Database and Workflow Optimizations for Spatial-Geometric Queries in GeoMine

Martin Poppinga[1,2], Joel Graef[2], Konrad Diedrich[2], Matthias Rarey[2] and Norbert Ritter[1]

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

Abstract
Addressing computational problems in science often involves customized algorithmic approaches, which can lead to overlooking well-established solutions in data management and storage. When scientific datasets grow, these customized approaches may struggle to query data efficiently. Effective data management is essential for ensuring accurate and fast analysis of scientific data. Describing changes in the *GeoMine* software, this paper highlights the potential for improvements in data-driven science.

GeoMine enables spatial-geometric searches in three-dimensional molecular space, facilitating tasks such as pharmaceutical drug discovery by finding similar geometric patterns in protein-ligand complexes. The original *GeoMine* application utilized a relational database solely for fundamental data storage and combined it with a tailored algorithmic pattern-matching strategy, leaving room for improvements. This work presents a technical overview of database and workflow optimizations in *GeoMine* to handle the increasing data size. Our improvements focus on moving the main computational tasks from the application level to the database system and optimizing the database utilization. A new query design, better utilization of indexes, and optimizations in textual queries led to a 15x speedup in our experiments, reducing the mean runtime of queries to under 8 seconds.

The presented improvements are essential for *GeoMine* to be offered as a service-oriented web application. The success of these improvements highlights the significance of database optimization in science, demonstrating the potential and necessity of proper data management.

Keywords
Database optimization, query optimization, data management, databases for bioinformatics

1. Introduction

Mining huge datasets is a central task in research. Analyzing molecular interactions between proteins and small organic molecules is essential for understanding disease treatments and advancing medical research. This includes searching for spatial similarities and geometric arrangements, which can provide vital insights into the functional aspects of proteins. Results can be used for further research, for example, in pharmaceutical drug discovery or biotechnology [1]. With the growth of accessible datasets, searching for patterns in this data becomes

LWDA'23: Lernen, Wissen, Daten, Analysen. October 09–11, 2023, Marburg, Germany
martin.poppinga@uni-hamburg.de (M. Poppinga); graef@zbh.uni-hamburg.de (J. Graef); diedrich@zbh.uni-hamburg.de (K. Diedrich); rarey@zbh.uni-hamburg.de (M. Rarey); norbert.ritter@uni-hamburg.de (N. Ritter)
0000-0001-8529-8376 (M. Poppinga); 0000-0001-8327-4936 (J. Graef); 0000-0001-8171-0888 (K. Diedrich); 0000-0002-9553-6531 (M. Rarey); 0000-0002-1502-1395 (N. Ritter)

increasingly challenging [2, 3]. Besides the continuous growth of available experimental data, machine-learning-based structure predictions add millions of new structural models [4].

GeoMine [3] is an application enabling a visual-guided geometric pattern search of molecular data in three-dimensional space. It is embedded in the *proteins.plus*[1] server [5], a collection of different web-based tools for various tasks in protein-based research. The server is a free service based on publicly available datasets handling over half a million page requests per year. The back end of GeoMine was derived in prior work from the *PELIKAN* application developed in the same group [6], which was utilizing a custom algorithmic approach for query processing. With the Protein Data Bank (PDB) [7] as a fast-growing dataset underlying GeoMine and the shift from a desktop application to a server-based approach, GeoMine required an overhaul of the original query workflow to maintain the ability to provide results in a fast manner.

With this work, we investigate the potential of adopting a database-driven architecture, focusing on the database as the main part of query execution and reducing application-side processing. We were able to reduce the mean runtime in our experiments from about 2 minutes per query to less than 8 seconds, utilizing changes in the workflow and database optimizations. As we present in this work, a substantial performance enhancement has been achieved by shifting to a more database-centric method.

The paper is organized as follows: Section 2 provides an overview of the field of work, the data structure, and the query design; Section 3 details the improvements made to the query workflow and database optimizations; Section 4 presents and discusses the experimental results; Section 5 concludes the paper and outlines future work.

2. Background and Related Work

2.1. Data Management and Storage

Data management in scientific research involves the systematic collection, organization, storage, and sharing of data to facilitate its reusability and ensure the reproducibility of research findings. In the context of our work, which focuses on querying structured data sets, the storage aspect is particularly important. In the scientific domain, many existing applications are designed for single-user usage, often locally storing data in various formats or utilizing object stores with limited retrieval possibilities [8, 9]. For structured data, *Relational Database Management Systems* (RDBMS) are the most commonly used systems, providing robust and efficient solutions. Commonly, embedded systems are used, such as SQLite [10] for applications with smaller or medium-sized data sizes or DuckDB [9] for analytical workloads. For *Online Transaction Processing* (OLTP) workloads which require fast query performance and regular updates, server-based RDBMS are a popular choice. Large analytical queries are often served by designated *Online Analytical Processing* (OLAP) systems such as data warehouses, which are often proprietary solutions. For handling large-scale semi-structured datasets, NoSQL systems are frequently used, with columnar and graph databases being popular for analytical queries. The choice of data management and storage solutions is crucial to ensure efficient processing, reduced resource consumption, and accurate and fast analysis of scientific data.

[1]https://proteins.plus

PostgreSQL GeoMine utilizes PostgreSQL [11], a robust and widely accessible open-source database management system. As multiple users can access a web-based application such as GeoMine at the same time, the ability of a client-server-based database system to handle multiple queries efficiently in parallel is required. PostgreSQL's widespread adoption [12] enables cloud-agnostic hosting on every major platform since most cloud platforms offer PostgreSQL solutions or other PostgreSQL-compatible scalable databases. Additionally, setting up on-premise or local instances is straightforward. PostgreSQL is suited for OLTP and also OLAP workloads [13]. The required workloads here can be depicted in the area of OLAP, given the potential complexity of the designed queries. However, given the use case of an interactive search mask for a web service, fast responses are a requirement. PostgreSQL's efficient query planning and extensibility for additional approaches (e.g., PostGIS [14] for spatial data or Citus [15] for distributed and columnar storage) make it a suitable foundation for GeoMine's use case.

2.2. Protein-Ligand Interactions and Binding Pockets

Protein-ligand interactions are of particular interest in biomolecular and pharmaceutical research. Ligands are small molecules that can interact and bind to the generally much larger proteins. Protein complexes can contain multiple pockets of varying sizes, partly containing ligands. Drug molecules used as pharmaceuticals are generally designed to target specific proteins. Researchers can gain valuable insights by investigating specific three-dimensional structures and searching for potential candidates to bind with these proteins.

Protein Data Bank The PDB [7, 2], established in 1971, is a comprehensive repository of 3D structural data of proteins and nucleic acids. The structural information is primarily obtained through experimental methods, predominantly X-ray crystallography, from research facilities worldwide [2]. As a freely available resource, the PDB has become vital for research in various fields by providing atomic-scale structural insights for drug design and understanding biological processes, containing more than 200,000 structures as of April 2023. Further, with the advantage of *Computed Structure Models*, which are protein structure predictions, for example, by *AlphaFold2* [4], additional datasets with about 1,000,000 structures are available now [2].

2.3. GeoMine

Discovering similar structures across distinct complexes or finding molecules that bind to a specific pocket of interest is a major task in medical research. GeoMine is able to construct comprehensive databases derived from the PDB and supports exploring these databases with a web-based search interface. [3]

The preprocessing and database creation procedures employ components of the NAOMI library [16]. For example, pockets are classified in a complex preprocessing pipeline when constructing the database [3]. Central components are the DoGSite algorithm [17], which identifies empty binding pockets within protein structures, and the calculation of interactions [18].

The central part of the search and unique key feature is the ability to specify geometric properties, for instance, distances and angles between any points, such as atoms. Further, point properties can be specified, such as an atom's chemical element and interactions between points. This way, precise structural motifs (structural patterns) in protein-ligand complexes can be searched. While GeoMine's predecessor *PELIKAN* was a single-user application based on an integrated *SQLite* [10] database, the *GeoMine* back end is aimed at a server-focused architecture. In the initial development of GeoMine [3], the query execution capabilities of PELIKAN were extended for new functionality but were not changed in structure to adapt to the new architecture.

Database Design For our experiments in Section 4, we used a PostgreSQL15 database created with the PDB dataset from October 2022. For querying the dataset, the database can be considered read-only. The database requires approximately 165GB of disk space.

For the geometric search, we focus on two tables. The first table, the *point table*, comprises all atoms and other definable points, such as the center of aromatic rings. It contains 340,716,693 searchable entries. These points are distributed across 1,382,853 distinct pockets, which serve as containers for groups of points. The largest pocket identified in our dataset contains 20,306 points, while the smallest pocket only holds 9 points. Each entry in the point table has a unique identifier, references the containing pocket, and contains various other fields with properties per point. Some properties, such as the accessible surface area of an atom, are floating point numbers. Other attributes, such as the chemical element, contain only a few distinct values, represented as integers or short strings.

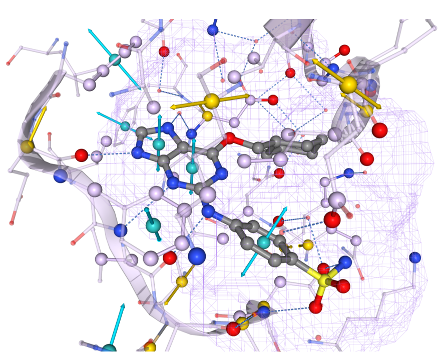
The second table, the *interaction table*, stores pre-calculated interactions [18]. These interactions represent noteworthy connections between two points, for example, hydrogen bonds. 13,018,225 point pairs are stored here.

Query Creation When creating a query, users can specify multiple constraints. The most fundamental categories encompass *Textual and Numerical Searches*, wherein metadata filters at the protein structure or pocket level can be defined. Users can directly pre-select several structures or create various filters, such as the minimum number of particular chemical elements or a certain molecular weight range for the ligand. It also enables filtering using patterns that describe a local environment using the chemical substructure language SMARTS strings [19].

The central search element and origin of GeoMine's name are geometry-based searches. To build the query, users may interactively select points in the web front end [21] (see Figure 1), utilizing an arbitrary PDB file as a template structure or define them without a template.

Users may select an arbitrary number of points, which can be filtered based on different properties. Moreover, the specification of distance ranges between two points and angles between specified distances is possible. Further, interactions between points, as stored in the interaction table, can be added to the query. Together they resemble an atomic substructure, which will be searched for. Each pocket can be examined individually as the interactions between one ligand and an individual pocket in a protein are of interest.

Query Execution The initial approach for query execution was first described for the predecessor tool PELIKAN by Inhester et al. [6]. The most significant enhancement for the runtime in developing the original GeoMine approach — utilizing a PostgreSQL database instead of SQLite — did not change the workflow of the searching process. The approach remained mostly


(a) View of a pocket (violet mesh) in structure *1H1S*

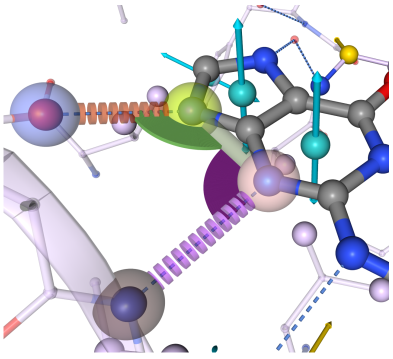
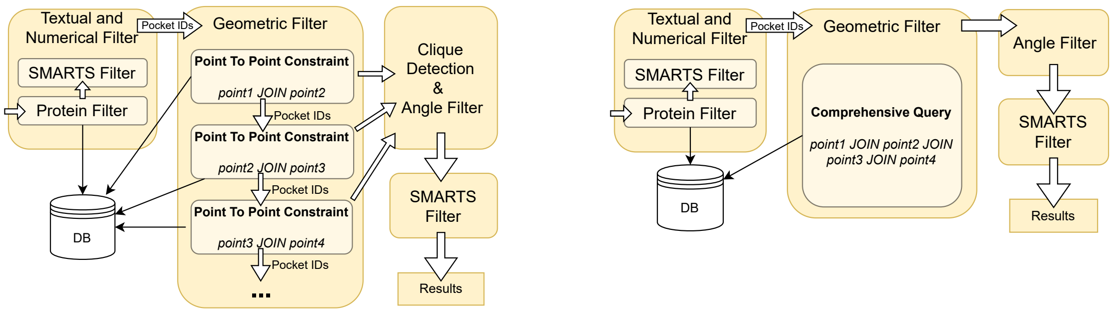

(b) Specifing points, distances, and angles for the query

Figure 1: GeoMine's three-dimensional view of a binding pocket based on the *NGL viewer* [20]. Users can interactively select atoms and other points and specify distances and interactions between them to generate the query. Here, a pocket around a ligand (bold bonds) is shown, together with the surrounding atoms of the protein.

algorithmic focused, with all major computational steps performed within the application (see Figure 2a), as the original PELIKAN software was designed to be a standalone desktop application. In the original approach of GeoMine [3], four major steps were performed strictly sequentially for each query to filter the potential results:

1. Textual and Numerical Constraints - A filter eliminates all proteins and pockets that do not meet specified properties or do not correspond to a given restrictive SMARTS filter. This step yields a list of all matching proteins and their pockets.
2. Obtaining all point pairs - For each point pair in the query, all possible results are returned, and distances, as well as interaction constraints, are checked.
3. Clique detection - An algorithm reconstructs the coherent component graph for all obtained point pairs and checks all defined angle constraints.
4. Less restrictive SMARTS filters for points were applied to the now-generated results.

Steps two and three of the query processing presented particular challenges. All point and point pair constraints were queried individually in the database. Since a single constraint for a point pair is often not very specific, it leads to big intermediate results. Only by chaining several constraints the number of points is sufficiently reduced. The need to cross-verify each point with all matching points in its pocket demanded significant computational resources, especially if the filter for the points were unspecific. The list of potential pockets needed to be recreated for each pair, as only pockets which contained results in prior pair subqueries remained in the search space. This caused the search to be strictly sequential and required the serialization and deserialization of long pocket-ID lists for the SQL WHERE clauses. As the application and database system are separate processes or running on separate servers, the required repeated transfer of these lists also affected the performance. Because some point-to-point constraints were specific (less frequent in the dataset) and others were unspecific (frequent in the dataset), a hand-crafted scoring function was utilized to estimate the best ordering of queries, starting



(a) Original workflow

(b) Improved workflow

Figure 2: The original and the improved processing workflow of GeoMine (Simplified) for a given search

with the most specific queries to reduce the search space early [6]. Although this improved the join order in many cases, it had the disadvantage of preventing the database system from executing classical optimizations, such as parallelism and join order optimization.

Further, an additional algorithm was required since the results from the preceding steps consisted only of point pairs. The *Bron-Kerbosch algorithm* [22], a graph-based backtracking algorithm for clique detection, was used. This algorithm recursively verified whether all discovered point pairs constituted a complete graph and checked for angle constraints. This demanded substantial computational effort, taking several hours on large potential result sets.

3. Optimizations

This research aims to achieve optimal performance and ease of setup across various environments. Alongside the contributions of this work, the application has transitioned to a containerized setup for cloud environments. The optimizations presented in this work are essential for facilitating the deployment of a scalable application. In this section, we will distinguish between the *original approach* in GeoMine [3] and the *improved approach* we present in this work. The yielded results for each query remained identical.

3.1. Optimizing SQL Queries

The most significant change from the original approach was the redesign of the SQL query generation. Sequential processing of each constraint within a query led to severely limited query-level parallelism and long processing times as described in Section 2.3. Therefore, all SQL queries are now designed to make use of PostgreSQL's internal planning and optimization. In contrast to the original approach, where each point-to-point constraint was queried separately, a single comprehensive query containing all attributes and constraints for geometrical patterns is now constructed, see Figure 2b. This reduces overhead by eliminating the need to repeatedly serialize extensive lists of pocket IDs or create temporary tables. To achieve this, the point table joins itself as often as points were specified in the query, usually 5-15 times. As a match occurs inside a single pocket, we only need to join points within the same pocket. With information about the distribution of properties like the chemical element, the RDBMS can estimate which part of the query restricts the search space the most and improve the join order. The original approach required running the checks on all points within all remaining pockets, not being able to skip points that were not matched in earlier subqueries. Intermediate results now remain within the database system and do not require serialization for application transfer. Additionally, merging all constraints (points, distances, and interactions) into one query eliminates the need for clique detection, as the output of the RDBMS is a connected and valid result.

Among all the geometric properties, only the angle checking between point pairs remains a separate step in the application, as this increases the complexity of the query without showing the benefits of an early reduced search space in our tests. Textual and numerical filters remain in a separate query to allow prior filtering, as SMARTS patterns require in-application processing. Allowing the RDBMS to determine the join order and the parallel execution resulted in a significant speedup of benchmark queries. The results are detailed in Section 4.

3.2. Enhanced Utilization of PostgreSQL Indexes

In the original approach, a single extensive index structure was created, covering 15 out of 17 table columns. Although PostgreSQL allows for the construction of multi-column indexes with a large number of attributes, these structures are only effective in certain situations due to their size and depending on the used attributes. However, using multiple single-column indexes and allowing PostgreSQL to combine them as recommended in the documentation [23] did not achieve the desired performance improvement.

Only the combination of several attributes could substantially reduce the number of yielded points. The best-found solution for our workload was a balanced compromise between index size and utilization, including only the most frequently used columns in a multi-column index. We identified two separate cases for index usage. Firstly, the earliest scheduled subquery focused solely on the attributes, disregarding their pocket, in cases without textual and numerical filters. Secondly, an index for subsequent subqueries was needed to filter for pocket IDs required for the join. In almost all instances, the optimizer determined to filter for the pocket ID in the second subquery. In some instances, a parallel index scan was performed. Filtering by the pocket ID reduced the search space best in these cases since the most restrictive subquery had already been executed as the first scheduled subquery. Therefore, we introduced a second index with the pocket identifier positioned first in the index. For both structures, we utilized PostgreSQL's default *B-Tree index* as other index structures seemed not beneficial in our tests. As pockets usually contain only a few hundred points, spatial indexes, like r-trees provided by PostGIS [14], did not provide the desired benefits. Filtering points and calculating all distances performed better in our tests than spatial operations due to the overhead of utilizing a spatial column. Index creation only needed a few minutes, but additional indexes for specific queries would no longer fit into the filesystem read cache and reduce performance.

3.3. Improving Text Search

The initial step of the workflow involves filtering structures based on textual and numerical attributes. These filters target various properties, the most important being the PDB identifiers used to select a pre-defined or user-defined subset of protein structures. A short alphanumeric

Table 1
Experiment overview. Showing enabled improvments between baseline ***ex01*** and all improvments ***ex06***

Improvement	**ex01**	ex02	ex03	ex04	ex05	**ex06**	ex07	ex08	ex09	ex10
Index Improvement		x				x	x	x	x	
No Wildcards			x			x	x	x		x
New Query Design				x		x	x		x	x
No ILIKE					x	x		x	x	x

code identifies each structure.

Previously, an SQL *ILIKE* (case insensitive match) statement with a wildcard match at the beginning and end of the string was executed to check for the desired properties. For the PDB codes, we could make two changes. We could discard the wildcards in the query unless explicitly desired, which enables the utilization of a search index. And as the codes are not case-sensitive, we can replace the *ILIKE* with a *LIKE*, allowing for a case-sensitive search and resulting in a substantial speedup, as demonstrated in Section 4.

4. Evaluation and Discussion

4.1. Methods

To evaluate the impact of each modification suggested for GeoMine, several experiments were derived from the original GeoMine approach *ex01* (see Table 1). Experiments *ex02* to *ex05* each contain only one of the improvements, *ex06* contains all improvements, while experiments *ex07* to *ex10* contain all except one. This way, we show which change impacts the performance most, as different improvements benefit from each other.

For evaluating the performance across different workloads, we used a set of nine queries already used in previous work [3], designed to highlight available features, show examples for common applications and estimate the runtime of different patterns common in GeoMine practical applications. They emitted between 2 and 7117 results.

We used a PostgreSQL15 database system. All data was stored on an SSD. Unless otherwise specified, a dedicated server with 400GiB RAM and 80 Cores was used (PostgreSQL 128GB *sharedbuffers*, 16 *parallel workers*). Podman [24] was used to deploy the system. Each experiment was repeated five times. The GeoMine application was executed on the same node as the PostgreSQL database. We configured PostgreSQL to utilize less memory than available, as GeoMine required a high amount of working memory for some workloads. Additionally, we conducted tests on commodity systems by employing two setups (*small*/*medium*) using virtual servers. Both setups stored data on SSDs and were equipped with 12 cores and 24GB RAM, resp. 18 cores and 48GB RAM.

4.2. Results

Figure 3 shows the mean runtime of the nine test queries for each experiment as depicted in Table 1. Each change led to better performance, with the highest performance gain occurring

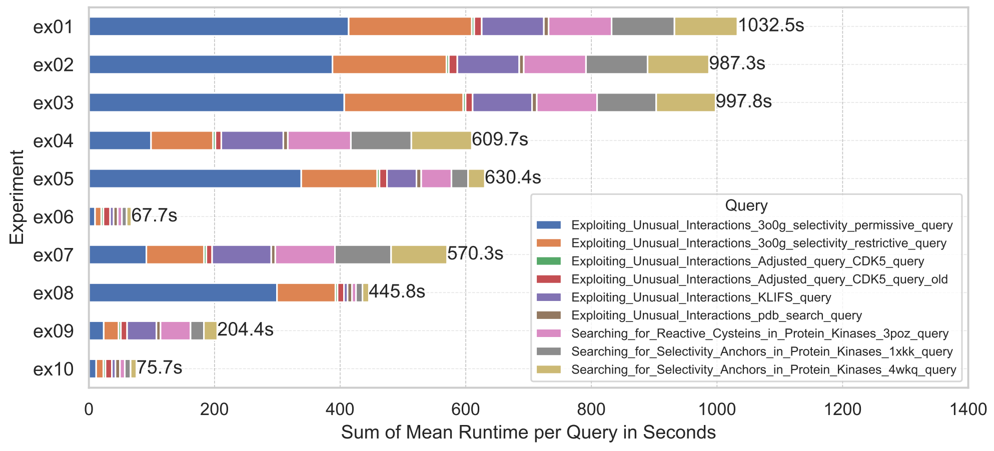


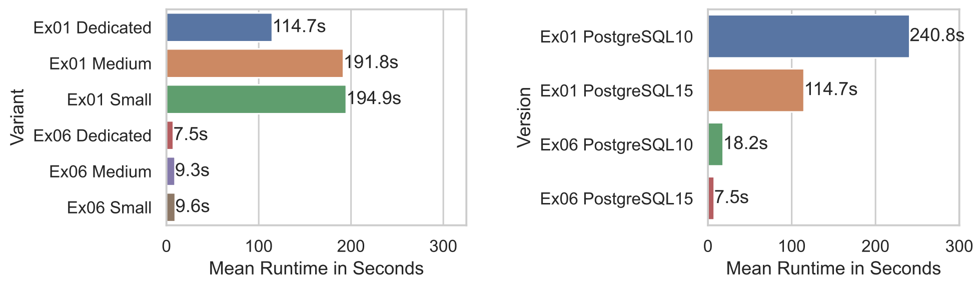
Figure 3: The sum of mean runtimes in seconds for each experiment as described in Section 4.1. Each color represents one distinct query

when all changes were applied together. The required time for performing all nine queries decreased from 1033sec of the original approach (*ex01*) to 68sec with all improvements (*ex06*).

The new query design (*ex04*) had the most substantial impact on performance, particularly visible in the long-running queries. Also, the transition from the ILIKE to the LIKE statement notably reduced runtime. The performance gain is most noticeable on the medium-running queries containing a long list of PDB IDs for a preselection. The experiments 02 and 03, the *new index* and *no wildcards* in the PDB ID selection showed only a small improvement. However, experiment 09, which contains all changes except the wildcard improvement, shows that it has an impact on the overall runtime, presumably benefiting from the switch to the LIKE statement. The changes in index structures showed less impact than expected, demonstrating that PostgreSQL can handle indexes with an inflated number of columns. However, the performance was drastically worse if no index was used or index structures did not combine multiple attributes. For instance, combining one index per attribute led to an increase of the sum of the mean runtimes from 68sec (*ex06*) to 134sec.

Unspecific Queries Some of the used queries include a protein filter to reduce the number of searched pockets. When removing these filters and searching the whole dataset, the original approach reached its set limits (needing more than 100GB RAM or 1h time) on some of these and other queries with less restrictive geometric filters. With the improved approach, some queries with extensive intermediate results could now be computed for the first time, often within minutes.

Alternative Setups As large database instances are not always accessible, for example, due to cost constraints in cloud environments, we also conducted our experiment on two smaller virtual servers. As shown in Figure 4a, the performance gains were also visible on these smaller server instances. These tests were performed on shared hardware, so they can only show a general trend rather than precise comparative data. However, they demonstrate the feasibility



(a) Virtual servers and dedicated hardware (b) PostgreSQL in Version 10 and 15

Figure 4: Mean runtimes in alternative configurations of experiment 01 and 06

of processing on shared virtual servers. Additionally, we observed a substantial speedup while transitioning from PostgreSQL10 to PostgreSQL15 as displayed in Figure 4b. Combined with our improvements, we achieved a speedup factor of 32.

5. Conclusion and Future Work

GeoMine is a unique application for geometric searches in large collections of protein-ligand complexes with high relevance for life-science research. We showed that it was possible to achieve a large speedup on our query processing by moving major parts of the processing from a custom-written logic inside the software to a PostgreSQL database system. Additionally, different approaches in database optimization contributed to further performance gain. Overall, these achievements are critical for the practical use of the system handling the growing dataset. Some queries could be executed for the first time on our setup due to these changes. In this work, we focused on optimizations of the database and query design. We demonstrated the substantial benefits of database optimizations in scientific applications, achieving a fifteen-fold speedup in GeoMine. Coupled with a halving of the runtime through the use of a newer PostgreSQL version, we managed to reduce the average runtime from minutes to seconds.

Looking ahead, we plan to explore additional database paradigms, such as distributed or column-based systems, and establish schema changes for further optimizations. The caching of intermediate results, as well as determining the join order by extended statistics or by utilizing machine learning, may potentially provide additional benefits. This way, we aim to achieve even better performance for searching scientific data with a service-oriented web service.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research as part of CompLS and de.NBI (031L0172 and 031L0105).

References

[1] T. Inhester, M. Rarey, Protein–ligand interaction databases: advanced tools to mine activity data and interactions on a structural level, WIREs Computational Molecular Science 4 (2014) 562–575. doi:10.1002/wcms.1192.

[2] S. K. Burley, C. Bhikadiya, C. Bi, S. Bittrich, H. Chao, L. Chen, e. a. Craig, RCSB Protein Data Bank (RCSB.org): delivery of experimentally-determined PDB structures alongside one million computed structure models of proteins from artificial intelligence/machine learning, Nucleic Acids Research 51 (2022).

[3] 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.

[4] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, et al., Highly accurate protein structure prediction with alphafold, Nature 596 (2021) 583–589. doi:10.1038/s41586-021-03819-2.

[5] K. Schöning-Stierand, K. Diedrich, R. Fährrolfes, F. Flachsenberg, A. Meyder, E. Nittinger, R. Steinegger, M. Rarey, Proteins plus: interactive analysis of protein–ligand binding interfaces, Nucleic acids research 48 (2020) W48–W53. doi:10.1093/nar/gkaa235.

[6] T. Inhester, Mining of Interaction Geometries in Collections of Protein Structures, Ph.D. thesis, Universität Hamburg, 2017.

[7] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, P. E. Bourne, The Protein Data Bank, Nucleic Acids Research 28 (2000) 235–242. doi:10.1093/nar/28.1.235.

[8] C. Tenopir, N. M. Rice, S. Allard, L. Baird, J. Borycz, L. Christian, B. Grant, R. Olendorf, R. J. Sandusky, Data sharing, management, use, and reuse: Practices and perceptions of scientists worldwide, PloS one 15 (2020) e0229003.

[9] M. Raasveldt, H. Mühleisen, Data management for data science-towards embedded analytics., in: CIDR, 2020.

[10] R. D. Hipp, SQLite, 2020. URL: https://www.sqlite.org/.

[11] The PostgreSQL Global Development Group, Postgresql: The world's most advanced open source relational database, 2023. URL: https://www.postgresql.org.

[12] solid IT gmbh, Db-engines ranking, 2023. URL: https://db-engines.com/en/ranking.

[13] A. Conrad, Database of the year: Postgres, IEEE Software 38 (2021) 130–132. doi:10.1109/MS.2021.3089730.

[14] The PostGIS Development Group, Postgis, 2023. URL: https://postgis.net.

[15] U. Cubukcu, O. Erdogan, S. Pathak, S. Sannakkayala, M. Slot, Citus: Distributed postgresql for data-intensive applications, in: Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, 2021, p. 2490–2502. doi:10.1145/3448016.3457551.

[16] S. Urbaczek, A. Kolodzik, J. R. Fischer, T. Lippert, S. Heuser, I. Groth, T. Schulz-Gasch, M. Rarey, Naomi: On the almost trivial task of reading molecules from different file formats, Journal of Chemical Information and Modeling 51 (2011) 3199–3207. doi:10.1021/ci200324e.

[17] J. Graef, C. Ehrt, M. Rarey, Binding site detection remastered: Enabling fast, robust,

and reliable binding site detection and descriptor calculation with dogsite3, Journal of Chemical Information and Modeling 63 (2023) 3128–3137. doi:10.1021/acs.jcim.3c00336, pMID: 37130052.

[18] 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.

[19] I. Daylight Chemical Information Systems, Smarts-a language for describing molecular patterns, 2007.

[20] A. S. Rose, A. R. Bradley, Y. Valasatava, J. M. Duarte, A. Prlić, P. W. Rose, NGL viewer: web-based molecular graphics for large complexes, Bioinformatics 34 (2018) 3755–3758. doi:10.1093/bioinformatics/bty419.

[21] K. Diedrich, J. Graef, K. Schöning-Stierand, M. Rarey, GeoMine: interactive pattern mining of protein–ligand interfaces in the Protein Data Bank, Bioinformatics 37 (2020) 424–425. doi:10.1093/bioinformatics/btaa693.

[22] C. Bron, J. Kerbosch, Algorithm 457: finding all cliques of an undirected graph, Communications of the ACM 16 (1973) 575–577.

[23] PostgreSQL 15, Documentation, 2023. URL: https://www.postgresql.org/docs/15/.

[24] Containers, podman, 2023. URL: https://podman.io/.