Caching Partition Identifiers for Fast Geometric Pattern Matching

Martin Poppinga [1], Annett Ungethüm[1], and Matthias Rarey [2]

Abstract: Pattern matching in large datasets is fundamental across scientific domains, as in bioinformatics or geographic information systems. As datasets grow and search complexity increases, these operations often result in long-running queries. While various approaches exist to improve query execution times, they face limitations when dealing with complex geometric patterns in comprehensive datasets. This paper presents a novel caching-based architecture combined with query decomposition to significantly reduce computation time for geometric pattern searches in large partitioned datasets. We focus on GeoMine, a software tool for searching protein-ligand complexes in life sciences, but demonstrate the approach's broader applicability. Our key innovation lies in recomposing queries into smaller subvariants, which are separately executed. The results are stored as an index-like structure, allowing subsequent new queries to utilize one or multiple entries of this cache to reduce the search space to relevant partitions. We evaluated our approach on a lightweight variant of GeoMine and tested various storage backends, including RocksDB, Redis, and PostgreSQL. With this approach, we could substantially reduce the computation time for our use case. We further applied our approach to geospatial data using an OpenStreetMap dataset to demonstrate transferability and show its potential beyond its original bioinformatics use case. Our work contributes to more efficient data exploration in scientific research, accelerating queries on datasets often occurring in the scientific domain.

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

1 Introduction

A common and crucial task in research is finding patterns of interest in large datasets. These patterns often involve complex geometric relationships between multiple data points, such as in the analysis of protein structures in bioinformatics [Ku16] or spatial analyses in geographic information systems [Ch22]. Large search spaces, consisting of millions or billions of points, together with individual and changing search patterns, which put an arbitrary number of these data points in relation to each other, make this a computational challenge that requires efficient approaches. Although some searches are expected to run experiment-wise for a long time, other workflows are based on an iterative or interactive approach [Di24]. In these approaches, an existing search result is reviewed, and the search is repeated with changed parameters, allowing explorative pattern searches in these datasets.

[1] University of Hamburg, Department of Informatics, 22527 Hamburg, Germany, martin.poppinga@uni-hamburg.de, https://orcid.org/0000-0001-8529-8376; annett.ungethuem@uni-hamburg.de

[2] University of Hamburg, ZBH – Center for Bioinformatics, 22761 Hamburg, Germany, matthias.rarey@uni-hamburg.de, https://orcid.org/0000-0002-9553-6531

doi:10.18420/BTW2025-23

In this work, we aim to improve the query execution time of such tasks. We applied it to *GeoMine* [Gr22], an application for searching geometric patterns in large protein-ligand complexes, enabling explorative search in life science. *GeoMine* relies on a relational database system to store the protein structures and to execute the queries. It is used for public research, as it can be accessed for free, utilizing public datasets, via the *proteins.plus* [Sc22] web server, and is also used by pharmaceutical companies internally.

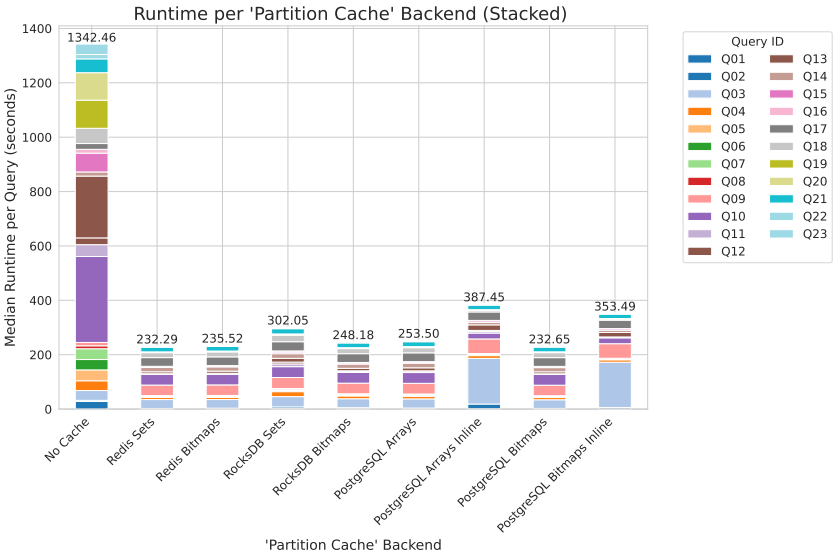


Fig. 1: Median runtime per query from *GeoMine* test queries. 'No Cache' shows the runtime without our approach. The other values show the runtime with our approach, using a fully populated cache with different storage backends.

Pattern matching, as required for *GeoMine*, is often used in the context of spatial data and relates to the detection of subgraph isomorphisms. It is a heavily studied type of query that has been optimized on different levels of abstraction. At the system level, the focus is on the distribution of workload in scale-up or scale-out systems, the reduction of messages between partitions, or the elimination of synchronization barriers [Fa13, MWM15, Kr17, Kr19]. Generally, bulk-synchronous processing does not work well for pattern matching [Fa13]. Instead, the performance benefits from a more asynchronous processing model, which can be adapted and refined in different ways [Fa13, Kr17]. On the algorithmic side, performance can be optimized by various techniques, e.g. pruning strategies [Fa18], cardinality estimation of feasible matches [HS08], or graph indexing [ZLY09]. However, most of these approaches are designed specifically for graph databases or spatial databases. Moreover, an experimental analysis of different subgraph isomorphism algorithms has shown that the best-performing algorithm can differ depending on the exact use-case [KMS16]. Hence, we present an approach that does not restrict the choice of the underlying algorithm and is independent of the processing model used.

Our approach shows a novel caching-based architecture that leverages query decomposition and recomposition to accelerate geometric pattern matching in large partitioned datasets significantly. It is based on storing and reusing results of partial queries, allowing subsequent queries to benefit from cached results. We utilize a concurrently running workflow that executes recomposed query variants and stores the result in an index-like structure for subsequent new queries to use for search space reduction. This way, queries sharing common search elements benefit from each other. This leads to significant performance gains compared to using only existing caching strategies, especially if the system has not seen the query before. Our architecture supports flexible integration of storage backends, which we evaluated using RocksDB, Redis, and PostgreSQL. Fig. 1 shows the speed-up of our approach for several test queries[3] under optimal conditions.
In this paper, we specifically target the properties we found in our workloads in *GeoMine*, namely a partitioned dataset and query components repeating across multiple queries. The dramatically reduced computation time for complex search queries accelerates the overall workflow for research in the applicable fields.

This work is structured as follows:
First, in Section 2, we provide an overview of our field of work, i.e., the tasks *GeoMine* solves and the specific challenge we are targeting. In Section 3, we describe our methods and provide insight into our overall approach. In Section 4, we describe the experimental workflow and discuss the results. In Section 5, we discuss related work. In Section 6, we identify the implications and limitations of our work and give an overview of future work.

2 Background

Searching for patterns in large datasets is fundamental in many scientific domains. In bioinformatics, searching for geometric spatial patterns is an essential task in early drug design research. Similarly, researchers look for patterns of co-located landmarks or other points of interest in geographical information systems [Ch22, Fa18].
For the design of medical drugs, the workflow consists of many steps [Ku16]. For instance, early exploration begins with discovering specific structures in protein datasets. The most comprehensive data source of experimentally derived three-dimensional structures of proteins is the Protein Data Bank (PDB) [Be00]. This dataset is freely available and widely used by researchers and research tools. The PDB serves as input for tools such as *GeoMine* [Gr22], CSD-CrossMiner [Ko16] or Relibase [He03], which are developed to search for structures and patterns, which are then investigated in more detail in subsequent research steps. Such geometric properties and patterns are of special interest as these define the functionality of proteins, which is crucial for understanding the interaction between molecules and the various processes in organisms.

[3] Different examples from the drug design domain, searching for various patterns across a large dataset

2.1 GeoMine

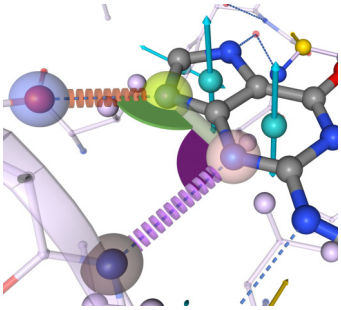
GeoMine [Gr22] is a web-based tool, freely available on the *proteins.plus* [Sc22] webserver[4]. It allows the search for specific geometric properties, like distances between atoms or angles between atom pairs and other non-geometric properties within all the protein structures in the dataset. *GeoMine* enriches the dataset with additional data, such as the computed surface areas or interaction bonds between atoms. All atoms are assigned to binding site pockets. While there is no standard definition of what a pocket is [CA24], it always describes areas of interest for researchers. It is the part of the protein molecule where interactions with other molecules occur. For example, smaller molecules, the so-called ligands, interact in pockets with the larger proteins, influencing the protein's functions and potentially creating medical effects. Each of the more than 200,000 structures of the PDB used by *GeoMine* contains one to several pockets that are computed using the DoGSite3 [GER23] algorithm. Each pocket can be searched separately for pattern matching as matches are found within individual pockets. Therefore, the dataset can be viewed as disjunctly partitioned into more than 2,000,000 of these binding pockets, which contain all surrounding atoms.

User Interface (UI) The UI allows researchers to design the geometric queries in a three-dimensional space in a web-UI [Di24]. For this, the user manually specifies the searched properties or uses a structure as a template in a three-dimensional view (see Fig. 2a) or two-dimensional abstraction view. The user may specify an arbitrary number of points, such as atoms or centers of aromatic rings, and define the distances or angles between them. In addition, various properties can be filtered for each point, such as its chemical element or the calculated exposure to the protein's surface. The so-built query is then executed on the whole dataset or, if desired, a subset of the data. The user is then presented with all matches of the structure for the given search, which can then be refined by adapting the query.

Queries The queries in *GeoMine* are built as follows: Each search consists of specifying several points from a *pointTable* (see Fig. 2b). For each point, a varied set of attributes, such as the chemical element or its surface exposure, can be specified for selection. The cardinality of each attribute is quite low, as, for example, many points are of the element *carbon*. In addition to the attributes, functional relationships between two or more points are specified. This can be a geometric property, like a distance, which is directly calculated using the coordinates of the points, a user-defined function to calculate angles or an additional table that stores precalculated interactions between two points. Further, subqueries can restrict the search space as sometimes only a subset of pockets is relevant. These subqueries restrict the number of pockets based on attributes stored per pocket or protein structure.

Backend Each *GeoMine* query is translated into an SQL query and executed on a PostgreSQL system. In an earlier work [Po23], we applied various classical optimization techniques, allowing the underlying database management system to run the query efficiently in contrast to an earlier algorithmic-centered approach. Execution times range from a few seconds to

[4] https://proteins.plus



(a) Example view from the NGL-based UI, using a protein structure 1H1S to design a query.

```
SELECT p2.pocket_key AS pk,
       p1.id AS id1, p2.id AS id2, p3.id AS id3
FROM points AS p1, points AS p2, points AS p3
WHERE dist(p1.x, p1.y, p1.z, p2.x, p2.y, p2.z)
    BETWEEN 16.0 AND  36.0
  AND (p1.origin = 2 OR p1.origin = 1)
  AND (p2.origin = 2 OR p2.origin = 1)
  AND p2.element = 7 AND p2.surface >= -1
  AND p3.element = 8 AND p3.surface >= 0.5
  AND p2.pocket_key = p1.pocket_key
  AND p2.pocket_key = p3.pocket_key
  AND p2.pocket_key IN
      (SELECT pocket_key FROM po_with_ligands)
```

(b) Truncated query from *GeoMine*, joining all points within a pocket, filtering for atoms attributes, and using a UDF to search for a specified distance between two points and selecting a subset of the dataset.

Fig. 2: Example of GeoMine UI and SQL query structure

several minutes or hours, based on the complexity of the query and the number of matched structures. Queries with fewer predicates that filter primarily on the geometric distances require high computation time due to the limited possibilities of query optimization and the overall high amount of necessary computations.

2.2 Key Properties and Challenges

This work aims to improve query execution times for the tasks observed in *GeoMine*, which are pattern-matching queries on geometrical structures. Searching patterns involves the self-joining of the table and putting several data points in relation to each other, e.g., distances. We extracted the properties of our application, which define the scope of our work. They can be divided into two categories: a) Properties describing the complexity of the targeted challenge and b) Properties describing the limitations of the application.

Complexity

- *Variability of queries* Unlike simple key lookups, the queries targeted here can vary widely. For each point, a different set of attributes could be specified. Further, the number of points specified within a query ranges from 4 to 15 in most observed workloads, with more involved points in some scenarios.
- *Data volume* The current dataset contains more than 300,000,000 points. Although still manageable by relational database systems, the size has the potential to multiply in the future with the emergence of new data sources like Alphafold [Ju21], which use generative AI to generate datasets. Together with the characteristics of the relationships, the theoretical search space is enormous. At this point, a complete denormalization of all distances or functions is not feasible.

- *Individual relationships* As the individual relationships between points are of interest, such properties must be calculated between all candidates. For example, the Euclidian distance in three-dimensional space, although not per se complicated, requires computational effort. Due to the number of possible candidates and the lack of practicability to precompute all distances, these computations impact overall performance notably. For other functions, e. g., the computation of angles, which include more than two points, the number of possible combinations is even larger.

Limitations

- *Read-only* As is often the case in scientific datasets, the dataset is imported once and then updated occasionally, but it usually does not get changed while processing.
- *Natural Partitions* For a search, only the relationships between points within a disjunct partition are of interest. For example, it is not necessary or desired to check the distances to all points in the dataset; only the distances in each individual binding pocket are of interest.
- *Recurring query patterns* In the queries, we see partially recurring elements. As explorative pattern search is often an iterative workflow, equal or similar query parts are often present in subsequent searches if some parameters change or points are added or removed. Further, searching for similar targets may involve similar structures across different users.

For our approach, which we will present in more detail in Section 3, we aim to utilize the defined *complexities* and *limitations*. Our approach aims to make caching robust enough to support changing queries and allow it to be combined with denormalization steps.
For better transferability and integration in our existing system landscape, we decided to design it as a loosely coupled application that consumes and enriches SQL queries. This way, our approach must not rely on system-specific features that could be deprecated in a later version or require changing the database engine to stay maintainable. A reference implementation[5] can be utilized for integrating our approach in other applications and domains.
Hence, while the motivation for this work originated from the dataset and database system used by *GeoMine*, it is abstractable and applicable to other systems, and other comparable workloads can occur for spatial analysis or data mining tasks in general.

Generalization Based on the defined challenges and limitations, our presented approach applies to workloads that fulfill the following properties: Queries that yield results for individual groups/partitions of data, such as geometric relationships between points within a definable subset of the data or aggregation functions on such subsets. Individual tables must be able to filter their tuples to the given partitions, ideally utilizing some index structure. To fully leverage the benefits of this approach, it is beneficial if the queries are computationally exhaustive or have computational complexity due to the data volume and, therefore, can strongly benefit by reducing the search space. While our approach is flexible in the query

[5] https://github.com/MPoppinga/PartitionCache

design, matches can only be found if queries or query parts are recurring. Further, our current approach is only suitable for read-only datasets, which we plan to extend in future work to appendable datasets.

3 Architecture

Our proposed architecture addresses the challenges of geometric pattern matching by introducing a caching strategy that acts as an index for matches in partitioned datasets. This approach is designed to improve the query processing time by reducing the search space to relevant partitions. In the case of *GeoMine*, these partitions are binding pockets. The overall process is illustrated in Fig. 3. In summary, while executing queries, the cache is populated with references to partitions that contain matching results for partial queries. Subsequent queries are recomposed, checked in the cache, and then rewritten to skip unnecessary data by restricting their search space to relevant partitions. In the remainder of this Section, we discuss the key components of this approach, which are:

1. Query de- and recomposition to create variants of queries that get separately executed and populate the cache. For this, all conjunctive conditions get separated, and subsets of these conditions are used to recompose into query variants.
2. Depending on the use case, not only elements of observed queries can be utilized, but also common and normalized constraints can be added, for example, wider distance ranges or frequently used attributes.
3. Hashing and execution of query variants by a concurrent process.
4. Storing references to all partitions that contain at least one valid match for each variant.
5. Cache utilization when executing a subsequent query. The approach recomposes queries and looks them up in the cache. To improve the cache hit rate, additional variants (ensuring query containment) can be generated to be checked in the cache.
6. Adding the intersection of all found partition keys to the original query. The cache acts as an inverted index to the relevant partitions to speed up the subsequent queries.

In Section 3.5, we show the workflow using an example from the geospatial domain and give a brief overview of the performance benefits.

3.1 Query Recomposition

To increase the probability of later use by subsequent changed queries, we relax the active query conditions. We achieve this by de- and recomposing the query as described in the following, resulting in a set of query variants. An example is shown in Fig. 4:

1. Ensuring the query is in CNF and labeling of the individual conditions with each involved table (see Fig. 4c)

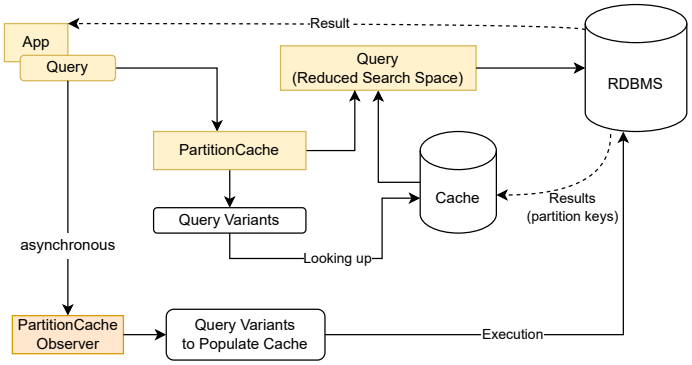


Fig. 3: Overview of our workflow. A concurrent process populates the cache, while active queries are limited to relevant partitions based on previously stored entries in the cache.

2. Create combinations of tables with at least one relationship, e.g. a distance specified to each other. Creating subsets of connected component graphs (see Fig. 4b) leads to many smaller queries combining different subsets of searched points.
3. Recompose queries for found combinations of tables
4. Normalize continuous attributes and distances by widening the ranges to defined values to add robustness against smaller changes in these values. For example, decreasing a value to the previous integer or a distance range to the next 100m bracket. (see Fig. 4d)
5. Adding or removing subqueries that limit the number of partitions
6. Create a stable representation of the query, for example, by normalizing aliases, ensuring stable syntax and ordering attributes. (see Fig. 4d)

In our evaluation of the *GeoMine* queries, between 20 and 250 variants were stored per query, depending on their complexity.

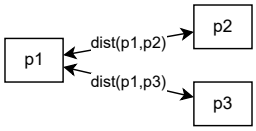
3.2 Cache Population

To populate the cache, we use an asynchronous approach to minimize the impact on the running query. For this, a concurrent process is monitoring the executed queries. For our workload, all queries are considered for caching. However, the approach can be adapted to consider only queries with a certain runtime, complexity, or maximum number of results to reduce the number of queries to execute. The concurrent process recomposes the queries and generates multiple variants of the query as described. Each query variant is executed on the database system or a replica, returning all partitions where at least one match was found. An example workflow is described in Section 3.5. The returned identifiers(*partition keys*) are then stored in the cache for each executed query variant. We use a SHA-1 hash of the

Query Q

```
SELECT *
FROM p AS p1, p AS p2, p AS p3
WHERE p1.p_key = p2.p_key
  AND p1.p_key = p3.p_key
  AND p1.element = 12
  AND p2.element = 6
  AND (p3.type = 3 OR p3.type = 2)
  AND distance(p1, p2) < 8.7
  AND distance(p1, p3) > 4.2;
```

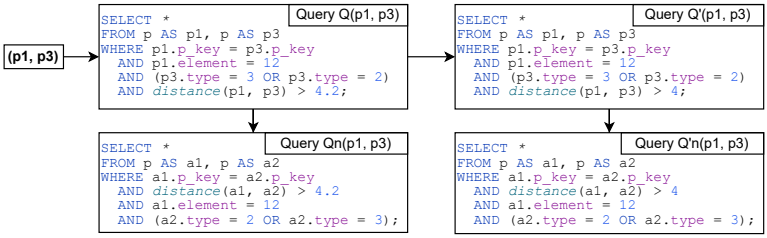
(a) Example Query Q



(b) Connected component graph of points

p1	[p1.element = 12]
p2	[p2.element = 6]
p3	[(p3.type = 3 OR p3.type = 2)]
(p1, p2)	[p1.element = 12, p2.element = 6, distance(p1, p2) < 8.7]
(p1, p3)	[p1.element = 12, (p3.type = 3 OR p3.type = 2), distance(p1, p3) > 4.2]
(p2, p3)	[p2.element = 6, (p3.type = 3 OR p3.type = 2)]
(p1, p2, p3)	[p1.element = 12, p2.element = 6 (p3.type = 3 OR p3.type = 2), distance(p1, p2) < 8.7, distance(p1, p3) > 4.2]

(c) Constraints for each combination of point pairs



(d) Creation of Q(p1,p3) and further an additional variant Q'(p1,p3) by relaxing a constraint to the next integer. Normalizing the query variants Q(p1,p3) and Q'(p1,p3) to ensure a stable representation. (Operations accordingly for all other valid point combinations of the connected component graph)

Fig. 4: Generating multiple variants for an example query Q by selecting subsets of involved points and rewriting conditions.

normalized query to represent the query.
This approach can also be extended by specifying common or predicted conditions, which would enrich the cache with more specific entries that may match in the future. Due to its exponential character, this variant creation can generate a high number of queries.
Limiting the number of resulting partition keys or a maximum running time may be beneficial to limit storage and computation requirements. Results of such queries that exceed this limit are ignored, and the queries are to be ignored for further caching.

3.3 Cache Utilization

To utilize the cache, the recomposition steps are executed for each query before executing it on the database, generating a set of hashes representing the query variants. Further variants with broader conditions are created for which we can guarantee query containment. Since checking hashes in the cache is relatively fast, this has only a minor impact on cache access

time but may restrict the search space further, for example, if the new query only added a condition or narrowed down a distance. In contrast to storing these query variants in the cache, this does not add additional storage requirements or computation time on creation. As all created query variants only removed or relaxed conditions and are therefore less restrictive, reducing the search space to the intersection of all matched partition keys is possible. Depending on the cache backend, the intersection of entries can be done directly within the caching database system, for example, utilizing *SINTER* (Redis) or *BIT_AND* (PostgreSQL). Only partitions that fulfill all query variant conditions can contain valid matches for the current query. As only the partition keys are stored, it is possible to use cached entries sourced from two or more original queries. The cache result usually contains false positives, which the relational database will remove while query processing.
To expose to the relational database system which parts of the dataset need to be searched, the set of partition keys can be added to the query as part of the selection, for example, by adding "`partition_key IN (1, 42, ...)`". Alternatively, if the cache is accessible via SQL, all cache usage can be included as an inlined subquery. This way, the list does not need to be materialized in the query. Extending the SQL query with the partition key list or the subquery can be done in different ways, depending on the optimizer's capabilities and the functionality of the underlying relational database providing the set of partition keys. For our use case, we determined rules for the way the query is extended, which depend on the number of partition keys and the presence of other subqueries. Using plain lists, temporary tables, common table expressions, or joins is possible, as described in Section 4.2.

3.4 Cache Storage

The purpose of the cache backend is to store the hash representation of each query with the information within which partitions matches of the query were found. The set of partition keys can either be stored as a set or a list-like structure, which are efficient if just a small number of hits is present per query, or as a bitarray, encoding in a fixed length for each partition key if a match is present by setting the individual bits to 1. We choose for our evaluation *RocksDB* to demonstrate embedded application-side storage, *Redis* for in-memory processing, and *PostgreSQL* for direct integration in the relational query. With all backends, we implemented a bitarray-based and a set-based variant.
Various cache eviction approaches are possible to limit storage consumption, like removing the *least recently used* entries. However, models that respect the size of the individual entries are open for evaluation in future work. For storage consumption and performance comparison, see Section 4.3.

3.5 Geospatial Example Workflow

We evaluated a simple pattern search on geospatial data to demonstrate our workflow. We imported an OpenStreetMap (OSM)[6] dataset for Germany and loaded all *pois* (points of interest) from the current OSM dataset. To create partitions, we used zip codes as partition keys. The dataset contains about 5 million points across about 8,200 zip codes.
We defined three spatial queries, defining specific patterns. *QueryA* is looking for an ice cream shop with a pharmacy and a supermarket of chain A nearby, as shown in Fig. 5. *QueryB* is looking for an ice cream shop with a swimming pool and a supermarket of chain B nearby. *QueryC* is looking for an ice cream shop with a swimming pool and a supermarket of chain A nearby. We use a PostgreSQL16.4 database with the PostGIS extension for spatial queries. The execution required between 0.24 and 0.28 seconds for each query on our test system. With the chosen conditions, *QueryA* yielded 145 results in 45 partitions, *QueryB* 44 results in 18 partitions, and *QueryC* 11 results in 4 partitions.

```
SELECT p1.name AS ice_cream_name, p2.name AS pharmacy_name,
p3.name AS supermark_name, p1.zipcode
FROM pois AS p1, pois as p2, pois as p3
WHERE p1.name LIKE '%Eis␣%'  AND p2.subtype = 'pharmacy'
  AND p3.subtype = 'supermarket' AND p3.name LIKE '%ALDI%'
  AND p2.zipcode = p1.zipcode AND p3.zipcode = p1.zipcode
  AND p3.zipcode = p2.zipcode AND ST_DWithin(p1.geom, p3.geom, 300)
  AND ST_DWithin(p1.geom, p2.geom, 400)
```

Fig. 5: Example for a spatial query, defining conditions between several points

Cache population Executing query *A* and *B*, they are added to a queue. The concurrently running process picks up the queries and recomposes them each into 6 query variants, each consisting of different combinations of the points. One variant for each *poi*, the two pairs connected via a distance query, and the whole query, which consists of all three *pois*. The recomposition, computation, and storage of all query variants of *Query A* and *B* takes about 4 seconds. The cache is filled with 11 hashes, each hash representing one subquery, holding all partition keys (zipcodes) where at least one matching result was found. As the query variant for the single point ice cream shop was present in *A* and *B*, it was only computed and stored once.

Cache Usage If a query is executed, all variants are created and checked if they are present in the cache.The intersected set of found partition keys can now be attached to the SQL query. Rerunning *Query A* and *B* again, the runtime is reduced to 0.11 and 0.10 seconds, including the required time for building the variants and checking the cache. The performance benefit originates from the much faster access to the cache, in this case, based on a Redis database, and limiting the search space for the SQL query, in contrast to needing to check all partitions. Furthermore, if *QueryC*, which was not stored in the cache, is executed, it also benefits from the cache. Six hashes are generated. As the combination of attributes is new, but the

[6] https://www.openstreetmap.org

individual parts already have been seen, 5 of the hashes are present (three pois and two of the pairs) in the cache. Intersecting these sets results in only four partitions to be searched, lowering the execution time to 0.08 seconds, which is about 1/3 of the original time without cache.

4 Evaluation

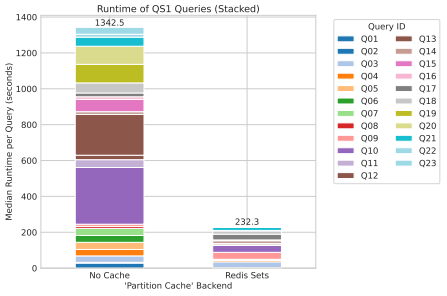
For the following evaluation, we used a lightweight Python-based variant of *GeoMine*. This variant allows the integration of caches and builds and executes the SQL query in the PostgreSQL database. The database we used is PostgreSQL 16.4. All data is stored on SSD, and the server is equipped with 400GiB RAM and 80 cores (PostgreSQL 128GB *sharedbuffers*, 16 *parallel workers per query*). The point table consists of the individual points and requires about 38GB, including indexes. Additional tables, including point-to-point interactions, direction vectors, and metadata, which were partially denormalized, amount to about 250GB with additional indexes of about 150GB. The Redis database was deployed on the same node as PostgreSQL. The *GeoMine* application and the databases are on different servers on the same network.
In the following, the term *caching* relates to our presented approach; internal buffering and shared memory caching of the databases were not affected. Unless otherwise stated, for evaluation, we used a set with 23 queries that required without caching between 5 seconds and 5 minutes each.

4.1 Query Runtime

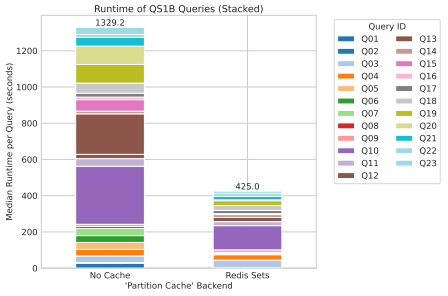
Fig. 1 shows the runtimes for subsequent runs of the same query. The cache was fully populated, and subsequent executions of the same queries were, on average, 5-6 times faster, as only the relevant partitions needed to be considered for the search in the relational database. There is a big difference between different queries, ranging from a 200x faster runtime to three queries that achieved worse performance(up to 0.3x performance). Here, queries that have results across a larger number of partitions show less benefit from our approach. Decreased performance is in the observed cases related to the optimizer generating a less optimal execution plan.
While Fig. 1 showed the runtimes with a fully populated cache, a more realistic workload is the queries with changed conditions. We compare the runtime using altered queries against the Redis backend. We first populated the cache with our set $QS1$ of 23 original queries. By adding all resulting partition keys for each query variant to the cache. Only query variants running for more than 15 minutes were skipped.
We created query sets $QS1B$ and $QS1C$, where we made more minor query changes so they were not fully cached. And $QS1D$ with two changes per query. We made the changes so that the number of results does not change much to prevent this from having a larger impact on performance without cache. In Fig. 6, it is visible that the runtime benefits from the caching

approach even with changes in the query. For $QS1D$, it shows benefits too, except on one outliner, which required significantly longer.

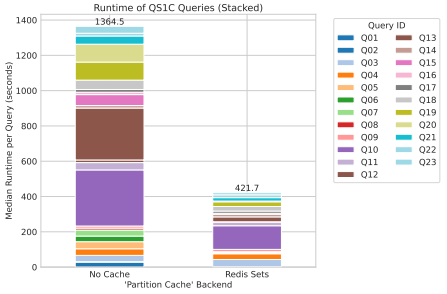
Further, we evaluated a small set of queries, in which we combined conditions from different



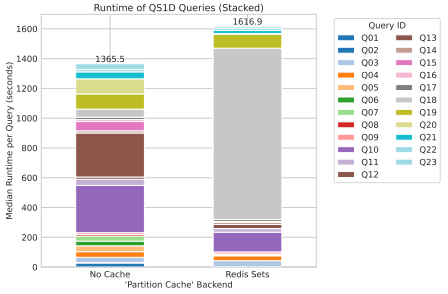
(a) Runtime improvement in a fully populated cache with QS1



(b) Runtime improvement with small distance changes in each query



(c) Runtime improvement with distance changes in each query



(d) Runtime improvement with two distance changes in each query

Fig. 6: Runtime improvements with small changes in each query for different variants.

seen queries and added new distances. The results were mixed here. In some cases, we achieved a high speedup, but in other cases, we had no performance benefit or even slightly worse performance than before. Especially if the cache returned many partitions (in the dimension of 25,000 entries and more), there were often no performance gains. Here, a better way to limit the computation to the desired spaces must be found so the query can fully benefit from a reduced search space.

4.2 Integration

Utilizing the cache by providing a list of partition keys via SQL proved to be challenging on its own, as depending on the queries, the PostgreSQL optimizer created query plans of different quality. Using some metrics, primarily the number of partition keys provided, the application can help the optimizer by changing between adding a simple condition if the

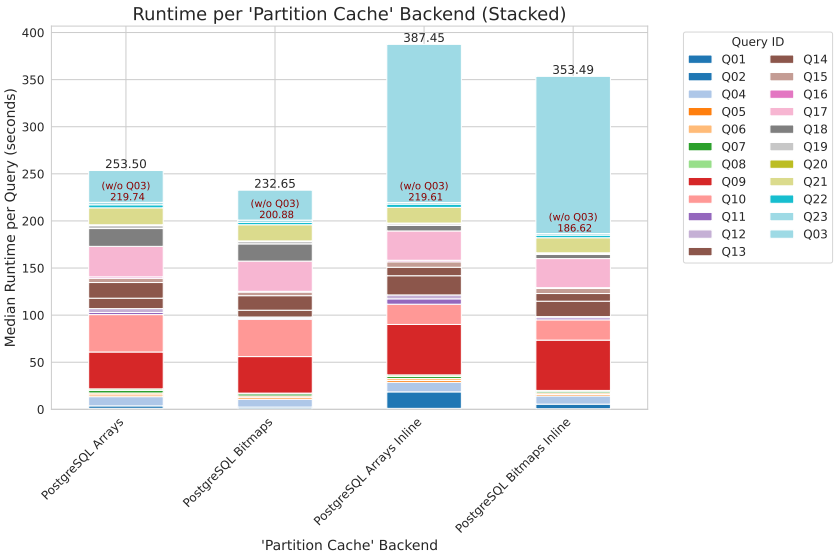


Fig. 7: 23 Queries with the materialization of the partition keys on the application side in SQL, in contrast to inlining the cache access as a subquery

points partition key is in a list, utilizing an inline temporary table with '`VALUES`', intersecting this table with other partition key queries, or forcing the optimizer to prioritize the partition keys by providing '`OFFSET 0`' to the query part. These hints significantly improve query performance in many cases while hindering the optimizer's flexibility, which should not be restricted under ideal conditions.

However, at the application level, information such as the number of matched partitions is not accessible if the cache access is inlined; such hints cannot be provided in these cases without materializing the partition keys first. For PostgreSQL-based cache variants, we measured the performance benefit of data locality and the removal of the need for a roundtrip of partition keys over the application and the materialization in SQL representation.

The cache performed similarly in most cases. In many cases, the query was slightly faster on the inlined subquery, benefiting from the data locality. In one case, the inlined query was even slower than removing the whole cache, as visible in Fig. 7; in this particular case, the maximum number of parallel workers planned by PostgreSQL for some operation seemed to slow down the original query parts. This shows that the way of exposing the cached partition keys to the relational database requires further work.

As shown in Fig. 1, the overall performance between different caching backends was very similar in contrast to the overall runtime.

4.3 Cache Performance

Redis and PostgreSQL allow the intersection of the sets and combining the bitmaps with a logical *and*. For RocksDB, the operation is on the application side; therefore, these results are affected by the implementation on the application side.

Storage Consumption We measured the storage consumption for each case after caching our 23 original queries $QS1$. The queries resulted in 1525 stored queries. To evaluate whether we can improve storage requirements, we removed in a separate step all queries that returned more than 50, 25, and 12.5 percent of our overall partition keys, as entries with many partition keys benefit less but consume more storage in their set representation. Fig. 8 does show the amount of storage used per backend. Disk usage was used for

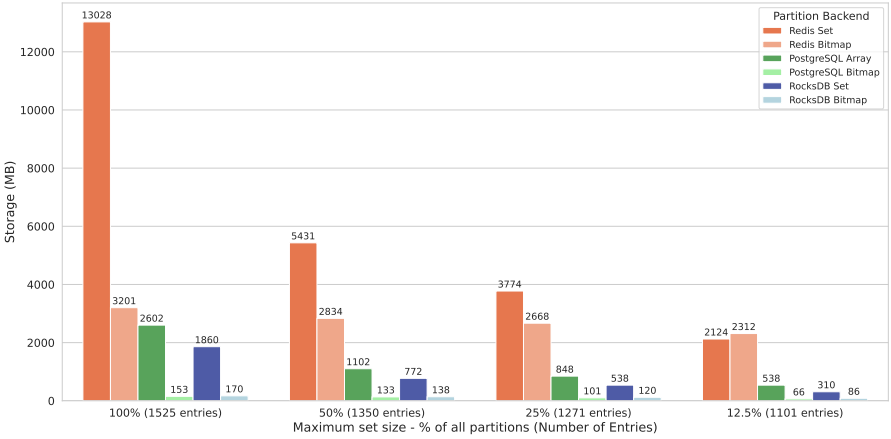


Fig. 8: Storage consumption for the different storage backends. With the removal of entries that exceed the given number of partition keys (in percent of the total number)

RocksDB, and the systems' self-reported values were used for Redis and PostgreSQL. Redis was consuming the most storage, in this case, in the working memory. In all cases, the bitarray representation was consuming less storage. However, the set representation benefits more if only queries with smaller sets of partition keys are stored. With about 2 million partitions, the bitarray per entry should consume 2Mb. PostgreSQL and RocksDB achieve less consumption due to compression, while Redis requires about 2MB per bitarray.

Intersection Speed While Redis consumes most memory, Fig. 9 shows that Redis, using sets, can intersect sets of given randomly selected hashes fast. For PostgreSQL and RocksDB, the bitmaps performed better than the array and set representations. The hash combination from our test queries ($QS1$) performed slightly differently than the randomly selected hashes, as the randomly selected often have no common partition keys across all selected sets, which can be used to return with an empty intersection set early.
The tests showed that storing only subqueries that have matches in a smaller share of all

partitions reduces storage consumption, especially for the set-based backends. At the same time, the performance of the approach is not directly affected. In our test, limiting it to queries with matches in 25 percent of partitions had no significant negative impact, as the number of partition keys returned only slightly increased.

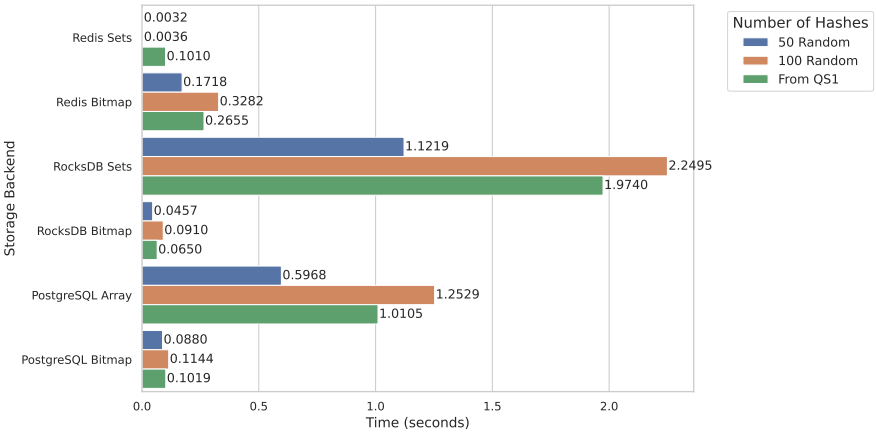


Fig. 9: Average required time to intersect sets within the cache.

However, for queries that only share a small number of conditions with previously cached entries, it has a bigger impact on the number of returned partition keys. However, the number of returned partition keys was quite large for such queries in either case. In many such cases, no performance benefit was measured, depending on the individual query and query plan of the optimizer. Therefore, keeping these unspecific entries in the cache is of limited use until we achieve a direct correlation between search space reduction and runtime.

Cache Population The required time to fully populate the cache depends on various aspects. Some unspecific query variants match many combinations, so their computation time grows accordingly. For such queries, we used a timeout of 15 minutes to prevent execution times that were too long. We limited the number of concurrent queries to 12 to reduce the impact on active user queries. In our base experiments, the overall creation to fully populate the cache took, on average, about 15 minutes per query for all their variants, including the small number of those that timed out.
Depending on the usage of the system, this requires an additional prioritization to ensure that query variants that help following queries most get executed first. Such prioritization is part of future work. Related, the order of execution is also of relevance while cache creation, as the presented cache can improve performance here too. Further, moving the cache creation to a replica or multiple replicas is possible, as this task can be highly parallelized.

5 Related Work

5.1 Database Systems

Various kinds of systems are available to store structured data. Relational systems work well with the flexibility of queries, especially as each query consists of a changing number of points that are selected by changing sets of attributes. Each datapoint row and attribute column may need to be considered for the search, hampering the advantages of alternative systems. The queries search for a precise match within their set parameters so that no results are lost or constraints are ignored. Alternative storage techniques like column-based databases [CK85], which are often used for analytical queries, have in our use case little benefit, as for each point, numerous different attributes may be specified for the selection, leading to varying subsets of rows and columns to be queried and compared. The ability of relational databases to relate different points to each other and enrich them with all required constraints, together with well-optimized joining and processing workflows, makes RDBMS a suitable choice for our workloads. Depending on the specific scenario, spatial extensions like the PostGIS[7] extension for PostgreSQL can be used, especially utilizing spatial index structures like r-trees. For *GeoMine*, using such extensions has not been beneficial so far. The dataset is separated into comparably small partitions, which reduces the effectiveness of spatial indexes, as only a subsection of points needs to be considered for distance evaluations. For our work, we choose PostgreSQL as it is a widely used and easily accessible relational system with a wide range of functionality. Further, it integrates with the existing software in the *GeoMine* deployment.

5.2 Indexing

Indexing, while highly effective for individual lookups on a datastructure such as a table, traditional approaches like b-trees [Co79] struggle if putting multiple relations in relation to each other. Structures like r-trees [Gu84] do perform better on ranges or multidimensional data fields. However, they still usually reference individual entries, pages, or ranges of entries within a single table and do not give the flexibility to reference a set of changing tables for differing queries without prior denormalization. Depending on the usage, different types of index structures are beneficial. They can also be extended to combine several search attributes and allow fast retrieval of values and value ranges. However, if multiple attributes are combined, the ordering becomes essential, and the index cannot necessarily be effectively used if the index does contain an attribute that is not queried. Bitindexes are especially useful for binary attributes, as they can efficiently store which rows' values are true or false by encoding them with a 0 or 1. They also allow a space-efficient representation of sets. This way, they also allow effective bitwise operations when multiple instances are combined, making them useful for intermediate data structures. Another type of index are

[7] https://postgis.net/

hash-based indexes in which an attribute or set of attributes is hashed and references the related entries, similar to hashmap datastructures. We used these as a part of our solution.

A variant of indexing can be observed in several applications where index structures are combined with custom fingerprints. In these cases, fingerprint-like structures are defined, and then the whole dataset is searched for such structures and stored where to find them. These fingerprints are often hand-crafted or created by domain-based rules. This approach is effective if enough specific structures are known, but it does not work well if patterns are not fully known beforehand. In contrast to our approach, they are pretty inflexible.

5.3 Data Skipping

Data skipping [Ta20] describes ways to read only relevant data to fulfill a query. Niu et al. [Ni21] presented a provenance-based approach called *provenance-based data skipping (PBDS)*. They derive provenance sketches from individual queries and use them to append additional rules to the SQL query, limiting the data to search for the database system. They focus on HAVING and top-k queries but use a similar approach to ours by enriching active queries with constraints that limit the search space on read-only datasets. While both, our approach and *PBDS*, aim to accelerate queries by identifying and skipping irrelevant data, they differ in their methods and targeted workloads. *PBDS* uses queries to capture and propagate provenance sketches based on value ranges, identifying which table fragments contain relevant data for that query, making *PBDS* more targeted at accelerating subsequent instances of the same parameterized query. In contrast, our partition cache breaks queries into variants, caches the partition identifiers of matches, and intersects these to reduce the search space for future queries. This allows for flexible reuse of cache results across different queries, even from additionally created variants based on predictive query conditions. Our approach targets disjunct partitions where no semantic connection can be derived, and therefore, using generated rules to combine different partitions into range-based representations is less efficient. The increased flexibility of executing and storing partition keys in our approach comes with higher storage consumption than rule-based approaches like *PBDS* have. However, this flexibility allows our approach to target the observed workload in our *GeoMine* domain, working well in environments with millions of partitions that cannot be easily described using rule-based annotations.

5.4 Denormalization

For read-heavy workloads, denormalization is often a well-suited choice to increase performance. Especially if several datapoints that are put in relation to each other are present, denormalization can reduce the expensive join operations or distance calculations. For this, expensive parts of the query are precomputed and stored so that the results are faster retrievable while executing the query. For geospatial domains, there exist approaches [Fa18]

using graphs as a datastructure, denormalizing all relevant distances.
However, in the context of geometric pattern matching, the number of potential relationships grows exponentially with the existing points, making a full denormalization impractical for larger datasets. In addition to all point pairs, angles between more than two points would also need to be precomputed to represent geometric attributes. This includes precomputing all distances between all potential points for graph databases or property graphs in general. Other denormalization techniques, such as materialized views [CY12], which are generally available in many relational database management systems, have limitations. Usually, they are created manually, which is not feasible in our use case as each search is individual and consists of a changing set of attributes, relations, and the number of tables to be joined. As materialized views denormalize data for a specific query, using this would also exponentially increase the number of views. Our approach leverages flexibility, as it can store sets for any query and happens transparently, and no manual addition of views is necessary.

5.5 Caching

In addition to denormalizing the data by precomputing intermediate results, it is also possible to store once-computed data in a cache. Caching usually happens either on a lower level, where individual relations are cached to be read faster again, which often happens transparent to the user, or on a higher level, where the complete result of a given query is stored for a specified time. If the same query is executed a second time, the cached result can be returned without redoing the more expensive computational tasks. This is often used, for example, in web applications for queries that yield the same result for all users for a certain time. Of special interest here is ensuring that the results remain valid if the underlying data changes. For read-only datasets, as in our case and often in the scientific domain, this is a, in general, negligible issue as it can be reduced to cache eviction to ensure a limitation of the storage consumption of the cache. Caching often uses key-value stores or similar systems to cache the resulting value for a given input [At12]. Data is often stored in memory, as fast results are desired and persistence is not prioritized. Such caching can happen in an application, middleware, or database side; the principles remain the same. Result caching is fast but struggles with changing queries. If a query changes, a classical caching approach has no concept of which of the results are still relevant for a changed query. As only the partition keys are stored in our approach, we can work with changing queries and even use entries originating from multiple previous queries. Further, storage consumption can be enormous if results are denormalized and stored in their entirety, especially if individual generic points are in multiple valid combinations of the resultset.

5.6 Query decomposition

SQL queries consist of various components that can be separated into a tree-like structure of individual predicates [WY76]. This process is called query decomposition and is used

for different types of use cases. It is an important step for the creation of the query plan. In terms of polyglot systems, they allow, for example, to push down specific operators to other databases or database structures [Gl22].
There are approaches for SBQL (stack-based Query Language) to use their internal object-oriented model to decompose a query and cache specific intermediate results [CS11]. Here, the intermediate results are cached, which allows direct access to the results. In contrast, storing only partition keys is beneficial if many results are found within a partition, as in this case, only the reference to a partition needs to be stored, not all intermediate results.

6 Conclusions and Future Work

This paper presents a novel architecture to accelerate geometric pattern-matching queries in large partitioned datasets. Our approach caches the results of partial queries in an index-like structure, enabling subsequent queries to benefit from the cached results and significantly reduce their search space. Different cache entries can be combined for subsequent queries as our approach stores references to partitions in which the partial queries yield results. Through the experimental evaluation on a real-world protein-ligand dataset, we could show the effectiveness of our approach. We were able to reduce the search space for queries as long as parts of a query had been seen before. Moreover, we observed average speedups of up to 5-6x for repeated queries and 3x for queries with partially changed conditions. Furthermore, we validated the transferability of our approach by applying it to a geospatial dataset, highlighting its potential for broader impact beyond the life sciences domain. Limitations of our work include the need for further optimizations in integrating the cached results with the relational database query, particularly for cases where the search space reduction is not significant. However, our current approach of adding the partition keys to the SQL query shows high flexibility as it can be performed in various environments. Computing the query variants creates additional load on the database system. For our use case, this was not an issue as our target was to improve the runtime of individual queries rather than the overall throughput of the system, as a low number of concurrent users utilize such research systems.

We plan to further evaluate and extend our approach to additional workloads. This includes appenable datasets and workloads using aggregation functions on individual partitions. Further looking ahead, promising directions for future work include exploring techniques for cache eviction, utilizing bloom filters to reduce storage consumption, or dynamic clustering partitions for more efficiency. The integration with other optimization techniques is of further interest, such as materialized views. Further, the effectiveness of our approach on even larger datasets and additional domains is an area worth exploring. This includes domains aside from geometric pattern matching where partitioned datasets and comprehensive search queries are found, e. g., explorative searches in extensive document datasets.
In conclusion, our work makes a significant contribution to accelerating geometric pattern matching queries in partitioned datasets. Our approach enables faster exploration and analysis of datasets by leveraging query recomposition and caching of partition identifiers, enabling researchers and domain experts to derive valuable insights more efficiently.

Bibliography

[At12] Atikoglu, Berk; Xu, Yuehai; Frachtenberg, Eitan; Jiang, Song; Paleczny, Mike: 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. pp. 53–64, 2012.

[Be00] Berman, Helen M; Westbrook, John; Feng, Zukang; Gilliland, Gary; Bhat, Talapady N; Weissig, Helge; Shindyalov, Ilya N; Bourne, Philip E: The protein data bank. Nucleic acids research, 28(1):235–242, 2000.

[CA24] Carpenter, Kristy A; Altman, Russ B: Databases of ligand-binding pockets and protein-ligand interactions. Computational and Structural Biotechnology Journal, 2024.

[Ch22] Chen, Yue; Feng, Kaiyu; Cong, Gao; Kiah, Han Mao: Example-based spatial pattern matching. Proceedings of the VLDB Endowment, 15(11):2572–2584, 2022.

[CK85] Copeland, George P.; Khoshafian, Setrag N.: A decomposition storage model. SIGMOD Rec., 14(4):268–279, may 1985.

[Co79] Comer, Douglas: Ubiquitous B-tree. ACM Computing Surveys (CSUR), 11(2):121–137, 1979.

[CS11] Cybula, Piotr; Subieta, Kazimierz: Decomposition of SBQL queries for optimal result caching. In: 2011 Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 841–848, 2011.

[CY12] Chirkova, Rada; Yang, Jun: Materialized Views. Foundations and Trends® in Databases, 4(4):295–405, 2012.

[Di24] Diedrich, Konrad; Ehrt, Christiane; Graef, Joel; Poppinga, Martin; Ritter, Norbert; Rarey, Matthias: User-centric design of a 3D search interface for protein-ligand complexes. Journal of Computer-Aided Molecular Design, 38(1):23, 2024.

[Fa13] Fard, Arash; Nisar, M Usman; Ramaswamy, Lakshmish; Miller, John A; Saltz, Matthew: A distributed vertex-centric approach for pattern matching in massive graphs. In: 2013 IEEE International Conference on Big Data. IEEE, pp. 403–411, 2013.

[Fa18] Fang, Yixiang; Cheng, Reynold; Cong, Gao; Mamoulis, Nikos; Li, Yun: On Spatial Pattern Matching. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). pp. 293–304, 2018.

[GER23] Graef, Joel; Ehrt, Christiane; Rarey, Matthias: Binding site detection remastered: enabling fast, robust, and reliable binding site detection and descriptor calculation with DoGSite3. Journal of Chemical Information and Modeling, 63(10):3128–3137, 2023.

[Gl22] Glake, Daniel; Kiehn, Felix; Schmidt, Mareike; Panse, Fabian; Ritter, Norbert: Towards Polyglot Data Stores–Overview and Open Research Questions. arXiv preprint arXiv:2204.05779, 2022.

[Gr22] Graef, Joel; Ehrt, Christiane; Diedrich, Konrad; Poppinga, Martin; Ritter, Norbert; Rarey, Matthias: Searching Geometric Patterns in Protein Binding Sites and Their Application to Data Mining in Protein Kinase Structures. Journal of Medicinal Chemistry, 65(2):1384–1395, 2022.

[Gu84] Guttman, Antonin: R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on Management of data. pp. 47–57, 1984.

[He03] Hendlich, Manfred; Bergner, Andreas; Günther, Judith; Klebe, Gerhard: Relibase: design and development of a database for comprehensive analysis of protein–ligand interactions. Journal of molecular biology, 326(2):607–620, 2003.

[HS08] He, Huahai; Singh, Ambuj K: Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 405–418, 2008.

[Ju21] Jumper, John; Evans, Richard; Pritzel, Alexander; Green, Tim; Figurnov, Michael; Ronneberger, Olaf; Tunyasuvunakool, Kathryn; Bates, Russ; Žídek, Augustin; Potapenko, Anna et al.: Highly accurate protein structure prediction with AlphaFold. nature, 596(7873):583–589, 2021.

[KMS16] Kotthoff, Lars; McCreesh, Ciaran; Solnon, Christine: Portfolios of subgraph isomorphism algorithms. In: International Conference on Learning and Intelligent Optimization. Springer, pp. 107–122, 2016.

[Ko16] Korb, Oliver; Kuhn, Bernd; Hert, Jérôme; Taylor, Neil; Cole, Jason; Groom, Colin; Stahl, Martin: Interactive and versatile navigation of structural databases. Journal of Medicinal Chemistry, 59(9):4257–4266, 2016.

[Kr17] Krause, Alexander; Ungethüm, Annett; Kissinger, Thomas; Habich, Dirk; Lehner, Wolfgang: Asynchronous Graph Pattern Matching on Multiprocessor Systems. In (Kirikova, Marite; Nørvåg, Kjetil; Papadopoulos, George A.; Gamper, Johann; Wrembel, Robert; Darmont, Jérôme; Rizzi, Stefano, eds): New Trends in Databases and Information Systems - ADBIS 2017 Short Papers and Workshops, AMSD, BigNovelTI, DAS, SW4CH, DC, Nicosia, Cyprus, September 24-27, 2017, Proceedings. volume 767 of Communications in Computer and Information Science. Springer, pp. 45–53, 2017.

[Kr19] Krause, Alexander; Ungethüm, Annett; Kissinger, Thomas; Habich, Dirk; Lehner, Wolfgang: NeMeSys - Energy Adaptive Graph Pattern Matching on NUMA-based Multiprocessor Systems. In (Grust, Torsten; Naumann, Felix; Böhm, Alexander; Lehner, Wolfgang; Härder, Theo; Rahm, Erhard; Heuer, Andreas; Klettke, Meike; Meyer, Holger, eds): Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme"(DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings. volume P-289 of LNI. Gesellschaft für Informatik, Bonn, pp. 537–541, 2019.

[Ku16] Kuhn, Bernd; Guba, Wolfgang; Hert, Jerome; Banner, David; Bissantz, Caterina; Ceccarelli, Simona; Haap, Wolfgang; Koörner, Matthias; Kuglstatter, Andreas; Lerner, Christian et al.: A real-world perspective on molecular design: Miniperspective. Journal of medicinal chemistry, 59(9):4087–4102, 2016.

[MWM15] McCune, Robert Ryan; Weninger, Tim; Madey, Greg: Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Computing Surveys (CSUR), 48(2):1–39, 2015.

[Ni21] Niu, Xing; Glavic, Boris; Liu, Ziyu; Li, Pengyuan; Gawlick, Dieter; Krishnaswamy, Vasudha; Liu, Zhen Hua; Porobic, Danica: Provenance-based data skipping. Proceedings of the VLDB Endowment, 15(3), 2021.

[Po23] Poppinga, Martin; Graef, Joel; Diedrich, Konrad; Rarey, Matthias; Ritter, Norbert: Database and Workflow Optimizations for Spatial-Geometric Queries in GeoMine. In: Lernen, Wissen, Daten, Analysen (LWDA) Conference Proceedings. pp. 86–97, 2023.

[Sc22] Schöning-Stierand, Katrin; Diedrich, Konrad; Ehrt, Christiane; Flachsenberg, Florian; Graef, Joel; Sieg, Jochen; Penner, Patrick; Poppinga, Martin; Ungethüm, Annett; Rarey, Matthias: Proteins*Plus*: a comprehensive collection of web-based molecular modeling tools. Nucleic Acids Res., 50(W1):611–615, 2022.

[Ta20] Ta-Shma, Paula; Khazma, Guy; Lushi, Gal; Feder, Oshrit: Extensible Data Skipping. In: 2020 IEEE International Conference on Big Data (Big Data). pp. 372–382, 2020.

[WY76] Wong, Eugene; Youssefi, Karel: Decomposition—a strategy for query processing. ACM Trans. Database Syst., 1(3):223–241, sep 1976.

[ZLY09] Zhang, Shijie; Li, Shirong; Yang, Jiong: GADDI: distance index based subgraph matching in biological networks. In: Proceedings of the 12th international conference on extending database technology: advances in database technology. pp. 192–203, 2009.