a certain n-dimensional range, we present a method for adding an index for an RDF class to any Linked Data interface in this section. First, we introduce new terminology, after which we explain a model for representing the interface index. Next, we discuss how a client is supposed to use this index. Finally, we explain how to instantiate this model for any class.

3.1 Terminology

Definition 1. *An* n-dimensional resource *is an RDF resource with n properties whose ranges are strict linearly ordered sets. An RDF class of n-dimensional resources is called an* n-dimensional class*.*

In Definition 1, we introduce the terms n-dimensional resource and n-dimensional class. A radio's tracklist element for example can be modeled as a 1-dimensional class as it is identified by a timestamp in the temporal dimension, which is 1-dimensional. A sensor measurement for example is a 3-dimensional resource, because it has one temporal dimension and two geospatial dimensions.

Definition 2. *A* Linked Data interface *exposes certain* Linked Data Fragments *of a Linked Dataset. A selector can be applied to an interface to get a fragment, which is uniquely defined by that selector.*

Definition 3. *We use the term* Range Fragment *to refer to a Linked Data Fragment [10] that has an interval as selector, which applies to dimensional classes at one of its n dimensions.*

An n-dimensional RDF class has n levels of Range Fragments, where a Range Fragment at level i acts as a selector for the dimensional class at dimension i. The Range Fragments at level i can also be seen as being *fragmented* at dimension i by its selectors.

Definition 4. *A* Range Gate *is a Linked Data interface through which Range Fragments can be selected by interval. This interface selects all Range Fragments whose interval overlap with the Range Gate's interval.*

In order to gain access to Range Fragments at level i, a Range Gate at level i exposes the list of these Range Fragments at level i. This collection of Range Fragments at the same level have no restrictions with relation to each other, their identifying interval selector may partially, completely or not overlap. The number of possible Range Fragments is unlimited and can vary through time. Ideally, the different Range Fragments must have an equal size to have a balanced index. The fragmentation will therefore strongly depend on the dimensional class.

For example, if we have Range Fragments for 1-dimensional resources where one area has a large amount of elements compared to the complete domain, then the Range Fragments for this area will have a smaller identifying selector interval than the other Range Fragments.

3.2 Index Model

Our index model is a tree structure of Range Fragments and Range Gates which is exposed at interface-level so that clients can navigate through it to locate dimensional resources in a certain range. The root of our tree is the original interface to which we want to add this index. We can interpret this root element as the Range Fragment at level -1, indicating that no dimension of the dimensional class can be selected at this level. Each Range Fragment at level i has a link to Range Gate at level $i+1$ if $i+1 < n$. Fig. 2a shows an overview of an n-dimensional index model.

The Range Fragment at level -1 must provide a way to test whether or not a certain triple pattern applies to each index. Therefore it should either supply a list of triple patterns, or a control to a simple interface to test whether or not a certain triple pattern is present in the index.

This index can either be *static* or *dynamic*. Static indexes are made once and should be used when n-dimensional resources can not be added to the dataset after initial publication. Dynamic indexes on the other hand allow for n-dimensional resources to be added after initial publication. Depending on the use case, this index can either be used as the single point in which the n-dimensional resources can be fetched, which we call a *primary index*, or as an alternative way next to the main dataset to fetch this data, which

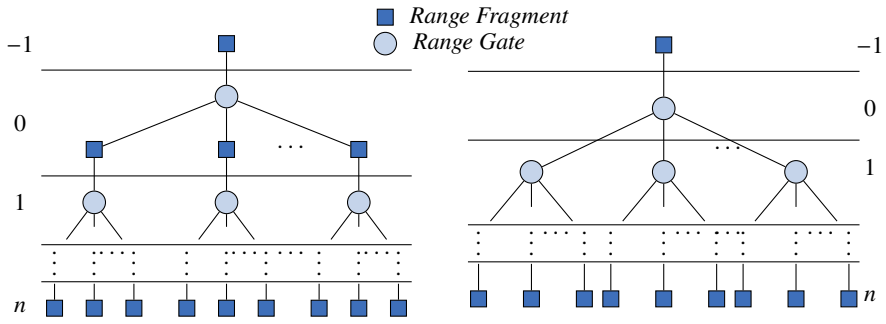

Fig. a: The n-dimensional index model.

Fig. b: The reduced n-dimensional index model.

we call a *secondary index*. The latter is advised when needing to remain compatible with clients who are not capable of using this index.

The fact that non-leaf Range Fragments exist can have an impact on the complexity of the internal server index data structure because it has several possible entry points to the same data. Because of this, a *reduced multidimensional index* can be used where Range Fragments only exist at the very last level of the index, and all internal Range Gates at level i will immediately link to Range Gates at level $i + 1$. As a consequence, the client will have to navigate to the last level of the tree before it will be able to start requesting n-dimensional resources. Fig. 2b shows an overview of a reduced n-dimensional index model.

The main reasoning behind the idea of exposing this index to the client is that these index navigations become cacheable and that index handling is moved from the server to the client. A client can for example navigate through the n-dimensional index by a vector $v_1 = [e_0, e_1, ..., e_n]$ where each element e_i specifies the Range Fragment index that should be taken at level i. If the client later has to navigate through this index again using vector $v_2 = [f_0, f_1, ..., f_n]$, then it could reuse parts of this navigation path. More specifically, the vector $v_s = [e_0, e_1, ..., e_i]$ where with $P(x) = \bigwedge_{k=0}^{x}(e_k = f_k)$, $i = j : 0 \leq j \leq n \wedge (\forall x \leq j \Rightarrow P(x)) \wedge (\forall y > j \Rightarrow \neg P(x))$ is the part of the navigation path that can be reused from cache. When many consecutive Range Fragments need to be requested by the client, this shared vector v_s will typically be very long which will lead to frequent navigation path element cache hits.

3.3 Client Access

The goal of our multidimensional index is to improve the selectivity of the process to find n-dimensional ordinal data. When a Linked Data interface exposes such an index for an n-dimensional class, it should add a link to this index in its metadata so that clients can discover this index. In practise, a client can exploit this index when for example a FILTER clause is defined in a SPARQL query to select certain values in a time range.

When a client needs to request data in a certain n-dimensional range, it must first determine if one of its filters can be performed using an index from the server. It can do this by testing index-applicability for all n-dimensional classes over which a filter

is defined using the hypermedia controls related to all indexes on the server. If a test succeeds, the client should find the link to the corresponding index in the server's metadata. If such an index is found, the client can use this index to more efficiently filter on these values, otherwise it should fall back to its default method of working.

Depending on the number of dimensions of the class, the client will need to perform more or less requests to find the relevant data. The client algorithm starts at dimension -1, which is the original entrypoint at the LDF server. At this initial Range Fragment, a link to the Range Gate at dimension $i + 1$ exists, which will be dimension 0 in this case. The client must then request this Range Gate entrypoint, interpret its controls and fill in the interval to filter on at dimension i. This will result in a paged list of Range Fragments at level i that overlap with the given interval. From this list, the Range Fragments must be selected that are at least partially contained in the client's original range for dimension i. Depending on the number of dimensions n, this process must be repeated until $i = n$ for each of the selected Range Fragments at level i. When more than one of those Range Fragments are selected, these can be recursively handled in parallel by the client. When a Range Fragment at level n is reached, its contained dimensional resources are highly selective. Because the data provider is responsible for defining the Range Fragment's intervals, the client's original n-dimensional range might not completely apply to the data contained in this Range Fragment, so the client will still have to filter out some of the data afterwards.

If the client notices that a certain Range Fragment at level $i < n$ has a very small number of triples, it can start fetching that data instead of having to navigate deeper in the index. This optimization is only possible when using a non-reduced multidimensional index, otherwise the client must always go to the n^{th} level of the index before requesting dimensional resources.

3.4 Model Instantiation

n-dimensional resources have n properties whose ranges are strict linearly ordered sets. These resources are therefore defined by an ordinal variable of dimensionality n, and exist at a certain n-dimensional point. The dimensionality n determines the amount of levels required in the index.

The multidimensional interface specification [7] is ongoing work, in which we explain how to describe a multidimensional interface with metadata and how it can be used with hypermedia controls. We use the Hydra Core Vocabulary [3] to define these controls in RDF. Range Fragments expose controls for searching triples by triple pattern, as is described by the TPF specification [9]. Range Gates expose similar controls to search through Range Fragments by overlapping interval. These controls allow clients to automatically discover how to navigate in this index and find the data they need.

In order to determine the distribution of resources in Range Fragments, we must develop a fragmentation strategy for the dimensional resources. At each dimension i of the dimensional class of the dimensional resources, the values must be fragmented in such a way that their Range Fragments at this level are balanced. This balancing requirement includes Range Fragments at the same level whose parent Range Gates are not equal. Balanced Range Fragments improve the effectiveness of the index. Range Fragments are balanced when they contain approximately the same amount of data.

Another requirement for good fragmentation strategies is that they do not require the re-distribution of information in the Range Fragment when new data is added. This is because we want this index to be cacheable by clients.

We distinguish the following categories of fragmentation strategies:

Simple The most straightforward but not always very intelligent strategies like for example equally sized n-dimensional intervals.

Stochastic Depending on the n-dimensional distribution of the dimensional resources, a prediction for the optimal intervals can be determined by for example selecting a fixed number of equally-distanced percentiles in this distribution and deriving their n-dimensional values.

Clustering Clustering algorithms can be used to either cluster the dimensional resources in a fixed or variable number of clusters which have a low inter-cluster distance in the n dimensions. Depending on the dataset being static or dynamic, respectively batch or online clustering algorithms should be used.

Dynamic indexes might require a more simple fragmentation strategy than static indexes if this strategy requires complex operations for determining its target Range Fragment. Static indexes can do all required fragmentation operations in an offline preprocessing step.

This multidimensional index only defines an addition to the server on interface-level, the actual internal implementation can consist of anything, as long as it supports the index interface. A simple way, and in some cases an efficient way, of exposing this index is by pre-generating the Range Fragments of the index so that they can be accessed as static resources. When the dataset becomes larger, this naive way becomes inefficient because of the high number of possible resources to be generated. This is why internal data structures like B+ trees, red-black trees, or other binary tree variants can be used to index dimensional resources. Red-black trees can for example be used when the index structure can fit into the main memory of the server, otherwise B-trees or B+ trees can be used. If the multidimensional index is primary, the dimensional resources can be stored directly into the search tree, otherwise the tree can simply hold a reference to the dimensional resources in the actual dataset to avoid storing redundant information.

4 Theoretical Analysis

In this section, we will discuss the theoretical effects of a multidimensional index on the performance of both client and server, after which we will discuss possible optimal parameters for these two.

4.1 Client

The goal of the multidimensional index is to improve the selectivity for the client when querying dimensional resources. The index will only be used when the client has to filter over the range of the dimensional resources. In these cases, the client will look for the dimensional resources through the index instead of through the original interface.

Equation 1 shows the worst case for the total number of requests when requesting data for only one Range Fragment leaf through this index. The first term represents the single

request that is required to retrieve the root element of the index and discover the index controls. The second term represents the number of requests that is required to navigate to the deepest Range Fragment. The last term identifies the number of requests needed to retrieve all elements of the deepest Range Fragment. $range\_fragment\_elements(i, j)$ returns the number of dimensional resources in the jth fragment at level i. In this context, $count\_range\_fragments(i, j)$ is the amount of children the jth Range Gate at level i has, which depends on the used Range Fragment fragmentation strategy. $range\_gates(i)$ is the number of Range Gates on level i, which equals 1 if $i = 0$ and $\sum_{j=0}^{range\_gates(i-1)} count\_range\_fragments(i-1, j)$ otherwise. The best case for the number of requests is simply the empty index, where $n = 0$ and all data must be retrieved from the main dataset.

$$
\begin{aligned}
&O(\#requests) = 1 \\
&+ \sum_{i=0}^{n} \left(1 + max_{j=0}^{range\_gates(i)} \left\lceil \frac{count\_range\_fragments(i, j)}{gate\_pagesize} \right\rceil \right) \\
&+ max_{i=0}^{n} max_{j=0}^{range\_gates(i)} \frac{range\_fragment\_elements(i, j)}{fragment\_pagesize}
\end{aligned}
\tag{1}
$$

In practice, the n-dimensional interval from the original user's query can overlap with multiple Range Fragments. Assuming that the user's interval spans s Range Fragments, the simplified equation for the worst case of number of requests to fetch this data is $s * O(\#requests)$. Client-side caching of the index structure when it is being requested can greatly reduce the number of requests defined by this simplified equation. More specifically, when caching the index structure, the second term of Equation 1 will experience a logarithmic reduction when more requests are being done in the worst case. In the best case, when the Range Fragments that must be fetched are consecutive and have the same Range Gate parent, Equation 1 for s requests becomes the one in Equation 2.

$$
\begin{aligned}
&O(\#requests) = 1 \\
&+ \sum_{i=0}^{n} \left(1 + max_{j=0}^{range\_gates(i)} \left\lceil \frac{count\_range\_fragments(i, j)}{gate\_pagesize} \right\rceil \right) \\
&+ s * \max_{i=0}^{n} max_{j=0}^{range\_gates(i)} \frac{range\_fragment\_elements(i, j)}{fragment\_pagesize}
\end{aligned}
\tag{2}
$$

The second term in Equation 1 can be seen as the overhead the index has for the client. These are the requests that are required to navigate through the index to find leaf Range Fragments. If the client requests many different non-consecutive ranges, this index navigation overhead will become larger. If a large fraction of the dimensional resources is requested, it might be better to request this data through the regular interface if the index is secondary. The client can detect this when the number of filtered Range Fragments through the Range Gate at level 0 is near the total number of Range Fragments this Range Gate offers. This decision can be formalized as $skip\_index = \#total\_RF(0) - \#filtered\_RF(0) > d$. This constant d is representative for the size of the index, and can be approximated as the total number of Range Gates that the index has at level 1.

The third term in Equation 1 represents the maximum amount of pages a Range Fragment has. Assuming the data provider has a good Range Fragment fragmentation strategy, this index will be balanced and this maximum will be near the average number of pages a Range Fragment has. This means that this third term will be typically lower when there is a better fragmentation strategy in use. Again, this number will also be lower with a larger pagesize. This same reasoning can be applied to the second term, which will be decreased with a better fragmentation strategy and a larger pagesize. A larger pagesize will reduce the amount of requests, but this means that a higher load will be placed on the server when just the first page of a fragment must be requested for its metadata. This is however not a valid reason not to use larger pagesizes, since it could be made possible to fetch metadata without fragment data.

In our example, traditional TPF interfaces provide sequential paged access to fragments. Due to the hierarchical nature of our multidimensional index, fetching dimensional resources becomes highly parallelizable. This is because our hierarchical index structure makes it possible to know in advance which Range Fragments the client will need. Each Range Fragment can in fact be handled in parallel, so the higher this number of Range Fragments the user needs to request, the higher this parallelizability.

4.2 Server

When a multidimensional index is added to the interface of a server, the expressivity of the interface is increased, which will potentially increase the server's workload. As mentioned in Section 3.4, the multidimensonal index is only defined on interface-level, so the actual performance of the index will strongly depend on the internal datastructure used to store the index. We will make a distinction between the Range Fragment requirements and the Range Gate requirements.

Range Fragments contain the actual data. If the index is primary, the dimensional resources can be stored in the internal index itself. If the index is secondary, the data should only be stored on one location, and other places should be able to internally fetch data from that location. If the index is reduced, meaning that data can only be retrieved from level n in the index, this data structure can be chosen for optimized access for only n-dimensional ranges instead of $1..n$-dimensional ranges.

This Range Gate data should support efficient lookup by n-dimensional ranges, which could be achieved using n layers of B+ trees, red-black trees or other binary search tree variants. For example, a B+ tree at layer i of our multidimensional index could represent the index structure of our dimensional resources at dimension i. Each B+ tree leaf then contains a pointer to a B+ tree at layer $i + 1$ or a list of Range Fragments if $i = n$.

The number of dimensions, the fragmentation strategy and the used search tree will determine the required resources to maintain this internal index on the server and the lookup efficiency. The total number of fragments $\sum_{i=0}^{n} range\_gates(i)$ is the amount of internal search trees that will need to be stored, the higher the number of fragments at dimension i, the larger the trees at level i will be, and the more storage size these will take. Equation 3 represents the size of an internal index, where $tree\_size(n)$ represents the total size required for a search tree with n nodes and *pointer_size* represent the maximum

size of a pointer to a Range Fragment.

$$O(internal\_index\_size) = \sum_{i=0}^{n} \sum_{j=0}^{range\_gates(i)} \Big(tree\_size(count\_range\_fragments(i,j)) + count\_range\_fragments(i,j) * pointer\_size \Big) \quad (3)$$

The Range Fragments will not require any additional storage, because these can be stored in traditional triple or quad stores.

4.3 Optimal parameters

A good index should minimize both the total amount of requests the client should perform and the total storage size of the internal server-side index, therefore it should minimize respectively Equation 2 and 3. This means that we should minimize both $count\_range\_fragments(i, j)$ and $range\_gates(i)$ if we look at the common factors. This is however not possible, as they contradict each other. Decreasing the first factor, which represents the total amount of elements in a Range Fragment at level i, will increase the second factor, which equals the total amount of Range Fragments at level $i - 1$, and the other way around. This is because decreasing the total amount of elements in a Range Fragment at level i means that you have to increase the number of Range Fragments at level i over which the data at dimension i is being fragmented. This increase of Range Fragments thus exists for all i, and thus also for $i - 1$. So decreasing $count\_range\_fragments(i, j)$ will increase $range\_gates(i)$. Depending on the page size, the optimal balance between these two parameters will depend on the used fragmentation strategy of the dataset.

5 Conclusions

In order for clients to efficiently query datasets describing n-dimensional resources, we introduced a theoretical extension for any Linked Data interface with Range Gates and Range Fragments. By adding hypermedia controls, supporting clients can automatically discover this fragmented dataset. The movement of responsibility for filtering, ordering or window-functions from server to client, lowers the server load, as the HTTP cache can be used more efficiently. The client can also reuse parts of its paths through the index when requesting consecutive fragments. An important factor when building a multidimensional index for an RDF class is how to fragment its instances in n dimensions.

Our generic index can, with the necessary changes, be implemented for many different use cases. For example, a 1-dimensional temporal index could be instantiated for storing train departure delays with a stochastically optimal fragmentation strategy based on public transit API usage logs [1]. Secondly, substring filtering support could be exposed using a 1-dimensional index instead of the single filter control that is currently being used [8].

We discussed the theoretical client and server performance for this index. The more selective the n-dimensional filtering by the client, the more efficient this index will become when compared to non-indexed-access. There is a point at which it becomes more efficient for the client to query the data without the index, this will be when the filter has a very low selectivity. The client and server will both benefit from smaller, and thus more selective, Range Fragments, while at the same time this results in a higher number of Range Fragments which will thus require a larger index structure i.e. a larger number of Range Gates. This larger number of internal Range Gates, *increases* the required number of requests by the client and the storage requirements for the server, which is why a balance between the Range Fragment size and count should be determined.

These parameters introduced in our theoretical analysis offer new trade-offs for data publishers in offering optimized access to n-dimensional Linked Data on the Web, as they can choose how much work the client and server should each do for index handling. This trade-off can even be determined dynamically, by, for example, only providing access to a highly-fragmented index during low server load and a low-fragmented index during peak-times. Dynamic fragmentation will however increase the complexity of index handling and caching by clients.

One future optimization for the multidimensional index to further reduce the required number of requests and processing times is by guaranteeing the ordering of triples inside Range Fragments. This would make it possible for the client to make attempts to skip certain pages of a Range Fragment if it only has to look for a subset of that Range Fragment.

The proposed index currently only allows fragments to be divided up linearly per dimension. These are linear because our Range Fragment identifiers are intervals, which are identified by two points in one dimension and thus can be represented by a straight line across this dimension, which we will call a *constant border function*. For some use cases, a division using more general *polynomial border functions*, which may overlap several dimensions, might result in a better fragmentation of data. Figure 3 shows an example of two-dimensional data that may benefit from polynomial border functions. This, however, goes beyond the capabilities of our hierarchical index with one level per dimension. Our index could be adapted so that it can *combine* several dimensions into one layer as opposed to considering the dimensions independently, so that polynomial functions become possible. This will, however, increase the cost of the server as it will give more responsibility of the index evaluation to the server.

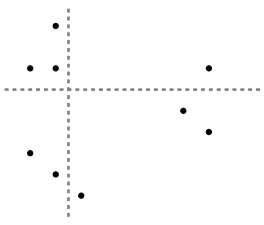

Fig. a: Constant border functions not capable of evenly dividing the data.

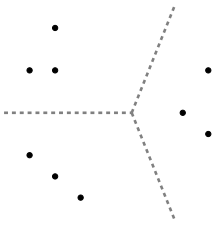

Fig. b: Polynomial border functions capable of evenly dividing the data.

Fig. 3: Fragmentation of two-dimensional data.

In future work, we plan on making a generic implementation of this multidimensional index for TPF servers with a generic TPF client extension that can consume it. We plan on using this index for both offering efficient access to dynamic data, like train delays, and access for ranges on genomes. This index should be tested thoroughly for both the generic case as for specific use cases and the statements made in our theoretical analysis should be tested.

References

1. Colpaert, P., Chua, A., Verborgh, R., Mannens, E., Van de Walle, R., Vande Moere, A.: What public transit API logs tell us about travel flows. In: Proceedings of the 25th International Conference Companion on World Wide Web. pp. 873–878. International World Wide Web Conferences Steering Committee (2016)
2. Comer, D.: Ubiquitous b-tree. ACM Computing Surveys (CSUR) 11(2), 121–137 (1979)
3. Lanthaler, M.: Hydra core vocabulary. Unofficial draft, Hydra W3C Community Group (Sep 2014), http://www.hydra-cg.com/spec/latest/core/
4. Özsu, M.T., Valduriez, P.: Principles of distributed database systems (2011)
5. Rietveld, L., Hoekstra, R.: Man vs. machine: Differences in SPARQL queries. In: 4th USEWOD Workshop on Usage Analysis and the Web of Data, ESWC (2014)
6. de Sompel, H.V., Nelson, M.L., Sanderson, R., Balakireva, L., Ainsworth, S., Shankar, H.: Memento: Time travel for the web. CoRR abs/0911.1112 (2009)
7. Taelman, R.: Multidimensional linked data interfaces. Unofficial draft, Ghent University - iMinds - Data Science Lab (Jun 2016), https://w3id.org/multidimensional-interface/ontology
8. Van Herwegen, J., De Vocht, L., Verborgh, R., Mannens, E., Van de Walle, R.: Substring filtering for low-cost linked data interfaces. In: The Semantic Web–ISWC 2015, pp. 128–143 (2015)
9. Verborgh, R.: Triple pattern fragments. Unofficial draft, Hydra W3C Community Group (Aug 2014), http://www.hydra-cg.com/spec/latest/triple-pattern-fragments/
10. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. Journal of Web Semantics 37–38, 184–206 (2016)