Optimizing Geometric Pattern Matching Utilizing Caching of Decomposed Queries in Partitioned Datasets

Martin Poppinga[1,2]

[1]Universität Hamburg, Fachbereich Informatik, 22527 Hamburg, Germany
[2]Universität Hamburg, ZBH – Center for Bioinformatics, 20146 Hamburg, Germany

Abstract
In research, searching for patterns within large datasets is a common task but often results in long-running queries. Various approaches exist to speed up individual searches, for example, indexing, denormalization, or caching. These approaches come with their advantages, but also limitations. For caching, the data is usually either cached on a low level, for example, to buffer data structures in memory to increase the read access performance, or the complete results are stored for specific queries so that if the same query is observed again, the system can serve the cached result instead of recomputing the results in the database management system. In this work, we propose an architecture that can increase read performance by utilizing a caching approach that acts as an index by storing references to distinct partitions based on partial queries. This allows memory-efficient caching while gaining the ability to improve not only already-seen queries but also queries that have not been computed before. This approach is designed for workloads often observed in scientific domains, targeting analytical queries that search for patterns in datasets, for example, in bioinformatics, spatial data, or time series.

Keywords
Caching, Index Structures, Partitioning, Pattern Matching, Query Decomposition, RDBMS, SQL

1. Introduction

In the scientific domain, one of the often occurring tasks is to search for defined patterns, for example, searching for patterns in protein structures in bioinformatics [1] or patterns in spatial datasets [2]. While simple properties are fast to find in datasets using index structures, other properties are more complex or require computational effort if several data points are put in relation to each other. If the pattern contains, for example, several spatial points with distance constraints, the distances between many possible points need to be calculated for each search. This can be even more challenging if distances are not Euclidean but, for example, require a shortest-path routing. Complete denormalization is often impractical since the selected points can vary depending on the query and use case. Although relations between points can be denormalized in a small dataset, for example, in a property graph, there would be too many possible combinations in large datasets. If a changing set of rows, depending on specified attributes, is put into relation to each other, the data is often stored in a relational database management system (RDBMS).

As usually only points within the same area need to be compared, many problems can be divided into individual search problems if the dataset has multiple disjunct partitions. Examples of natural partitions in datasets are individual days in temporal data, regions in geospatial data, or experiments in scientific data. To obtain all matches where the given properties are fulfilled, usually all partitions need to be searched. Putting different data points in relation to each other requires joining them. Optimizing the join order or using search trees speeds up processing. If the attribute conditions are not specific, a high number of join candidates need to be considered, and indexes may be of limited use. Optimizations do not remove the necessity of performing time-consuming computations in many cases. This is especially troublesome if these queries have an increased runtime due to complex conditions and a large dataset spanning across many partitions.

Even if analytical queries often have conditions specified that are used frequently among several searches, their results are often recomputed for each new search. To reduce the number of required condition checks, we aim to store already computed conditions of partial queries.

We propose utilizing an architecture that enriches a cache with already-seen queries and their derivatives. To gain the derivatives, we utilize query decomposition to find partial queries. This cache acts as an index structure to map queries to partitions to reduce the search space for a given query. The cache stores references to all partitions with at least one result for the specific cached query, reducing the number of partitions that must be searched if the system encounters a query found in the cache. This allows for more flexible utilization, as improvements are not restricted to a specific kind of query, as in many denormalization approaches. It also allows

35th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), May 22-24, 2024, Herdecke, Germany.
martin.poppinga@uni-hamburg.de (M. Poppinga)
0000-0001-8529-8376 (M. Poppinga)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3710/paper12.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

for combining multiple cache hits for one query, unlike a common result cache, allowing it to work for new queries. Utilizing already-seen query parts can be combined with a warmed-up cache approach, where we pre-populate the cache with expected conditions, like an often-used timespan, a specific set of regions or experiments, or predefined fingerprints.

2. Background

2.1. Definitions

A *partition* consists of a set of data points where the data can be put in relation to each other. These can be, for example, spatial areas, time frames, or individual experiments. This work focuses on disjunct natural partitions defined by some partition key per data point. If overlapping partitions are required, data points can be duplicated to all relevant partitions to ensure that matches are found within one partition.

A *query* is a defined search in a dataset. For this work, we focus on analytical SQL queries that search for groups of data points related to each other. They specify conditions that describe a pattern searched in the data to mine for matching occurrences. They match within a single partition and often place data points in relation to each other by some property, such as a distance. See also Section 3.1.

A *partial query* is a query that uses only a subset of predicates of another query by removing one or multiple conditions or dimensions involved. One query can have multiple partial queries as derivates. Such partial queries are created by decomposing queries[3].

2.2. Related Work

Database Systems Depending on the workload, different storage models in database systems have advantages over others. Designated solutions target scientific data, such as SciDB[4] or DuckDB[5]. As column-based databases generally perform well for aggregations [6], they are often used for analytical queries. Systems like DuckDB target OLAP (Online Analytical Processing) queries, while most relational systems focus on OLTP (Online Transaction Processing). While column-based systems are beneficial for most analytical queries, as aggregations on individual rows are of interest, queries that need to be executed on various subsets of the data and use changing attributes for selections often rely on row-based systems. Many approaches to improve query speed include data denormalization [7]. If relationships are of interest, graph databases are often utilized. However, on large datasets, if each point has a potential (spatial) relationship to all other points in the same partition, the number may become too large to be represented as a graph or otherwise denormalized.

Relational database management systems (RDBMS) are generally utilized to store normalized data. Here, each data point is stored in a row in a table and joined with other data points depending on the query, computing relations, and checking conditions if needed. RDBMS can handle large amounts of data and, in many cases, achieve very good performance metrics. Depending on the data and queries, different index structures are used to increase read performance, common structures are B-trees [8], r-trees [9] for spatial data, or bitmap indexes [10] for range conditions. Index structures allow individual rows or sets of rows to be found efficiently, but can lose effectiveness in cases where rows are put in relation to each other and single conditions are not very selective (see Section 3.1).

Searching Patterns This work aims mainly to improve query runtime to find geometric patterns in protein data [1]. However, it is also targeted to resemble a useful approach for searching for spatial properties in geospatial datasets [2] and other domains that face similar workloads. In previous work [11], we showed that for our scientific workload in GeoMine [1], the database management system achieved better performance if given flexibility to reorder and optimize query execution plans, in contrast to customized algorithmic approaches. Although systems can optimize query execution plans, the query and index design must also be sufficient to allow such optimizations. However, with increasing dataset size, these classical approaches came to their limits if the whole dataset needed to be searched.

Caching Different techniques are utilized to improve query response times in databases. One way to reduce the computational load is by reusing already computed results by caching parts of the data. RDBMS often buffer pages in memory to prevent slow access to secondary storage, for example, often accessed indexes or tables. As relational databases focus on data consistency, serving outdated results would be a big issue. Cache invalidation, deciding if a cache no longer holds a valid result set, is a big problem and can cause computational overhead.

It is also possible to cache the complete results related to a query, allowing for a fast return of previously seen queries. This reduces the load on databases that frequently serve the same queries, for example, on a website. This can be directly integrated into the database or added as a separate caching layer in the application logic or by using some middleware. Often, such approaches utilize designated in-memory key-value stores for caching [12].However, these approaches only serve the results to exactly these queries that have been computed before,

and if large result sets are returned, memory consumption is also of concern. Other approaches, for example, for SBQL (Stack-based Query Language), use the object-oriented model to decompose a query and cache specific subqueries based on the object tree [13].

Materialized Views On many systems, materialized views [14] do exist, which serve similar purposes as result caching. They allow precomputation of SQL expressions and transformations, reducing the need to compute these parts for each query execution. These views can be utilized directly in queries by specifying the view in contrast to the base table. Furthermore, there are approaches to redirect queries from the base table directly to the materialized view by rewriting the queries [14].

Keeping the views valid is usually transparent to the user as the RDBMS handles the updates if entries are added or updated. However, as with other denormalization techniques, they come with increased storage consumption and reduced write performance. In addition, precomputations are usually only beneficial for specific queries; if varying queries are expected, multiple materialized views are required, often with individual index structures, which further increase storage consumption.

Filter Individual solutions exist for applications where indexes to fingerprinted structures are specified. Such descriptors describe the areas in which these predefined structures are present. For example, in [15], spatial triangle structures are defined in a fingerprinting approach. The triangle descriptor, a bit vector, represents whether the given triangle is in a partition. This allows the extraction of all specified triangles (or, more generally, defined patterns or fingerprints) from a query and compares all bit filters to reduce the number of partitions to search in the following steps.

There are two ways to specify which structures are present in which partitions. An inverted index describes which partitions each descriptor is present in, for example, by providing a list of partition keys or using a bit vector in which the partitions are encoded. Alternatively, a forward index can be used where it is for each partition specified which descriptors are present. This can be stored as a list again, or if only estimated containment is needed by structures like bloom filters, which reduces the memory footprint. Both approaches have advantages and drawbacks depending on the individual workload.

3. Basic Concept

3.1. Problem Description

Our work aims at scientific workloads with no frequent updates, where we want to improve query response times. The described environment is related to the workload described in GeoMine [1], but it is also expected to be similar in other scientific domains. It is designed for a user-defined search through a research application where the user defines properties that must be matched in the data. Such flexible data mining tasks are often exploratory, running similar queries until the desired pattern is found.

The conditions bring various data points in relation to each other, for example, by defining the distances between various points selected by their attributes. This creates queries that join several large tables in a single query [11], as, for example, an SQL query processed by an RDBMS. Such a query looks for all occurrences of the specified properties within several areas. The different areas can be viewed as natural partitions specified by partition keys. In our example (see Listing 1), these are *cities*, reducing required table joins to those within a city. In other workloads, such partitions could be individual experiments or data sources with no relevant connections between two partitions. Data points are selected by their attributes, such as the type or name of a point of interest (POI), and placed in relation to each other, for example, by a maximum distance. One query consists of multiple references to one or more tables that are related to each other, if no special conditions between individual data points are specified, the result consists of all combinations of all matching points within a partition. Further, partitions can be part of other conditions like a subquery restricting the number of partitions.

Listing 1: Example for a spatial query, defining conditions between several points

```
SELECT p1.name
FROM poi as p1, poi as p2, poi as p3
WHERE p1.type = 'school'
AND p2.name ILIKE '%museum%'
AND p3.type = 'soccer_field'
AND dist(p1.geom, p2.geom) < 200
AND dist(p1.geom, p3.geom) < 500
AND p1.city = p2.city
AND p1.city = p3.city
AND p1.city in (SELECT city FROM
    cities WHERE population < 10000)
```

As only spatial relations within a partition need to be computed, this reduces the number of required comparisons; still, many combinations of data points must be checked. As the number of relevant points can often be reduced by filtering for the point's attributes, the usage of spatial indexes in the presence of a partition key is of limited use if the partitions are small. In our example, we could use a spatial index to select all points that are within the defined distance. However, since attributes need to be checked for each point, in many cases, it is faster to instead filter for the attributes and compute the

distances to all matching points within the partition. As each query may filter for different attributes, multiple indexes would be required, creating even more overhead maintaining combined spatial indexes. Furthermore, because of the high number of theoretical combinations, the distances can usually not be easily precomputed, which limits the usage of range-based indexes.

In cases where we have a highly selective point, we can directly limit the search to these partitions. However, individual data points, even if filtered by multiple attributes, are often not very specific, as similar points are present in many, if not most, partitions. For example, most cities will have a school, many a school with a nearby soccer field, but only a few also have a nearby museum, while each type of POI may be present in most cities. If the search contains, for example, conditions with text searches with a prefixed wildcard, a high number of points may need to be evaluated, even if a point has only a few matches, due to limited indexing possibilities. Although each point itself is not very selective, the combination of several points, together with distance constraints, reduces the number of results.

Depending on the environment, a dataset can consist of thousands to millions of partitions and data points can have low cardinality, so that single conditions are very unspecific. As matches can theoretically occur in all partitions, the entire dataset with many points must be considered for each query. One way to reduce the search space is to utilize caching.

3.2. Basic Approach

Our approach does not aim to replace the RDBMS; it tries to help the system search more efficiently by reducing the search space. If only a limited number of partitions need to be considered, less data must be read from the secondary storage, and the computational effort of distance checks needs to be performed in fewer partitions. In contrast to other caching approaches, we focus on the partition keys instead of the actual results; this has several benefits, as will be discussed later. We must know which partitions the query may find valid results, to reduce the search space. As we search for conjunctive conditions, we can determine that partitions in which at least one query condition is not fulfilled cannot contain a valid result. The same is true for any subset of conditions or tables involved.

Using this property, we can check for each (sub)set of conditions if we have information stored; if we have information, we can restrict the search to the stored partitions, as shown in Figure 1. A, B and C are sets of partition keys based on conditions a, b and c. These conditions can be selections on attributes of a single point, as well as multiple points that are put in relation to each other via a distance or similar. If we have stored the set of

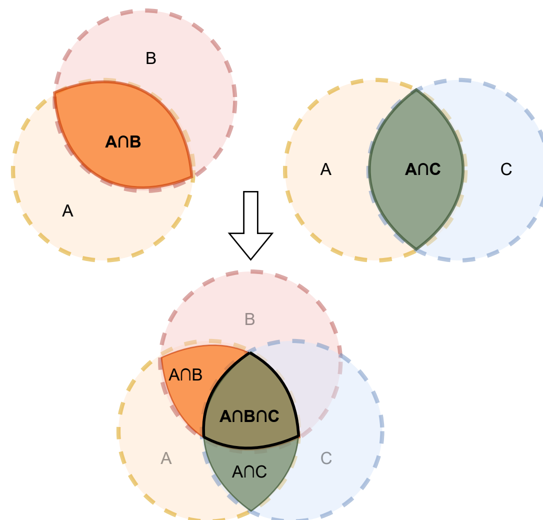

Figure 1: Sets of partitions of different conditions. Each set consists of partition keys based on the corresponding condition or set of conditions. If intersecting sets of partitions, the search space can reduced for queries. If different subsets are known, they also can be intersected.

partition keys in which conditions $a \wedge b$ are fulfilled and the set of partition keys in which $a \wedge c$ is fulfilled from previous queries, we can restrict the search space for a new query $a \wedge b \wedge c$ to a smaller number of partitions in $A \cap B \cap C$.

This way, we can restrict the search to the intersection of all known conditions if we have information on multiple subsets of conditions. In the best case, combining several conditions can restrict the search space to only those partitions where we have valid results. This can be, for example, the case if seeing a very similar query or having encountered multiple queries that contain each one part of the new query. This approach works best if a restrictive part of the query has been cached, so new queries which added or changed other constraints benefit, but also combining several less restrictive sets of partition keys can lead to a overall reduced number of partitions to search.

3.3. Populating the Cache

The cache will be populated by analyzing queries observed on the database system. The system will monitor the database logs and obtain all queries that are run against the database. It will rerun the queries seen in the database and store the resulting partition keys. Furthermore, the system will identify partial queries, which are queries that are based on the original search but omit one or more of the joined tables. These partial queries

consist of all possible combinations that are conjunctively connected. For this, the conjunctive normal form is created. With this approach, it is more resilient to changes in queries, as we have valid results in the cache if a part of the original query is changed and searched again. The system can be restricted to limit the number of derivations of the query to prevent too many unspecific partial queries from being created from a decomposed query. To ensure better cache utilization, we can additionally generate more generalized queries, for example, by rounding numeral conditions like distances to integers to prevent cache misses if a distance is slightly changed. However, defining which generalization steps are helpful may depend on the dataset. In a final step, the query is rearranged and normalized so that a unique hash is generated for each query, regardless of different named aliases or a different ordering of conditions. We utilize a concurrently running system to avoid a negative impact on the user experience, which would occur if we materialize intermediate results, as they usually existed only in memory or were never computed due to optimization techniques of the database. The cache now contains hashes of (partial)queries with all partition keys where results for this query can be found.

3.4. Utilizing the Cache

If a query is run, it is checked against all known entries in the cache. For this, the query is, like while cache population, normalized and decomposed into all possible partial queries. In addition, more variants can be created than were created while populating the cache. Additional query variants, for example, with a wider range condition, can be utilized as long as we can guarantee query containment so that no results are lost. From all partial queries found in the caches, the sets of stored partition keys can be intersected, resulting in a set of partition keys that can then be appended to the SQL query, which is executed on the RDBMS (see Figure 2). The partition keys may contain false positives, but the result remains valid as the RDBMS still verifies all conditions.

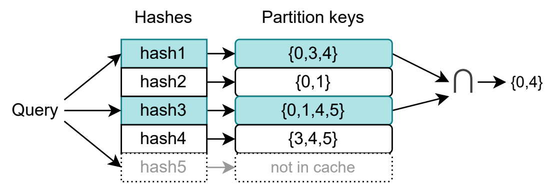

Figure 2: A query is decomposed in partial queries, each represented by a hash; In this example hash1, hash3 and hash5, of whom hash1 and hash3 are already present in the cache. Hash5 was not found in the cache, as this partial query was not seen before and is ignored. The partition keys of the matched hashes are intersected and can then be used to restrict the search space to partitions 0 and 4.

4. Considerations

Storage Usage If a partial query has a large set of results, it needs more storage to store partition keys, and, at the same time, it has less benefit in the overall approach. It is possible to store only queries that yield fewer than a specified number of partitions to reduce the required storage space. To prevent the entry from being calculated repeatedly, a separate table can hold the number of partitions resulting from each query, indicating that the query has already been computed. To focus on more specific queries, it can be beneficial to only cache query variants with table combinations that have a specific relation to each other, like a distance metric, as otherwise too many results could be found, increasing computation time and the number of partitions. Furthermore, to reduce memory consumption, it could be possible to group partitions and reference these clusters; this can be beneficial if the data has many similarities between different partitions. In a similar approach, queries with similar result sets can be grouped. Here, the hashing function must be adapted and the results for all combined hashes must be united. If there are a high number of unspecific conditions, it is also possible to maintain a negative cache, containing per query all partitions where no matches are found; all such matches can then be united and used in the SQL query or cut with the result for the positive cache. How effective it is to store the partition keys instead of the actual results in terms of storage consumption depends on the size of the partitions and the number of results per partition. Here, further analyses are necessary for specific workloads.

As we focus on read-only and append-only datasets, cache invalidation is currently not in our focus. For scientific datasets, many projects work with dataset dumps with no updates or only periodic ones. To prevent indefinite growth of the cache, eviction strategies, such as *least recently used* or removing the largest (least specific) cache entries, can be used.

Dataset Updates Datasets can be updated as long as partitions are immutable. Deleting partitions would increase the false positive rate, but not create false negatives. Append-only datasets can be handled by maintaining a list of the newest partition available when adding a query to the cache. If we hit a cache entry created in an old database state, we can identify the oldest state and add all partitions to our result set that were added later. To prevent a growing number of false positives, cached queries can be rerun to update the set of partitions.

If data within an existing partition is changed, this par-

tition either needs to be considered for each subsequent search regardless of its presence in a cache entry, or all cache entries must be revalidated for this partition.

Prepopulation It is also possible to prepopulate the cache with expected queries, for example, by altering the seen queries by changing individual attributes. Also, often searched fingerprint-like patterns can be precomputed, and structures specific enough to reduce the number of partitions can be integrated, integrating approaches described in Related Work.

Effectiveness This approach relies on recurring patterns in queries and disjunct partitions. This approach creates some overhead, so the effectiveness must be high enough to reduce the overall runtime. For this, the frequency of the cache hit and the number of false positive quotes are most relevant. In addition, it may not be helpful for a search that requires less than a few seconds.

In addition to complex partial queries, it may also be beneficial to store simple queries. While these queries can usulay be quickly resolved by a relational system, it can also be beneficial to reduce the number of possible partitions early within a cache as it reduces computational effort in intersecting and rewriting the query.

5. Technical Realization

Usage Although this approach is designed to work with standard SQL, it should also directly work with common extensions, such as the spatial PostGIS[1] extension for PostgreSQL. The approach itself is also not limited to a defined language and can be utilized in various environments. Furthermore, this approach can still be combined with alternative techniques, such as a fan-out in distributed setups or specilized index structures.

Data Storage Different solutions are possible to store the cache. Each cache entry could be realized as a separate materialized view, while query rewriting would allow each existing matching view to be joined to restrict the search space. Similarly, large tables can be utilized to reference partition keys or key-value approaches within the RDBMS. These approaches allow all data to be stored within a single location, and the RDBMS to have maximum freedom for optimization steps. Alternatively, an external key-value store can be used. To integrate this, either a query engine like *PrestoDB*[2] can be utilized, or the RDBMS can provide such integration. Furthermore, query rewriting could happen at the application level, where the cache is based on an embedded store or a separate system.

[1]https://www.postgis.net/
[2]https://prestodb.io/

Data Structure Storing the cache in an inverted index or some related data structure helps fast retrieval of the set of partition keys for a given query. Adding a new cache entry is also simple, as just a new key value pair needs to be added, even if this may cause some balancing of the data structure. Also, it would be possible to store not a set of partition keys but utilizing a bit vector, describing in which partitions the query is matching. Fast intersections are possible here, but storage consumption is higher in contrast to sets with only a few members. Alternatively, a forward index can be used to store the queries that match for each partition. This can reduce storage consumption, but increases search time as all indexes need to be checked. However, using probabilistic data structures, such as a bloom filter or a hierarchical bloom filter, can reduce storage consumption and search time.

6. Conclusion

Although it does not eliminate the need for database computations, our approach aims to reduce the search space within the database and, therefore, the runtime. As we decided to store only the set of partitions and not the actual results, it consumes less memory and creates less overhead to intersect and serialize the query. In addition, cache hits of different partial queries can be combined. It allows the utilization of multicolumn indexes based on partition keys, reducing access time when verifying conditions. Since sets of partition keys of partial queries can be intersected, it is more resistant to changes in queries than other approaches. This allows it to provide cached results even if new queries are run that combine elements of already existing queries. In contrast to approaches that rely solely on precalculated structures, this approach is more flexible in terms of relevant structures expressed by the queries, which allows for a better reduction of the search space.

Open questions remain for future work as the choice of technology is still open for discussion. Although the general approach works with different technologies, depending on the specific workloads, the utilization of inverted and forward indexes, as well as the usage of sets and descriptors, must be evaluated. Different approaches may be combined to utilize positive and negative sets or probabilistic structures, such as a bloom filter. The options where to store the cache, range from an embedded system within the application, a separate key-value store, to storing it within the same database as the original data. Additionally, a middleware or separate processing layer could be used to integrate our approach with existing systems and approaches.

References

[1] J. Graef, C. Ehrt, K. Diedrich, M. Poppinga, N. Ritter, M. Rarey, Searching geometric patterns in protein binding sites and their application to data mining in protein kinase structures, Journal of Medicinal Chemistry 65 (2022) 1384–1395. doi:10.1021/acs.jmedchem.1c01046.

[2] Y. Fang, R. Cheng, G. Cong, N. Mamoulis, Y. Li, On spatial pattern matching, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE), 2018, pp. 293–304. doi:10.1109/ICDE.2018.00035.

[3] E. Wong, K. Youssefi, Decomposition—a strategy for query processing, ACM Trans. Database Syst. 1 (1976) 223–241. doi:10.1145/320473.320479.

[4] M. Stonebraker, P. Brown, D. Zhang, J. Becla, Scidb: A database management system for applications with complex analytics, Computing in Science & Engineering 15 (2013).

[5] M. Raasveldt, H. Mühleisen, Duckdb: an embeddable analytical database, in: Proceedings of the 2019 International Conference on Management of Data, 2019.

[6] G. P. Copeland, S. N. Khoshafian, A decomposition storage model, SIGMOD Rec. 14 (1985) 268–279. doi:10.1145/971699.318923.

[7] G. Sanders, S. Shin, Denormalization effects on performance of rdbms, in: Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 2001, pp. 9 pp.–. doi:10.1109/HICSS.2001.926306.

[8] D. Comer, Ubiquitous b-tree, ACM Computing Surveys (CSUR) 11 (1979) 121–137.

[9] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD international conference on Management of data, 1984, pp. 47–57.

[10] C.-Y. Chan, Y. E. Ioannidis, Bitmap index design and evaluation, in: Proceedings of the 1998 ACM SIGMOD international conference on Management of data, 1998, pp. 355–366.

[11] M. Poppinga, J. Graef, K. Diedrich, M. Rarey, N. Ritter, Database and workflow optimizations for spatial-geometric queries in geomine, in: Lernen, Wissen, Daten, Analysen (LWDA) Conference Proceedings, 2023, pp. 86–97.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny, Workload analysis of a large-scale key-value store, in: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, 2012, pp. 53–64.

[13] P. Cybula, K. Subieta, Decomposition of sbql queries for optimal result caching, in: 2011 Federated Conference on Computer Science and Information Systems (FedCSIS), 2011, pp. 841–848.

[14] R. Chirkova, J. Yang, Materialized views, Foundations and Trends® in Databases 4 (2012) 295–405. doi:10.1561/1900000020.

[15] T. Inhester, S. Bietz, M. Hilbig, R. Schmidt, M. Rarey, Index-based searching of interaction patterns in large collections of protein–ligand interfaces, Journal of Chemical Information and Modeling 57 (2017) 148–158. doi:10.1021/acs.jcim.6b00561.