Space Odyssey - Efficient Exploration of Scientific Data

Mirjana Pavlovic[¶], Eleni Tzirita Zacharatou[¶], Darius Sidlauskas[¶], Thomas Heinis[†¶], Anastasia Ailamaki[¶]
[¶]*École Polytechnique Fédérale de Lausanne, Switzerland*
[†]*Imperial College, London, United Kingdom*

ABSTRACT

Advances in data acquisition—through more powerful supercomputers for simulation or sensors with better resolution—help scientists tremendously to understand natural phenomena. At the same time, however, it leaves them with a plethora of data and the challenge of analysing it. Ingesting all the data in a database or indexing it for an efficient analysis is unlikely to pay off because scientists rarely need to analyse all data. Not knowing a priori what parts of the datasets need to be analysed makes the problem challenging.

Tools and methods to analyse only subsets of this data are rather rare. In this paper we therefore present Space Odyssey, a novel approach enabling scientists to efficiently explore multiple spatial datasets of massive size. Without any prior information, Space Odyssey incrementally indexes the datasets and optimizes the access to datasets frequently queried together. As our experiments show, through incrementally indexing and changing the data layout on disk, Space Odyssey accelerates exploratory analysis of spatial data by substantially reducing query-to-insight time compared to the state of the art.

1. INTRODUCTION

In astronomy, biology, neuroscience and other disciplines, scientists are increasingly overwhelmed by the amount of data they have at their disposal. With advances in sensor technology, i.e., increased resolution, and supercomputing for large-scale simulations, the amounts of data scientists have to analyse grow rapidly. Today's tools are frequently inadequate to analyse the data and answer key questions as already executing simple queries such as spatial range queries becomes challenging given the amount of data. Datasets can, of course, be indexed a priori to accelerate access but the areas analysed are rarely known beforehand and also only touch a subset of the entire dataset, making indexing an undue overhead.

In neuroscience, for example, scientists need to explore multiple massive datasets originating from different sources [11] to investigate particular areas of the human brain. The data in this use case is spatial and originates from different instruments (e.g., patch clamp, brightfield spectroscopy, MRI) of different resolutions. To perform an analysis, they need to query small parts of different combinations of datasets, each of a size in the order of Terabytes. What areas of the datasets they need to access and what combinations of them are not known a priori. It is consequently unclear what parts of what datasets need to be indexed. It is however clear that fully indexing all datasets introduces considerable overhead which is unlikely to pay off.

More formally, the problem is the efficient exploratory analysis of multiple spatial datasets through the execution of range queries: given n datasets and a subset of datasets $m \subseteq n$, scientists need to efficiently execute a spatial range query q on each of the datasets m. What combinations of datasets m will be queried together and what spatial ranges q will be accessed is not known beforehand. The challenges thus are twofold (a) what areas in the datasets are accessed and (b) what datasets are accessed together.

Multiple spatial indexes have been developed to accelerate access to spatial datasets addressing the first challenge [3]. All of them, however, require the whole dataset to be indexed at once. Incremental approaches to indexing (or reorganising data layout) have been developed for one-dimensional data stored in main memory [7, 8], but not for spatial data on disk. The first challenge of incrementally indexing spatial data on disk as well as the second, accelerating access to multiple datasets queried together, remain unaddressed in literature to the best of our knowledge.

Space Odyssey, the approach we develop, addresses both challenges and enables the efficient exploration of multiple spatial datasets. While the datasets are being queried, it incrementally indexes spatial datasets (based on space-oriented indexing) to accelerate access to the datasets in general and to the areas frequently queried in particular. At the same time, it reorganises the layout of the data on disk so that parts of the datasets queried together can be retrieved more efficiently. By incrementally indexing and reorganising the data, Space Odyssey accelerates explorative analysis of spatial data by substantially reducing query-to-insight time: Space Odyssey answers up to several hundred queries (more than half the queries of the benchmark) by the time the fastest existing approach has merely indexed the data.

In the remainder of this paper we give a brief overview of the related work in Section 2 before we discuss our approach in detail in Section 3. We subsequently analyse our approach experimentally in Section 4 before we conclude in Section 5.

2. RELATED WORK

Several approaches have been developed in recent years to adapt the data layout in response to incoming queries.

To accelerate access to the data, database cracking [7, 8] iteratively refines the layout of the data in memory. Cracking essentially amortizes the cost of index building over query processing and with each query the physical layout is refined to accelerate subsequent

ExploreDB'16, June 26-July 01 2016, San Francisco, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4312-1/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2948674.2948677

queries. It partially sorts the one-dimensional data in memory according to the queries.

Similarly, incremental indexing [4, 5] chooses and creates indexes as a side-effect of query processing. A user neither configures or creates indexes nor does she provide a representative workload. Instead, based on the queries executed, an adaptive index is only partially materialized and optimized such as to fit the current workload and storage budget. As queries arrive, the index is adapted on the physical level to suit the workload. Known adaptive indexing techniques, however, require all data to be loaded upfront.

Finally, several approaches skip pre-processing to reduce the cost of raw data querying. NoDB [1] accesses CSV data in situ to adaptively build positional and binary caches as a side-effect of query execution. RAW [9] extends NoDB and adapts its access layer using code generation techniques.

Numerous approaches have been developed to index spatial data [3]. Almost all spatial indexes, however, require the entire dataset to be loaded upfront and do not adapt to the query workload. One representative exception are adaptive index structures [14] which rearrange the nodes of data-oriented hierarchical indexes (including the spatial R-Tree [3]) in response to queries so that they can be accessed sequentially on disk. However, this reorganisation is performed only after the index has been fully built.

3. SPACE ODYSSEY

Space Odyssey enables exploratory access to multiple spatial datasets such that scientists can efficiently access particular areas in combinations of spatial datasets. Crucially, our novel approach enables efficient access without having to preprocess the data. Instead, Space Odyssey uses incoming queries to reorganize the physical layout of the data to better serve queries.

First, to enable efficient access to precisely the areas queried in individual spatial datasets, Space Odyssey incrementally indexes the datasets. Second, to better support querying the same areas in different datasets, it adapts the physical layout on disk, storing together the areas that are queried together to accelerate retrieval.

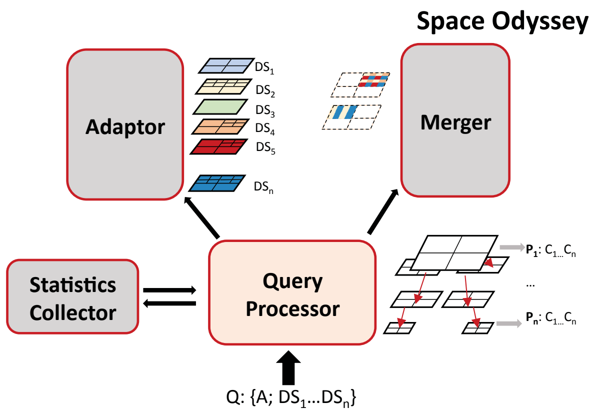

Figure 1: Space Odyssey: components, data structures and a snapshot of the physical layout.

Figure 1 illustrates the architecture of Space Odyssey – its components, data structures and a snapshot of the physical layout. The Adaptor is responsible for the incremental indexing and the Merger performs operations related to the physical layout. Finally, the Query Processor orchestrates the overall query execution process using information provided by the Statistics Collector.

3.1 Incremental Indexing

Indexing all datasets a priori has the major drawbacks that (a) scientists must wait until all data is indexed before they can start to query and (b) data that is never queried is indexed in a time-consuming process.

Space Odyssey therefore uses incremental indexing where in every step (with every query) we additionally refine the index structure in the frequently queried "hot" areas to accelerate future queries. At the same time, to keep the overhead of incremental indexing low, we use space-oriented indexing, as it introduces minimal processing overhead (compared to data-oriented partitioning [3]).

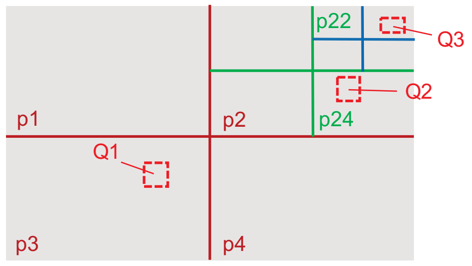

Figure 2: Incremental indexing strategy (in 2D).

3.1.1 Refinement Concept

More precisely, Space Odyssey incrementally builds an Octree [3] on each dataset queried. The Octree is the index of choice since we want to introduce minimal overhead during the query execution and thus, we split each dimension to a minimal number of partitions which corresponds to 2^d partitions in d-dimensional space. Figure 2 illustrates the indexing process with $d = 2$, i.e., 4 partitions per level. The indexing process starts with the first query Q1 where Space Odyssey partitions the space uniformly into four partitions (p1, p2, p3, and p4). It scans the dataset and assigns each object to the partitions it overlaps with. When the second query arrives (Q2), Space Odyssey identifies the partitions that it intersects with (only p2 in our example), refines this partition, i.e., divides p2 into four sub-partitions (p21, p22, p23, and p24) and reassigns its objects to the new partitions. In the same process it checks for the objects in the qualifying new partitions whether or not they are inside Q2. Space Odyssey applies the same procedure for query Q3 and all subsequent queries.

Space Odyssey refines partitions to curb the amount of data retrieved and checked for intersection with the query. Otherwise, entire massive partitions need to be checked even only for a small query. Intuitively, we want the partition size to approximate the query size. Then, in the best case, a query hits only one partition which covers just the queried range so that a single sequential scan of the partition retrieves all required objects. In the worst case, 2^d partitions are intersected by the query. Refining a partition further only incurs unnecessary processing overhead (the actual refining as well as retrieving and scanning multiple resulting partitions). Therefore, to control the degree of refinement Space Odyssey uses a *refinement threshold* (rt). A partition is refined following the execution of a query if the ratio $\frac{V_p}{V_q} > rt$ where V_p and V_q are partition and query volumes, respectively.

With this incremental refinement strategy the overhead of building the index and reorganizing the data on disk is spread over several queries. Areas frequently queried will be indexed fully, i.e., very fine granular such that range queries in these areas can be executed efficiently, as efficient as if executed on a fully built Octree.

Areas previously untouched will be partitioned at a coarser granularity thus queries in these areas can also benefit from the adaptive partitioning performed due to previous queries.

3.1.2 Optimizations

In case the query size is significantly smaller than the partition currently hit, it might require a considerable number of queries until the partition is refined enough. We can compute by how many queries a partition needs to be hit before it reaches the finest level of refinement (or put differently, before the Octree reaches the targeted depth) with the refinement threshold. The following equation gives the number of queries (or levels in the Octree built) required:

$$\log_{ppl}(V_p/(V_q \times rt))$$

where *ppl* is the number of partitions per level and $ppl = 2^d$ in a standard Octree. To allow for faster convergence we can set Space Odyssey to use a bigger *ppl*.

Since we use space-oriented partitioning a spatial object can intersect with several partitions which introduces additional intersection tests. To avoid object replication and thus curb the memory footprint while avoiding unnecessary comparisons, we translate the problem of indexing volumetric objects to indexing point objects by using the query window extension technique [13]. Space Odyssey assigns each object o to a partition based on o's center and keeps track of the maximum object extent (*maxExtent*) in each dimension. Then, to answer a query correctly ensuring that all intersecting objects are retrieved, its range is extended by *maxExtent* and the cells the extended query overlaps with are inspected.

Finally, Space Odyssey performs the updates in-place, i.e., it reads a partition p, refines it and uses the pages where partition p was stored for the newly created partitions. After refining p we may require more disk pages than were initially required to store p; we append these pages at the end of the file.

3.2 Combining Datasets

By building data structures incrementally we can significantly decrease the data-to-query time. At the same time we have the opportunity to optimize the placement of data structures on disk to accelerate the queries executed.

Particularly in the case where multiple datasets are analysed, apart from building an index structure incrementally for each dataset, Space Odyssey also rearranges the data on disk such that the areas in different datasets which are queried together are also stored together. Doing so allows Space Odyssey to avoid random disk access for retrieving the same area in different files and thereby accelerates access.

3.2.1 Merging Partitions

While executing queries Space Odyssey keeps statistics about the datasets queried together and the partitions retrieved from them. More precisely, given queries of the form $Q = \{A; DS_1, \ldots, DS_N\}$ where A is the area queried in datasets DS_1 through DS_N, it will store: 1) how often a given combination $C = \{DS_1, \ldots, DS_N\}$ is accessed and 2) what partitions are retrieved from C, i.e., what partitions P overlap with A.

Once the number of retrievals for a particular combination C of datasets exceeds a preset *merging threshold* (*mt*), Space Odyssey merges the data for all the partitions $p \in P$ retrieved in the context of C. It iterates over all partitions that have been queried for in C, retrieves them from every dataset $DS \in C$ and merges them on disk. Note that some of the merged partitions may be retrieved less frequently by past queries than others, but the overhead of including them in the merged file is minimal while there is a benefit in case they are accessed more frequently in the future.

Lastly, Space Odyssey merges data only for combinations of size $|C| \geq 3$ because merging is more beneficial for bigger combinations as it prevents (random) accesses to a large number of datasets.

3.2.2 Data Structures

Space Odyssey creates a new *merge file* where it stores the partitions P from different datasets queried together in a combination C so they can be read sequentially and hence more efficiently once they are again queried together. The partitions in the merge file are copies, meaning that Space Odyssey also keeps the original partitions to support efficient querying on an individual dataset DS.

For a given partition p, the merge files physically stores the objects contained in p from each dataset DS sequentially. Given for example datasets DS_x, DS_y, DS_z, Space Odyssey stores objects from DS_x on the first disk pages, followed by objects from DS_y followed by DS_z. Doing so allows to retrieve efficiently only the objects belonging to a queried subset of all datasets merged (e.g. DS_x and DS_z) by reading them sequentially while skipping over the rest (DS_y). The merge file is append-only, i.e. new partitions are always added at the end of the file.

Space Odyssey incrementally builds index structures per dataset and the same regions in different datasets may thus have a different level of refinement. In dataset DS_x, for example, the area may still only be covered by one partition p while it is divided into eight partitions in DS_y. Including copies of the unrefined partition p in merge files adds the challenge of having to refine all the copies once refinement of p is triggered by a new query, thereby introducing substantial overhead. Space Odyssey addresses this issue by only merging partitions which are at the same level of refinement. Additionally, in our current implementation the merged partitions are not refined any further.

3.2.3 Querying

To efficiently execute queries and take advantage of merge files, i.e., to decide whether to retrieve areas from individual datasets DS or from merge files, Space Odyssey maintains a directory where it keeps information about what partitions of what combinations of datasets are stored together.

Once a query $Q = \{A; DS_1, \ldots, DS_N\}$ is to be executed, Space Odyssey checks what partitions intersect with A and whether these partitions are stored in a *merge file*. There are four possibilities:
Exact merge file: if the exact combination $C_q = \{DS_1, \ldots, DS_N\}$ is stored in a merge file and contains the partitions intersecting with A, then it is used to retrieve those partitions sequentially.
Superset: if a superset $C \supset C_q$ is stored, i.e. the merge file contains more datasets than the ones requested, then the merge file will still be used. Using the merge file is more efficient than accessing individual datasets thanks to the internal organization of merge files: the objects from each dataset are organized sequentially, meaning that they can be read efficiently but also that if data from a particular dataset is not needed it can be skipped.
Subset: if a subset $C \subset C_q$ is stored, i.e. the merge file contains fewer datasets than the ones requested, then Space Odyssey uses the merge file to retrieve all data from the subset C as well as other merge files or individual files to retrieve the remaining datasets $C_q \setminus C$. The decision which of the merge files to use is based on maximizing the number of datasets already stored in a merge file and thus minimize (random) access to individual files. Space Odyssey chooses the one merge file which contains the most datasets queried for.
No merge file: if no merge file exists for a combination C, individual files are used.

3.2.4 Managing Storage Space

Space Odyssey maintains a space budget for merge files (and thus replicated partitions). Once the space budget is exceeded it removes the least recently used merge files to adhere to the budget.

3.2.5 Open Issues

Building a merged index for the hot areas where multiple data sets are queried together significantly accelerates queries but several challenges need to be addressed to fully automate the merging and maximize the performance gains. In particular, we plan to develop a cost model which indicates how to adapt the parameters (minimum size of combination to be merged $|C|$ and mt) at runtime based on the workload. Additionally, we plan to investigate the benefits of merging partitions at different refinement levels and examine alternative strategies for doing so, e.g., should all partitions be refined to the same level as the finest partition before merging or as the coarsest, or shall we allow multiple refinement levels to coexist in the merged index. Lastly, we plan to improve disk space management to avoid the replication of a dataset which is used in several different combinations whenever possible.

4. EXPERIMENTAL EVALUATION

In this section we first describe the experimental setup and methodology and then demonstrate the behavior of Space Odyssey by comparing it against state-of-the-art spatial indexing approaches using real neuroscience datasets.

4.1 Experimental Setup

Hardware Configuration: The experiments are run on a Linux Ubuntu 12.04 machine equipped with 2x Intel Xeon Processors each with 6 cores running at 2.8GHz, with 64kb L1, 256KB L2 and 12MB L3 cache and 48GB RAM at 1333MHz. The storage consists of 2 SAS disks of 300GB capacity each.

Competing Approaches: We have implemented Space Odyssey and set its configuration parameters $rt = 4$, $ppl = 64$, and $mt = 2$. Also, we consider the following three approaches:

FLAT: the state-of-the-art indexing technique for spatial range queries for which we obtained the source code from the authors [15].

RTree: the most widely used spatial indexing technique. We use an available implementation of a bulk-loaded variant of R-Tree (STR [10])[1].

As we need to index multiple spatial datasets, we implemented two strategies for each one of the above approaches: one-for-each (1fE) and all-in-one (Ain1). The first strategy, 1fE, builds one index for each dataset. To perform a query, all the indexes corresponding to the queried datasets are probed and the union of the retrieved results forms the final answer. The second strategy, Ain1, builds only one index structure containing all the spatial objects from all the datasets. To perform a query, the index is probed and items belonging to datasets which are not queried are filtered.

Grid: a static, uniform grid-based technique where the indexed space is uniformly partitioned into a fixed number of cells. We use our own implementation. The objects are assigned to the grid cells in-memory and flushed to disk when the memory buffer becomes full. Similarly to Space Odyssey, replicating objects to multiple grid cells is avoided by using the query window extension technique [13]. The configuration is set to 60^3 cells, which we determine through a parameter sweep, given the absence of heuristics.

[1]https://github.com/libspatialindex

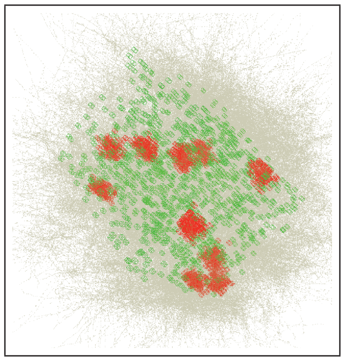

Figure 3: Clustered (red) and uniform (green) range queries on one neuroscience dataset (grey).

Software Setup: All implementations are written in C++, they are single-threaded and compiled using `g++` (v4.9.2) with the `-O3` optimization flag. The disk page size is set to 4KB. To obtain realistic run-times, where dataset sizes are significantly larger than the main memory size, all techniques are restricted to have the same main memory footprint (1GB). For all experiments only one disk is used (i.e., no RAID configuration) while the OS caches and disk buffers are cleared (overwritten with an empty file) before each query is executed (i.e., to avoid caching effect).

Datasets: We use 10 real neuroscience datasets that we obtained from our collaboration with neuroscientists in the Human Brain Project [11]. Each dataset represents a subset of neurons contained in the same brain volume. The neurons are modeled with a 3D surface mesh. Figure 3 shows a 2D projection of one brain area. An identifier is attached to each object to distinguish items belonging to different datasets. Each dataset requires approximately 5 GBs of storage on disk (and ~ 50 GBs in total).

Queries: Based on the previously described use cases, we synthetically generate queries each having a fixed volume (*qvol*) of 10^{-4}% of the queried brain volume. We use a *clustered* distribution and choose a number of *clustercenters* ($|clusterscenters| = 10$). Query centers are distributed around the cluster centers following the Gaussian distribution ($\mu = 0$, $\sigma = qvol \times 10$). For completeness and to test non-skewed cases, we also generate uniformly distributed query centers. Figure 3 illustrates the query ranges of both distributions.

To choose which subset of datasets is queried for each query range, we use a synthetic distribution generator based on Gray et al. [6]. The distributions we use are: (1) heavy hitter, (2) self-similar, (3) Zipf, and (4) uniform. These distributions have been used in other studies for similar purposes (e.g., in [2, 12]). In the heavy hitter distribution, one combination of queried datasets accounts for 50% of all possible combinations, while the other queried combinations are chosen uniformly from the remaining ones. The self-similar distribution uses an 80–20 proportion, and the Zipf distribution uses an exponent of 2. For non-skewed scenarios, we also choose the combination of datasets randomly using the uniform distribution.

4.2 Experimental Analysis

Total Processing Cost: Figure 4 depicts the total workload processing time when the number of queried datasets is increased from 1 to 9 (note that while the number of possible combinations to query increases from 10 and peaks at 252, the actually queried combinations are often smaller and depend on the distribution; also shown on the x axis). For Space Odyssey's competitors, the pro-

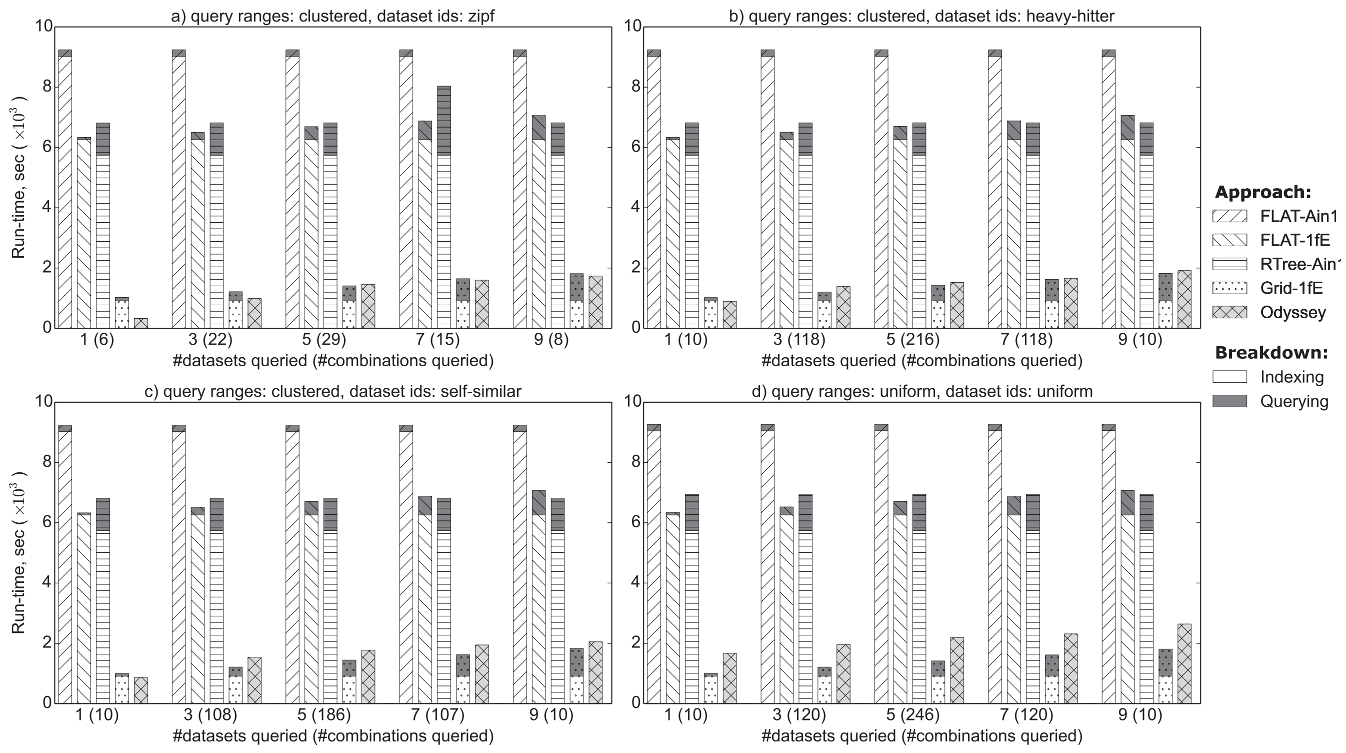

Figure 4: Performance when varying the number of queried datasets for each distribution.

cessing time is additionally broken down into indexing and querying. For Figures 4a, b, and c, we fix the query range distribution to clustered. For Figure 4d we uniformly choose both the query ranges as well as the queried datasets in order to demonstrate the worst-case performance where neither hot areas nor popular combinations exist.

First, building sophisticated spatial indexes (FLAT and RTree) takes at least 2 times longer than processing the entire workload of 1000 queries with Space Odyssey. The FLAT variants are the slowest to build among all competitors. Indexing with FLAT is slightly slower than RTree and up to $\times 5$ slower comparing to the simple uniform Grid[2]. As such, only Grid is competitive in terms of overall data-to-query time when compared to Space Odyssey. Nevertheless, by the time Grid finishes indexing the data, Space Odyssey has already answered half of the queries on average.

Second, once the related approaches have indexed the data they may process individual queries faster than in Space Odyssey. While FLAT is the slowest to build, its variants report the fastest querying times compared to other approaches – up to $\times 5$, $\times 6$, and $\times 9$ faster than the RTree, Grid, and Space Odyssey, respectively (considering just the querying time for static approaches and the total time for Space Odyssey). The important aspect of Space Odyssey however is that it has the lowest data-to-query time, because there is no need to build indexes for all the datasets in advance. The one-for-each (1fE) strategy accesses individual (smaller) indexes and only for the datasets queried. Consequently, the query processing cost increases with the number of queried datasets. The all-in-one (Ain1) strategy, on the other hand, always operates on a huge index structure and suffers from unnecessary data accesses. As such, when the number of queried datasets is less than 5, 1fE is preferred over Ain1. Space Odyssey follows a hybrid strategy, where the individual datasets are indexed adaptively (similarly to 1fE) but hot areas from different datasets are merged together (similarly to Ain1).

Third, while all related approaches are insensitive to skew in the workload, the adaptive mechanisms in Space Odyssey are able to exploit it. For example, in Figure 4, when the queried dataset combinations are coming from the very skewed zipf (a) and heavy-hitter (b) distributions, Space Odyssey quickly refines the hot areas, merges the partitions of the popular datasets together, and is often able to perform most of the queries before even Grid finishes building. This is not the case with the less skewed self-similar distributions (Figure 4c), where Grid (once its building phase is over) answers individual queries faster than Space Odyssey most of the time. When both query ranges and queried datasets are uniformly distributed (Figure 4d), Space Odyssey cannot benefit from adaptive refining and thus takes longer than Grid to process the entire workload of 1000 queries.

Query Performance: In Figure 5 we show the response time for each query in the sequence when 5 datasets are queried. In Figure 5a the queries are clustered and the queried combinations are chosen from the self-similar distribution while in Figure 5b both the queries and the combinations are chosen from a uniform distribution. We study Space Odyssey and two approaches that were previously identified as the most competitive ones (in terms of querying performance): FLAT-Ain1 and Grid-1fE. In both cases, the very first query is the most expensive for Space Odyssey as it fully scans and partitions at the first (coarsest) level the raw data files for all 5 datasets in the combination. Nevertheless, we observe that Space Odyssey converges to the speed of the fully indexed case under both skewed (Figure 5a) and uniform (Figure 5b) scenarios. As expected, however, the convergence is slower in the uniform scenario. FLAT-Ain1 has consistently better and more robust per-

[2]Favoring Grid, we assume that the optimal configuration is known. Otherwise, several builds of Grid are required to tune it.

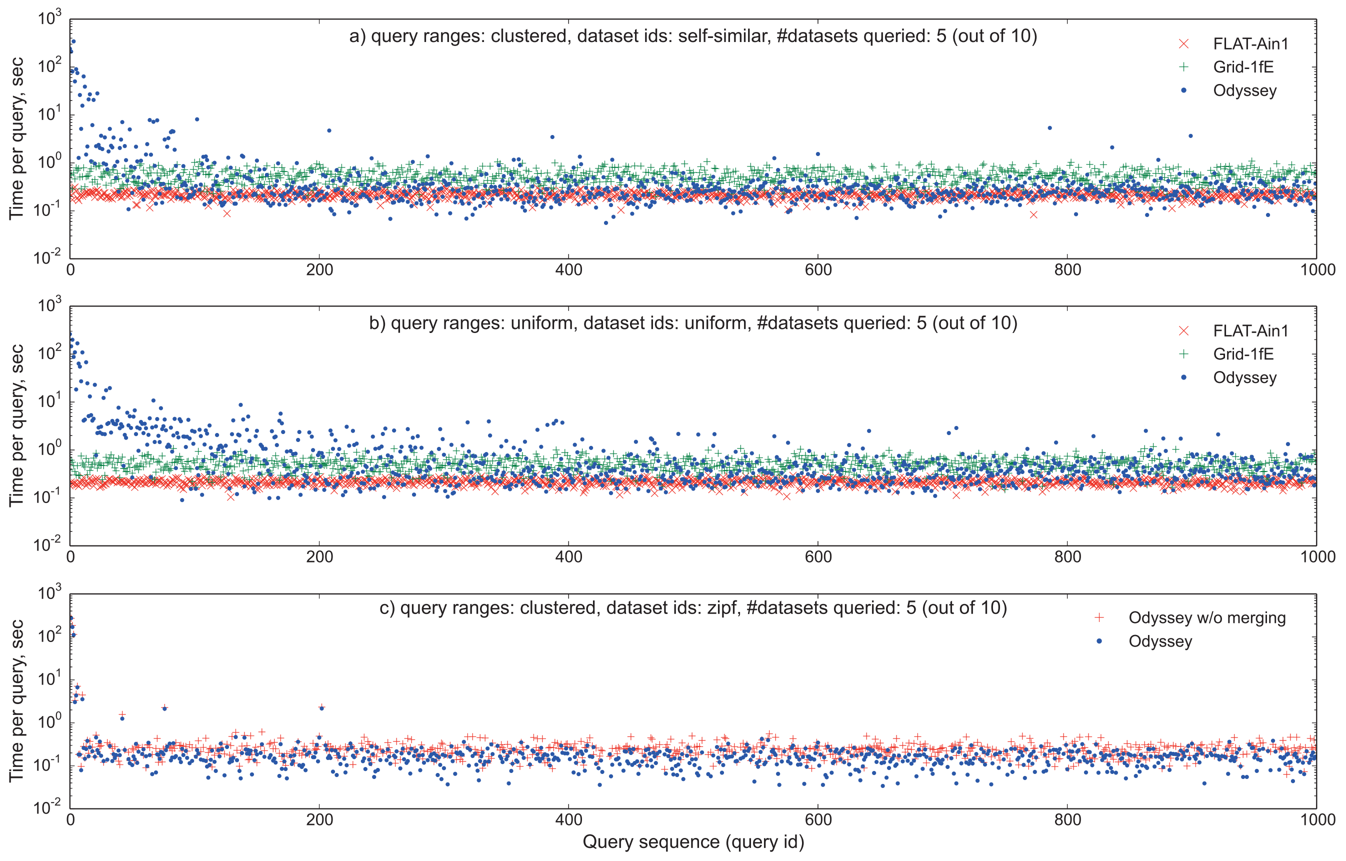

Figure 5: Query times for each query in a sequence.

formance than Grid-1fE because it is less sensitive to data skew. Once Space Odyssey has converged, its querying performance is between FLAT-Ain1 and Grid-1fE, while it performs some queries even faster than FLAT-Ain1. Finally, when an area that has not been previously refined and/or merged is queried, the querying time for Space Odyssey is still higher.

Effect of Merging: Lastly, to isolate the effect of merging partitions that are often queried together, we run Space Odyssey with and without merging enabled. In this experiment, clustered queries are produced using 5 instead of 10 *clustercenters* in order to ensure that the queries can benefit from merging. In Figure 5c, we plot the times only for the queries that request the most popular combination (for the zipf distribution, this combination is queried 751 times). While the same combination may still request completely different ranges (e.g., in different clusters), we see that eventually Space Odyssey benefits from the merged partitions for the majority of the queries. We observe 25% performance gain on average for the queries accessing the merged partitions.

5. CONCLUSIONS

In this paper we identify the challenge of efficiently exploring multiple spatial datasets with the same range query—a common type of analysis across scientific applications. State-of-the-art methods fall short in supporting this challenge efficiently as they require to index *all* data a priori including the parts never analysed.

As a consequence we develop Space Odyssey, an approach which incrementally indexes the bits of the data needed and that adapts the physical layout of the data on disk to efficiently support the queries executed. Our novel approach to incrementally indexing and reorganizing spatial data on disk shows benefits in decreasing the data-to-insight time.

Although the current implementation of Space Odyssey already achieves a speedup, we primarily consider it a starting point demonstrating the potential of our idea. In particular, we believe that refining the cost model for merging and indexing can further increase performance benefits.

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 650003.

6. REFERENCES

[1] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD '12*.

[2] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic Contention Detection and Amelioration for Data-intensive Operations. In *SIGMOD '10*.

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

[4] G. Graefe and H. Kuno. Adaptive Indexing for Relational Keys. In *ICDEW '10*.

[5] G. Graefe and H. Kuno. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *EDBT '10*.

[6] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-record Synthetic Databases. In *SIGMOD '94*.

[7] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR '07*.

[8] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. In *VLDB '11*.

[9] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. In *VLDB '14*.

[10] S. T. Leutenegger, M. Lopez, et al. STR: A Simple and Efficient Algorithm for R-tree Packing. In *ICDE '97*.

[11] H. Markram et al. Introducing the Human Brain Project. *Procedia Computer Science*, 7:39–42, 2011.

[12] D. Šidlauskas, C. S. Jensen, and S. Šaltenis. A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-intensive Workloads. In *DaMoN '12*.

[13] E. Stefanakis, Y. Theodoridis, T. Sellis, and Y.-C. Lee. Point Representation of Spatial Objects and Query Window Extension: A new Technique for Spatial Access Methods. *IJGIS*, 11(6), 1997.

[14] Y. Tao and D. Papadias. Adaptive Index Structures. In *VLDB '02*.

[15] F. Tauheed, L. Biveinis, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. Accelerating Range Queries For Brain Simulations. In *ICDE '12*.